Document Number:	P0970R1
Date:	2018-05-06
Project:	C++ Extensions for Ranges, Library Evolution Working Group

Reply to:	Eric Niebler eric.niebler@gmail.com

Better, Safer Range Access Customization Points

Note: this is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.

Contents

1 General [intro]

> "Begin at the beginning, the King said, very gravely, and go on till you come to the end: then stop."
>
> —*Lewis Carroll*

1.1 Revision history [intro.history]

1.1.1 Revision 1 [intro.history.rev1]

This paper has been rebased on P0789R3 ([1]) and P0896R1 ([2]) (and hence C++20).

1.2 Scope [intro.scope]

1 This document suggests improvements to the range access customization points (`begin`, `end`, *et.al.*) of ISO/IEC TS 21425:2017, otherwise known as the Ranges TS. The improvements suggested here apply to P0896 R1, "Merging the Ranges TS" ([2]), and to P0789 R3, "Range Adaptors and Utilities" ([1]).

1.3 Problems with `std::ranges::begin` [intro.problem]

1 For the sake of compatibility with `std::begin` and ease of migration, `std::`~~`experimental::`~~`ranges::begin` accepted rvalues and treated them the same as `const` lvalues. This behavior was deprecated because it is fundamentally unsound: any iterator returned by such an overload is highly likely to dangle after the full expression that contained the invocation of `begin`.

2 Another problem, and one that until recently seemed unrelated to the design of `begin`, was that algorithms that return iterators will wrap those iterators in `std::`~~`experimental::`~~`ranges::dangling<>` if the range passed to them is an rvalue. This ignores the fact that for some range types — `std::span`, `std::string_view`, and P0789's `subrange`, in particular — the iterator's validity does not depend on the range's lifetime at all. In the case where ~~a p~~an rvalue ~~`subrange<>`~~of one of the above types is passed to an algorithm, returning a wrapped iterator is totally unnecessary.

3 The author believed that to fix the problem with `subrange` and `dangling` would require the addition of a new trait to give the authors of range types a way to say whether its iterators can safely outlive the range. That felt like a hack~~, and that feeling was reinforced by the author's inability to pick a name for such a trait that was sufficiently succint and clear~~.

1.4 Suggested Design [intro.design]

1 We recognized that by removing the deprecated default support for rvalues from the range access customization points, we made design space for range authors to opt-in to this behavior for their range types, thereby communicating to the algorithms that an iterator can safely outlive its range type. This eliminates the need for `dangling` when passing an rvalue `std::span`, `std::string_view`, or `subrange`, an important usage scenario.

2 This improved design would be both safer and more expressive: users should be unable to pass to ~~`std2`~~ `std::ranges::begin` any rvalue range unless its result is guaranteed to not dangle.

3 The mechanics of this change are subtle. There are two typical ways for making a type satisfy the `Range` concept:

1. Give the type `begin()` and `end()` member functions (typically not lvalue reference-qualified), as below:

```
struct Buffer {
  char* begin();
  const char* begin() const;
  char* end();
  const char* end() const;
};
```

2. Define `begin` and `end` as free functions, typically overloaded for `const` and non-`const` lvalue references, as shown below:

```
struct Buffer { /*...*/ };

char* begin(Buffer&);
const char* begin(const Buffer&);
char* end(Buffer&);
const char* end(const Buffer&);
```

4 These approaches offer few clues as to whether iterators yielded from this range will remain valid even the range itself has been destroyed. With the first, `Buffer{}.begin()` compiles successfully. Likewise, with the second, `begin(Buffer{})` is also well-formed. Neither yields any useful information.

5 The design presented in this paper takes a two-pronged approach:

1. ~~std2~~std::ranges`::begin(E)` never considers `E.begin()` unless `E` is an lvalue.

2. ~~std2~~std::ranges`::begin(E)` will consider an overload of `begin(E)` found by ADL, looked up in a context that (a) does not include ~~std2~~std::ranges`::begin`, and (b) includes the following declaration:

```
// "Poison pill" overload:
template <class T>
void begin(T&&) = delete;
```

This approach gives ~~std2~~std::ranges`::begin` the property that, for some rvalue expression `E` of type `T`, the expression ~~std2~~std::ranges`::begin(E)` will not compile unless there is a free function `begin` findable by ADL that specifically accepts rvalues of type `T`, and that overload is prefered by partial ordering over the general `void begin(T&&)` "poison pill" overload.

6 This design has the following benefits:

(6.1) — No iterator returned from ~~std2~~std::ranges`::begin(E)` can dangle, even if `E` is an rvalue expression.

(6.2) — Authors of simple view types for which iterators may safely outlive the range (like P0789's `subrange<>`) may denote such support by providing an overload of `begin` that accepts rvalues.

7 Once ~~std2~~std::ranges`::begin`, `end`, and friends have been redefined as described above, the `safe_iterator_t` alias template can be redefined to only wrap an iterator in `dangling<>` for a `Range` type `R` if ~~std2~~std::ranges`::begin(std::declval<R>())` is ill-formed. In code:

```
template <Range R~~, class = void~~>
struct __safe_iterator {
  using type = dangling<iterator_t<R>>;
};
template <class R>
  requires requires (R&& r) { std::ranges::begin((R&&) r); }
struct __safe_iterator<R~~, void_t<decltype(std2::begin(declval<R>()))>~~> {
  using type = iterator_t<R>;
};
```

```
template <Range R>
using safe_iterator_t = typename __safe_iterator<R>::type;
```

Now algorithms that accept `Range` parameters by forwarding reference and that return iterators into that range can simply declare their return type as `safe_iterator_t<R>` and have that iterator wrapped only if it can dangle.

1.5 References [intro.refs]

1 The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2017, *Programming Languages - C++*

(1.2) — ISO/IEC TS 21425:2017, *Technical Specification - C++ Extensions for Ranges*

ISO/IEC 14882:2017 is herein called the *C++ Standard* and ISO/IEC TS 21425:2017 is called the *Ranges TS*.

Part I

Changes to P0896 R1 [P0896]

25 Strings library [strings]

25.4 String view classes [string.view]

25.4.1 Header <string_view> synopsis [string.view.synop]

[Editor's note: change the `<string_view>` header synopsis as follows:]

```
namespace std {
  // ??, class template basic_string_view
  template<class charT, class traits = char_traits<charT>>
  class basic_string_view;

  // 25.4.3, basic_string_view range access
  template<class charT, class traits>
    constexpr auto begin(basic_string_view<charT, traits> x) noexcept;
  template<class charT, class traits>
    constexpr auto end(basic_string_view<charT, traits> x) noexcept;

  // ... as before
}
```

[Editor's note: After [string.view.template], insert the following subsection and renumber all following subsections.]

25.4.3 basic_string_view range access [string.view.range_access]

1 [*Note:* The following two range access functions are provided for interoperability with ~~std2~~`std::ranges::begin` and ~~std2~~`std::ranges::end`. — *end note*]

```
template<class charT, class traits>
  constexpr auto begin(basic_string_view<charT, traits> x) noexcept;
```

2 *Returns:* `x.begin()`.

```
template<class charT, class traits>
  constexpr auto end(basic_string_view<charT, traits> x) noexcept;
```

3 *Returns:* `x.end()`.

29 Ranges library [ranges]

29.5 Range access [range.access]

1 In addition to being available via inclusion of the <~~std2/~~range> header, the customization point objects in 29.5 are available when <~~std2/~~iterator> is included.

29.5.1 `begin` [range.access.begin]

1 The name `begin` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::begin(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — ~~`ranges::begin(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::begin(E)` behaves similarly to `std::begin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]~~

(1.2) — ~~Otherwise,~~ `(E) + 0` if `E` has array type (6.7.2) and is an lvalue.

(1.3) — Otherwise, if `E` is an lvalue, *`DECAY_COPY`*`((E).begin())` if it is a valid expression and its type `I` meets the syntactic requirements of `Iterator<I>`. If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, *`DECAY_COPY`*`(begin(E))` if it is a valid expression and its type `I` meets the syntactic requirements of `Iterator<I>` with overload resolution performed in a context that includes the following declarations:

```
template <class T> void begin(T&&) = delete;
template <class T> void begin(initializer_list<T>&&) = delete;
```

and does not include a declaration of `::std::ranges::begin`. If `Iterator` is not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, `::std::ranges::begin(E)` is ill-formed.

2 [*Note:* Whenever `::std::ranges::begin(E)` is a valid expression, its type satisfies `Iterator`. — *end note*]

29.5.2 `end` [range.access.end]

1 The name `end` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::end(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — ~~`ranges::end(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::end(E)` behaves similarly to `std::end(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]~~

(1.2) — ~~Otherwise,~~ `(E) + extent_v<T>` if `E` has array type (6.7.2) ~~T~~and is an lvalue.

(1.3) — Otherwise, if `E` is an lvalue, *`DECAY_COPY`*`((E).end())` if it is a valid expression and its type `S` meets the syntactic requirements of `Sentinel<S, decltype(::std::ranges::begin(E))>`. If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, *`DECAY_COPY`*`(end(E))` if it is a valid expression and its type `S` meets the syntactic requirements of `Sentinel<S, decltype(::std::ranges::begin(E))>` with overload resolution performed in a context that includes the following declarations:

```
template <class T> void end(T&&) = delete;
template <class T> void end(initializer_list<T>&&) = delete;
```

and does not include a declaration of `::std::ranges::end`. If `Sentinel` is not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, `::std::ranges::end(E)` is ill-formed.

2 [*Note:* Whenever `::std::ranges::end(E)` is a valid expression, the types of `::std::ranges::end(E)` and `::std::ranges::begin(E)` satisfy `Sentinel`. — *end note*]

29.5.3 `cbegin` [range.access.cbegin]

1 The name `cbegin` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::cbegin(E)` for some subexpression `E` of type `T` is expression-equivalent to:

(1.1) — `::std::ranges::begin(static_cast<const T&>(E))` if `E` is an lvalue.

(1.2) — Otherwise, `::~~std2~~std::ranges::begin(static_cast<const T&&>(E))`.

2 ~~Use of `::std2::cbegin(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `::std2::cbegin(E)` behaves similarly to `std::cbegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]~~

3 [*Note:* Whenever `::std::ranges::cbegin(E)` is a valid expression, its type satisfies `Iterator`. — *end note*]

29.5.4 `cend` [range.access.cend]

1 The name `cend` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::cend(E)` for some subexpression `E` of type `T` is expression-equivalent to:

(1.1) — `::std::ranges::end(static_cast<const T&>(E))` if `E` is an lvalue.

(1.2) — Otherwise, `::~~std2~~std::ranges::end(static_cast<const T&&>(E))`.

2 ~~Use of `::std2::cend(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `::std2::cend(E)` behaves similarly to `std::cend(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]~~

3 [*Note:* Whenever `::std::ranges::cend(E)` is a valid expression, the types of `::std::ranges::cend(E)` and `::std::ranges::cbegin(E)` satisfy `Sentinel`. — *end note*]

29.5.5 `rbegin` [range.access.rbegin]

[Editor's note: This changes `rbegin` and `rend` into proper customization points, with "`rbegin`" and "`rend`" looked up via argument-dependent lookup. The idea is to support types for which reverse iterators can be implemented more efficiently than with `reverse_iterator`, and which might want to overload `rbegin` and `rend` for rvalue arguments. A simple example might be a `reverse_subrange` type, which would want to overload `rbegin` and `rend` to return the unmodified underlying iterator and sentinel (as opposed to `begin` which would return `reverse_iterators`).]

1 The name `rbegin` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::rbegin(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — ~~`ranges::rbegin(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `::std2::rbegin(E)` behaves similarly to `std::rbegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]~~

(1.2) — ~~Otherwise~~If E is an lvalue, *DECAY_COPY*((E).rbegin()) if it is a valid expression and its type I meets the syntactic requirements of Iterator<I>. If Iterator is not satisfied, the program is ill-formed with no diagnostic required.

(1.3) — Otherwise, *DECAY_*COPY(rbegin(E)) if it is a valid expression and its type I meets the syntactic requirements of Iterator<I> with overload resolution performed in a context that includes the following declaration:

```
template <class T> void rbegin(T&&) = delete;
```

and does not include a declaration of ::~~std2~~std::ranges::rbegin. If Iterator is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, make_reverse_iterator(::std2::end(E)) if both ::std::ranges::begin(E) and ::std::ranges::end(E) are valid expressions of the same type I which meets the syntactic requirements of BidirectionalIterator<I> (??).

(1.5) — Otherwise, ::std::ranges::rbegin(E) is ill-formed.

2 [*Note:* Whenever ::std::ranges::rbegin(E) is a valid expression, its type satisfies Iterator. — *end note*]

29.5.6 rend [range.access.rend]

1 The name rend denotes a customization point object (20.1.4.2.1.6). The expression ::std::ranges::rend(E) for some subexpression E is expression-equivalent to:

(1.1) — ~~ranges::rend(static_cast<const T&>(E)) if E is an rvalue of type T. This usage is deprecated. [*Note:* This deprecated usage exists so that ::std2::rend(E) behaves similarly to std::rend(E) as defined in ISO/IEC 14882 when E is an rvalue. — *end note*]~~

(1.2) — ~~Otherwise~~If E is an lvalue, *DECAY_COPY*((E).rend()) if it is a valid expression and its type S meets the syntactic requirements of Sentinel<S, decltype(::std::ranges::rbegin(E))>. If Sentinel is not satisfied, the program is ill-formed with no diagnostic required.

(1.3) — Otherwise, *DECAY_*COPY(rend(E)) if it is a valid expression and its type S meets the syntactic requirements of Sentinel<S, decltype(~~std2~~std::ranges::rbegin(E))> with overload resolution performed in a context that includes the following declaration:

```
template <class T> void rend(T&&) = delete;
```

and does not include a declaration of ~~std2~~std::ranges::rend. If Sentinel is not satisfied, the program is ill-formed with no diagnostic required.

(1.4) — Otherwise, make_reverse_iterator(::std::ranges::begin(E)) if both ::std::ranges::begin(E) and ::std::ranges::end(E) are valid expressions of the same type I which meets the syntactic requirements of BidirectionalIterator<I> (??).

(1.5) — Otherwise, ::std::ranges::rend(E) is ill-formed.

2 [*Note:* Whenever ::std::ranges::rend(E) is a valid expression, the types of ::std2::rend(E) and ::std2::rbegin(E) satisfy Sentinel. — *end note*]

29.5.7 `crbegin` [range.access.crbegin]

1 The name `crbegin` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::crbegin(E)` for some subexpression `E` of type `T` is expression-equivalent to:

(1.1) — `::std::ranges::rbegin(static_cast<const T&>(E))` if `E` is an lvalue.

(1.2) — Otherwise, ~~`std2`~~`::std::ranges::rbegin(static_cast<const T&&>(E))`.

2 ~~Use of `ranges::crbegin(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::crbegin(E)` behaves similarly to `std::crbegin(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]~~

3 [*Note:* Whenever `::std::ranges::crbegin(E)` is a valid expression, its type satisfies `Iterator`. — *end note*]

29.5.8 `crend` [range.access.crend]

1 The name `crend` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::crend(E)` for some subexpression `E` of type `T` is expression-equivalent to:

(1.1) — `::std::ranges::rend(static_cast<const T&>(E))` if `E` is an lvalue.

(1.2) — Otherwise, ~~`std2`~~`::std::ranges::rend(static_cast<const T&&>(E))`.

2 ~~Use of `ranges::crend(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::crend(E)` behaves similarly to `std::crend(E)` as defined in ISO/IEC 14882 when `E` is an rvalue. — *end note*]~~

3 [*Note:* Whenever `::std::ranges::crend(E)` is a valid expression, the types of `::std::ranges::crend(E)` and `::std::ranges::crbegin(E)` satisfy `Sentinel`. — *end note*]

29.6 Range primitives [range.primitives]

1 In addition to being available via inclusion of the `<`~~`std2/`~~`range>` header, the customization point objects in 29.6 are available when `<`~~`std2/`~~`iterator>` is included.

29.6.1 `size` [range.primitives.size]

1 The name `size` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::size(E)` for some subexpression `E` with type `T` is expression-equivalent to:

(1.1) — *`DECAY_COPY`*`(extent_v<T>)` if `T` is an array type (6.7.2).

(1.2) — Otherwise, *`DECAY_COPY`*`(`~~`static_cast<const T&>(`~~`E`~~`)`~~`.size())` if it is a valid expression and its type `I` satisfies `Integral<I>` and `disable_sized_range<remove_cvref_t<T>>` (??) is `false`.

(1.3) — Otherwise, *`DECAY_COPY`*`(size(`~~`static_cast<const T&>(`~~`E`~~`)`~~`))` if it is a valid expression and its type `I` satisfies `Integral<I>` with overload resolution performed in a context that includes the following declaration:

```
template <class T> void size(const T&&) = delete;
```

and does not include a declaration of `::std::ranges::size`, and `disable_sized_range<remove_cvref_t<T>>` is `false`.

(1.4) — Otherwise, *`DECAY_COPY`*`(::std::ranges::`~~`c`~~`end(E) - ::std::ranges::`~~`c`~~`begin(E))`, except that `E` is only evaluated once, if it is a valid expression and the types `I` and `S` of `::std::ranges::`~~`c`~~`begin(E)` and `::std::ranges::`~~`c`~~`end(E)` meet the syntactic requirements of `SizedSentinel<S, I>` (??) and `ForwardIterator<I>`. If `SizedSentinel` and `ForwardIterator` are not satisfied, the program is ill-formed with no diagnostic required.

(1.5) — Otherwise, `::std::ranges::size(E)` is ill-formed.

2 [*Note:* Whenever `::std::ranges::size(E)` is a valid expression, its type satisfies `Integral`. — *end note*]

29.6.2 `empty` [range.primitives.empty]

1 The name `empty` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::empty(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — `bool((E).empty())` if it is a valid expression.

(1.2) — Otherwise, `::std::ranges::size(E) == 0` if it is a valid expression.

(1.3) — Otherwise, `bool(::std::ranges::begin(E) == ::std::ranges::end(E))`, except that `E` is only evaluated once, if it is a valid expression and the type of `::std::ranges::begin(E)` satisfies `ForwardIterator`.

(1.4) — Otherwise, `::std::ranges::empty(E)` is ill-formed.

2 [*Note:* Whenever `::std::ranges::empty(E)` is a valid expression, it has type `bool`. — *end note*]

29.6.3 `data` [range.primitives.data]

1 The name `data` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::data(E)` for some subexpression `E` is expression-equivalent to:

(1.1) — ~~`ranges::data(static_cast<const T&>(E))` if `E` is an rvalue of type `T`. This usage is deprecated. [*Note:* This deprecated usage exists so that `ranges::data(E)` behaves similarly to `std::data(E)` as defined in the C++ Working Paper when `E` is an rvalue. — *end note*]~~

(1.2) — ~~Otherwise~~If `E` is an lvalue, *`DECAY_COPY`*`((E).data())` if it is a valid expression of pointer to object type.

(1.3) — Otherwise, `::std::ranges::begin(E)` if it is a valid expression of pointer to object type.

(1.4) — Otherwise, `::std::ranges::data(E)` is ill-formed.

2 [*Note:* Whenever `::std::ranges::data(E)` is a valid expression, it has pointer to object type. — *end note*]

29.6.4 `cdata` [range.primitives.cdata]

1 The name `cdata` denotes a customization point object (20.1.4.2.1.6). The expression `::std::ranges::cdata(E)` for some subexpression `E` of type `T` is expression-equivalent to:

(1.1) — `::std::ranges::data(static_cast<const T&>(E))` if `E` is an lvalue.

(1.2) — Otherwise, ~~`std2`~~`::std::ranges::data(static_cast<const T&&>(E))`.

2 ~~Use of `ranges::cdata(E)` with rvalue `E` is deprecated. [*Note:* This deprecated usage exists so that `ranges::cdata(E)` has behavior consistent with `ranges::data(E)` when `E` is an rvalue. — *end note*]~~

3 [*Note:* Whenever `::std::ranges::cdata(E)` is a valid expression, it has pointer to object type. — *end note*]

29.8 Dangling wrapper [dangling.wrappers]

29.8.1 Class template `dangling` [dangling.wrap]

1 Class template `dangling` is a wrapper for an object that refers to another object whose lifetime may have ended. It is used by algorithms that accept rvalue ranges and return iterators.

```
namespace std~~2 { inline namespace v1 {~~ namespace ranges {
  template <CopyConstructible T>
  class dangling {
  public:
    constexpr dangling() requires DefaultConstructible<T>;
    constexpr dangling(T t);
    constexpr T get_unsafe() const;
  private:
    T value; // exposition only
  };

  template <Range R>
  using safe_iterator_t = // see below
    ~~conditional_t<is_lvalue_reference_v<R>,~~
      ~~iterator_t<R>,~~
      ~~dangling<iterator_t<R>>>;~~
}}
```

2 `safe_iterator_t<R>` is defined as follows:

(2.1) — If ~~`std2`~~`::std::ranges::begin(std::declval<R>())` is a well-formed expression, `safe_iterator_t<R>` is an alias for `iterator_t<R>`.

(2.2) — Otherwise, it is an alias for `dangling<iterator_t<R>>`.

29.8.2 `dangling` operations [dangling.wrap.ops]

29.8.2.1 `dangling` constructors [dangling.wrap.op.const]

`constexpr dangling() requires DefaultConstructible<T>;`

1 *Effects:* Constructs a `dangling`, value-initializing `value`.

`constexpr dangling(T t);`

2 *Effects:* Constructs a `dangling`, initializing `value` with `std::move(t)`.

29.8.2.2 `dangling::get_unsafe` [dangling.wrap.op.get]

`constexpr T get_unsafe() const;`

1 *Returns:* `value`.

Part II

Changes to P0789 R3 [P0789]

29 Ranges library [ranges]

[Editor's note: The following changes are suggested for P0789.]

29.3 Header <range> synopsis [range.synopsis]

[Editor's note: Change section "Header `<range>` synopsis" [range.synopsis], as follows:]

```
namespace std { namespace ranges {
  // ... as before

  enum class subrange_kind : bool { unsized, sized };
  // 29.8.3.1:
  template <Iterator I, Sentinel<I> S = I, subrange_kind K = see below >
      requires K == subrange_kind::sized || !SizedSentinel<S, I>
  class subrange;

  template <class I, class S, subrange_kind K>
    constexpr I begin(subrange<I, S, K>&& r);

  template <class I, class S, subrange_kind K>
    constexpr S end(subrange<I, S, K>&& r);

  // ... as before

  template <ForwardIterator I, Sentinel<I> S>
    requires Permutable<I>
    tagged_pair<tag::begin(I), tag::end(I)>
    subrange<I>
      rotate(I first, I middle, S last);

  template <ForwardRange Rng>
    requires Permutable<iterator_t<Rng>>
    tagged_pair<tag::begin(safe_iterator_t<Rng>),
                tag::end(safe_iterator_t<Rng>)>
    safe_subrange_t<Rng>
      rotate(Rng&& rng, iterator_t<Rng> middle);

  // ... as before

  template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
    tagged_pair<tag::begin(I), tag::end(I)>
    subrange<I>
      equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});

  template <ForwardRange Rng, class T, class Proj = identity,
      IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
    tagged_pair<tag::begin(safe_iterator_t<Rng>),
                tag::end(safe_iterator_t<Rng>)>
    safe_subrange_t<Rng>
      equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

```
    // ... as before
  }}
```

29.7 Range requirements [ranges.requirements]

29.7.2 Ranges [ranges.range]

[Editor's note: The equivalent change should be made in P0896R1 also.]

1 The `Range` concept defines the requirements of a type that allows iteration over its elements by providing a `begin` iterator and an `end` sentinel. [*Note:* Most algorithms requiring this concept simply forward to an `Iterator`-based algorithm by calling `begin` and `end`. — *end note*]

```
template <class T>
concept ~~Range~~range-impl = // exposition only
  requires(T&& t) {
    std::ranges::begin(std::forward<T>(t)); // not necessarily equality-preserving (see below)
    std::ranges::end(std::forward<T>(t));
  };

template <class T>
concept Range =
  range-impl<T&>;

template <class T>
concept forwarding-range = // exposition only
  Range<T> && range-impl<T>;
```

2 Given an ~~lvalue `t` of type `remove_reference_t<T>`, `Range<T>`~~expression `E` such that `decltype((E))` is `T`, *`range-impl`*`<T>` is satisfied only if

(2.1) — [`std::ranges::begin(`~~`t`~~`E)`,`std::ranges::end(`~~`t`~~`E)`) denotes a range.

(2.2) — Both `std::ranges::begin(`~~`t`~~`E)` and `std::ranges::end(`~~`t`~~`E)` are amortized constant time and non-modifying. [*Note:* `std::ranges::begin(`~~`t`~~`E)` and `std::ranges::end(`~~`t`~~`E)` do not require implicit expression variations (20.3.1.1). — *end note*]

(2.3) — If ~~`iterator_t<T>`~~the type of `std::ranges::begin(E)` satisfies `ForwardIterator`, `std::ranges::begin(`~~`t`~~`E)` is equality preserving.

3 Given an expression `E` such that `decltype((E))` is `T`, *`forwarding-range`*`<T>` is satisfied only if

(3.1) — The expressions `std::ranges::begin(E)` and `std::ranges::begin(static_cast<T&>(E))` are expression-equivalent.

(3.2) — The expressions `std::ranges::end(E)` and `std::ranges::end(static_cast<T&>(E))` are expression-equivalent.

4 [*Note:* Equality preservation of both `begin` and `end` enables passing a `Range` whose iterator type satisfies `ForwardIterator` to multiple algorithms and making multiple passes over the range by repeated calls to `begin` and `end`. Since `begin` is not required to be equality preserving when the return type does not satisfy `ForwardIterator`, repeated calls might not return equal values or might not be well-defined; `begin` should be called at most once for such a range. — *end note*]

29.7.11 Viewable ranges [ranges.viewable]

1 The `ViewableRange` concept specifies the requirements of a `Range` type that can be converted to a `View` safely.

```
template <class T>
concept ViewableRange =
  Range<T> && (~~is_lvalue_reference_v<T>~~forwarding-range<T> || View<decay_t<T>>); ~~// see below~~
```

2 There need not be any subsumption relationship between `ViewableRange<T>` and `is_lvalue_reference_v<T>`.

29.8 Range utilities [ranges.utilities]

29.8.3 Sub-ranges [ranges.subranges]

29.8.3.1 subrange [ranges.subrange]

```
namespace std { namespace ranges {
  // ... as before

  template <Iterator I, Sentinel<I> S = I, subrange_kind K = see below>
    requires K == subrange_kind::sized || !SizedSentinel<S, I>
  class subrange : public view_interface<subrange<I, S, K>> {
  private:
    static constexpr bool StoreSize =
      K == subrange_kind::sized && !SizedSentinel<S, I>; // exposition only
    I begin_ {}; // exposition only
    S end_ {}; // exposition only
    difference_type_t<I> size_ = 0; // exposition only; only present when StoreSize is true
  public:
    using iterator = I;
    using sentinel = S;

    subrange() = default;

    constexpr subrange(I i, S s) requires !StoreSize;

    constexpr subrange(I i, S s, difference_type_t<I> n)
      requires K == subrange_kind::sized;

~~    template <ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>
    constexpr subrange(subrange<X, Y, Z> r)
      requires !StoreSize || Z == subrange_kind::sized;~~

~~    template <ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>
    constexpr subrange(subrange<X, Y, Z> r, difference_type_t<I> n)
      requires K == subrange_kind::sized;~~

    template <not-same-as<subrange> R>
    requires forwarding-range<R> &&
      ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
    constexpr subrange(R&& r) requires !StoreSize || SizedRange<R>;

    template <forwarding-range R>
    requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
    constexpr subrange(R&& r, difference_type_t<I> n)
      requires K == subrange_kind::sized;

    template <not-same-as<subrange> PairLike>
      requires pair-like-convertible-to<PairLike, I, S>
    constexpr subrange(PairLike&& r) requires !StoreSize;
```

```
  template <pair-like-convertible-to<I, S> PairLike>
  constexpr subrange(PairLike&& r, difference_type_t<I> n)
    requires K == subrange_kind::sized;

  template <not-name-as<subrange> R>
    requires Range<R> && ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
  constexpr subrange(R& r) requires !StoreSize || SizedRange<R>;

  template <not-same-as<subrange> PairLike>
    requires pair-like-convertible-from<PairLike, const I&, const S&>
  constexpr operator PairLike() const;

  constexpr I begin() const;
  constexpr S end() const;
  constexpr bool empty() const;
  constexpr difference_type_t<I> size() const
    requires K == subrange_kind::sized;
  [[nodiscard]] constexpr subrange next(difference_type_t<I> n = 1) const;
  [[nodiscard]] constexpr subrange prev(difference_type_t<I> n = 1) const
    requires BidirectionalIterator<I>;
  constexpr subrange& advance(difference_type_t<I> n);
};

template <class I, class S, subrange_kind K>
  constexpr I begin(subrange<I, S, K>&& r);

template <class I, class S, subrange_kind K>
  constexpr S end(subrange<I, S, K>&& r);

template <Iterator I, Sentinel<I> S>
subrange(I, S, difference_type_t<I>) -> subrange<I, S, subrange_kind::sized>;

template <iterator-sentinel-pair P>
subrange(P) ->
  subrange<tuple_element_t<0, P>, tuple_element_t<1, P>>;

template <iterator-sentinel-pair P>
subrange(P, difference_type_t<tuple_element_t<0, P>>) ->
  subrange<tuple_element_t<0, P>, tuple_element_t<1, P>, subrange_kind::sized>;

template <Iterator I, Sentinel<I> S, subrange_kind K>
subrange(subrange<I, S, K>, difference_type_t<I>) ->
  subrange<I, S, subrange_kind::sized>;

template <Range R>
subrange(R&) -> subrange<iterator_t<R>, sentinel_t<R>>;

template <SizedRange R>
subrange(R&) -> subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

template <forwarding-range R>
subrange(R&&) -> subrange<iterator_t<R>, sentinel_t<R>>;

template <forwarding-range R>
```

```
    requires SizedRange<R>
  subrange(R&&) -> subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

  template <forwarding-range R>
  subrange(R&&, difference_type_t<iterator_t<R>>) ->
    subrange<iterator_t<R>, sentinel_t<R>, subrange_kind::sized>;

  // ... as before

  template <Range R>
    using safe_subrange_t =
      conditional_t<forwarding-range<R>,
        subrange<iterator_t<R>>,
        dangling<subrange<iterator_t<R>>>>;
}}
```

1 The default value for `subrange`'s third (non-type) template parameter is:

(1.1) — If `SizedSentinel<S, I>` is satisfied, `subrange_kind::sized`.

(1.2) — Otherwise, `subrange_kind::unsized`.

29.8.3.1.1 subrange constructors [ranges.subrange.ctor]

```
constexpr subrange(I i, S s) requires !StoreSize;
```

1 *Effects:* Initializes `begin_` with `i` and `end_` with `s`.

```
constexpr subrange(I i, S s, difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

2 *Requires:* `n == distance(i, s)`.

3 *Effects:* Initializes `begin_` with `i`, `end_` with `s`. If `StoreSize` is `true`, initializes `size_` with `n`.

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>
constexpr subrange(subrange<X, Y, Z> r)
  requires !StoreSize || Z == subrange_kind::sized;
```

4 *Effects:* Equivalent to:

(4.1) — If `StoreSize` is `true`, `subrange{r.begin(), r.end(), r.size()}`.

(4.2) — Otherwise, `subrange{r.begin(), r.end()}`.

```
template <ConvertibleTo<I> X, ConvertibleTo<S> Y, subrange_kind Z>
constexpr subrange(subrange<X, Y, Z> r, difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

5 *Effects:* Equivalent to `subrange{r.begin(), r.end(), n}`.

```
template <not-same-as<subrange> R>
  requires forwarding-range<R> &&
    ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R&& r) requires !StoreSize || SizedRange<R>;
```

6 *Effects:* Equivalent to:

(6.1) — If `StoreSize` is `true`, `subrange{ranges::begin(r), ranges::end(r), ranges::size(r)}`.

(6.2) — Otherwise, `subrange{ranges::begin(r), ranges::end(r)}`.

```
template <forwarding-range R>
  requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R&& r, difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

7 *Effects:* Equivalent to `subrange{ranges::begin(r), ranges::end(r), n}`.

```
template <not-same-as<subrange> PairLike>
  requires pair-like-convertible-to<PairLike, I, S>
constexpr subrange(PairLike&& r) requires !StoreSize;
```

8 *Effects:* Equivalent to:

```
subrange{get<0>(std::forward<PairLike>(r)), get<1>(std::forward<PairLike>(r))}
```

```
template <pair-like-convertible-to<I, S> PairLike>
constexpr subrange(PairLike&& r, difference_type_t<I> n)
  requires K == subrange_kind::sized;
```

9 *Effects:* Equivalent to:

```
subrange{get<0>(std::forward<PairLike>(r)), get<1>(std::forward<PairLike>(r)), n}
```

```
template <not-name-as<subrange> R>
  requires Range<R> && ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R& r) requires !StoreSize || SizedRange<R>;
```

10 *Effects:* Equivalent to:

(10.1) — If `StoreSize` is `true`, `subrange{ranges::begin(r), ranges::end(r), distance(r)}`.

(10.2) — Otherwise, `subrange{ranges::begin(r), ranges::end(r)}`.

29.8.3.1.2 subrange operators [ranges.subrange.ops]

```
template <not-same-as<subrange> PairLike>
  requires pair-like-convertible-from<PairLike, const I&, const S&>
constexpr operator PairLike() const;
```

1 *Effects:* Equivalent to: `return PairLike(begin_, end_);`.

29.8.3.1.3 subrange accessors [ranges.subrange.accessors]

```
constexpr I begin() const;
```

1 *Effects:* Equivalent to: `return begin_;`.

```
constexpr S end() const;
```

2 *Effects:* Equivalent to: `return end_;`.

```
constexpr bool empty() const;
```

3 *Effects:* Equivalent to: `return begin_ == end_;`.

```
constexpr difference_type_t<I> size() const
  requires K == subrange_kind::sized;
```

4 *Effects:* Equivalent to:

(4.1) — It `StoreSize` is `true`, `return size_;`.

(4.2) — Otherwise, `return end_ - begin_;`.

```
[[nodiscard]] constexpr subrange next(difference_type_t<I> n = 1) const;
```

5 *Effects:* Equivalent to:

```
auto tmp = *this;
tmp.advance(n);
return tmp;
```

6 [*Note:* If `ForwardIterator<I>` is not satisfied, `next` may invalidate `*this`. — *end note*]

```
[[nodiscard]] constexpr subrange prev(difference_type_t<I> n = 1) const
  requires BidirectionalIterator<I>;
```

7 *Effects:* Equivalent to:

```
auto tmp = *this;
tmp.advance(-n);
return tmp;
```

```
constexpr subrange& advance(difference_type_t<I> n);
```

8 *Effects:* Equivalent to:

(8.1) — If `StoreSize` is `true`,

```
size_ -= n - ranges::advance(begin_, n, end_);
return *this;
```

(8.2) — Otherwise,

```
ranges::advance(begin_, n, end_);
return *this;
```

29.8.3.1.4 subrange non-member functions [ranges.subrange.nonmember]

```
template <class I, class S, subrange_kind K>
  constexpr I begin(subrange<I, S, K>&& r);
```

1 *Effects:* Equivalent to:

```
return r.begin();
```

```
template <class I, class S, subrange_kind K>
  constexpr S end(subrange<I, S, K>&& r);
```

2 *Effects:* Equivalent to:

```
return r.end();
```

```
template <std::size_t N, class I, class S, subrange_kind K>
  requires N < 2
constexpr auto get(const subrange<I, S, K>& r);
```

3 *Effects:* Equivalent to:

```
if constexpr (N == 0)
  return r.begin();
else
  return r.end();
```

29.9 Range adaptors [ranges.adaptors]

29.9.4 `view::all` [ranges.adaptors.all]

1 The purpose of `view::all` is to return a `View` that includes all elements of the `Range` passed in.

2 The name `view::all` denotes a range adaptor object (??). The expression `view::all(E)` for some subexpression `E` is expression-equivalent to:

(2.1) — *`DECAY_COPY`*`(E)` if the decayed type of `E` satisfies the concept `View`.

(2.2) — `subrange{E}` if ~~`E` is an lvalue and has a type that satisfies concept `Range`~~that expression is well-formed.

(2.3) — Otherwise, `view::all(E)` is ill-formed.

Remark: Whenever `view::all(E)` is a valid expression, it is a prvalue whose type satisfies `View`.

29.10 Algorithms library [range.algorithms]

[Editor's note: Some of the algorithms in the Ranges TS (`rotate` and `equal_range`) actually return subranges, but they do so using `tagged_pair`. With the addition of a proper `subrange` type, we suggest changing these algorithms to return `subrange`.]

29.10.3 Mutating sequence operations [range.alg.modifying.operations]

29.10.3.11 Rotate [range.alg.rotate]

```
template <ForwardIterator I, Sentinel<I> S>
  requires Permutable<I>
  tagged_pair<tag::begin(I), tag::end(I)>
  subrange<I>
    rotate(I first, I middle, S last);

template <ForwardRange Rng>
  requires Permutable<iterator_t<Rng>>
  tagged_pair<tag::begin(safe_iterator_t<Rng>),
              tag::end(safe_iterator_t<Rng>)>
  safe_subrange_t<Rng>
    rotate(Rng&& rng, iterator_t<Rng> middle);
```

1 *Effects:* For each non-negative integer `i < (last - first)`, places the element from the position `first + i` into position `first + (i + (last - middle)) % (last - first)`.

2 *Returns:* `{first + (last - middle), last}`.

3 *Remarks:* This is a left rotate.

4 *Requires:* `[first,middle)` and `[middle,last)` shall be valid ranges.

5 *Complexity:* At most `last - first` swaps.

29.10.4 Sorting and related operations [range.alg.sorting]

29.10.4.3 Binary search [range.alg.binary.search]

29.10.4.3.3 `equal_range` [range.equal.range]

```
template <ForwardIterator I, Sentinel<I> S, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<I, Proj>> Comp = less<>>
  tagged_pair<tag::begin(I), tag::end(I)>
  subrange<I>
    equal_range(I first, S last, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

```
template <ForwardRange Rng, class T, class Proj = identity,
    IndirectStrictWeakOrder<const T*, projected<iterator_t<Rng>, Proj>> Comp = less<>>
  ~~tagged_pair<tag::begin(safe_iterator_t<Rng>),~~
              ~~tag::end(safe_iterator_t<Rng>)>~~
  safe_subrange_t<Rng>
    equal_range(Rng&& rng, const T& value, Comp comp = Comp{}, Proj proj = Proj{});
```

1 *Requires:* The elements `e` of `[first,last)` shall be partitioned with respect to the expressions `invoke(comp, invoke(proj, e), value)` and `!invoke(comp, value, invoke(proj, e))`. Also, for all elements `e` of `[first, last)`, `invoke(comp, invoke(proj, e), value)` shall imply `!invoke(comp, value, invoke(proj, e))`.

2 *Returns:*

```
{lower_bound(first, last, value, comp, proj),
 upper_bound(first, last, value, comp, proj)}
```

3 *Complexity:* At most $2 * \log_2($`last - first`$) + \mathcal{O}(1)$ applications of the comparison function and projection.

Annex A (informative) Acknowledgements [acknowledgements]

This work was made possible by the support of my employer, Facebook.

Bibliography

[1] Eric Niebler. P0789r3: Range adaptors and utilities, 05 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0789r3.pdf.

[2] Eric Niebler and Carter Casey. P0896r1: Merging the ranges ts, 05 2018. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r1.pdf.

Index

Index of library names