Better, Safer Range Access Customization Points

Note: this is an early draft. It’s known to be incomplet and incorrect, and it has lots of bad formatting.
# Contents

## Contents

<table>
<thead>
<tr>
<th>1 General</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Scope</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Problems with std::experimental::ranges::begin</td>
<td>1</td>
</tr>
<tr>
<td>1.3 Suggested Design</td>
<td>1</td>
</tr>
<tr>
<td>1.4 References</td>
<td>3</td>
</tr>
</tbody>
</table>

## I Changes to P0896

| 20 Standard Library, Version 1 | 5 |
| 21 Standard Library, Version 2 | 6 |
| 21.4 Ranges library | 6 |

## II Changes to P0789 R2

| 10 Ranges library | 13 |
| 10.6 Range requirements | 13 |
| 10.7 Range utilities | 14 |
| 10.8 Range adaptors | 19 |

| 11 Algorithms library | 20 |
| 11.1 General | 20 |
| 11.4 Mutating sequence operations | 20 |
| 11.5 Sorting and related operations | 21 |

## A Acknowledgements | 22 |

## Bibliography | 23 |

## Index | 24 |

## Index of library names | 25 |
1 General

“Begin at the beginning, the King said, very gravely, and go on till you come to the end: then stop.”

—Lewis Carroll

1.1 Scope

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, “Merging the Ranges TS” ([2]), and to P0789, “Range Adaptors and Utilities” ([1]).

1.2 Problems with std::experimental::ranges::begin

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.

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 — P0789’s subrange<> — the iterator’s validity does not depend on the range’s lifetime at all. In the case where a prvalue subrange<> is passed to an algorithm, returning a wrapped iterator is totally unnecessary.

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 succinct and clear.

1.3 Suggested Design

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 subrange, an important usage scenario.

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

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:

```cpp
class 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:

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

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.

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

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

2. `std2::begin(E)` will consider an overload of `begin(E)` found by ADL, looked up in a context that
   (a) does not include `std2::begin`, and (b) includes the following declaration:
```cpp
// "Poison pill" overload:
template <class T>
void begin(T&&) = delete;
```

This approach gives `std2::begin` the property that, for some rvalue expression `E` of type `T`, the expression `std2::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.

5 This design has the following benefits:

(6.1) — No iterator returned from `std2::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::begin, end, and friend..` 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::begin(std::declval<R>())` is ill-formed. In code:

```cpp
template <Range R, class = void>
struct __safe_iterator {
    using type = dangling<iterator_t<R>>;
};

template <class 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.
1.4 References

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++*


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
20  Standard Library, Version 1  [std]

[Editor's note: The following changes are suggested for P0896.]

20.6.4.1 Header <string_view> synopsis  [string.view.synop]
[Editor's note: change the <string_view> header synopsis as follows:]

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

    // 20.6.4.3, 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 sub-sections.]

20.6.4.3 basic_string_view range access  [string.view.range_access]

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

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

Returns: x.begin().

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

Returns: x.end().
21 Standard Library, Version 2 [std2]

21.4 Ranges library [std2.ranges]

21.4.4 Range access [std2.range.access]

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

1.4.4.1 begin [std2.range.access.begin]

The name `begin` denotes a customization point object (20.1.4.2.1.6). The expression `::std2::begin(E)` for
some subexpression E is expression-equivalent to:

1.1 — `::std2::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 `::std2::begin(E)` behaves similarly to `::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 re-
quirements of Iterator<I> with overload resolution performed in a context that includes the following
declarations:

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

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

1.5 — Otherwise, `::std2::begin(E)` is ill-formed.

[Note: Whenever `::std2::begin(E)` is a valid expression, its type satisfies Iterator. — end note]

1.4.4.2 end [std2.range.access.end]

The name `end` denotes a customization point object (20.1.4.2.1.6). The expression `::std2::end(E)` for
some subexpression E is expression-equivalent to:

1.1 — `::std2::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 `::std2::end(E)` behaves similarly to `::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) 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(::std2::begin(E))>. If Sentinel is not satisfied,
the program is ill-formed with no diagnostic required.
Otherwise, DECAY_COPY(end(E)) if it is a valid expression and its type $S$ meets the syntactic requirements of $\text{Sentinel}<S, \text{decltype} (::\text{std2}::\text{begin}(E))>$ with overload resolution performed in a context that includes the following declarations:

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

and does not include a declaration of ::std2::end. If $\text{Sentinel}$ is not satisfied, the program is ill-formed with no diagnostic required.

Otherwise, ::std2::end(E) is ill-formed.

2 [Note: Whenever ::std2::end(E) is a valid expression, the types of ::std2::end(E) and ::std2::begin(E) satisfy $\text{Sentinel}$. — end note]

21.4.4.3 cbegin

The name cbegin denotes a customization point object (20.1.4.2.1.6). The expression ::std2::cbegin(E) for some subexpression $E$ of type $T$ is expression-equivalent to:

(1.1) — ::std2::begin(static_cast<const T&>(E)) if $E$ is an lvalue.

(1.2) — Otherwise, ::std2::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 ::std2::cbegin(E) is a valid expression, its type satisfies $\text{Iterator}$. — end note]

21.4.4.4 cend

The name cend denotes a customization point object (20.1.4.2.1.6). The expression ::std2::cend(E) for some subexpression $E$ of type $T$ is expression-equivalent to:

(1.1) — ::std2::end(static_cast<const T&>(E)) if $E$ is an lvalue.

(1.2) — Otherwise, ::std2::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 ::std2::cend(E) is a valid expression, the types of ::std2::cend(E) and ::std2::cbegin(E) satisfy $\text{Sentinel}$. — end note]

21.4.4.5 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).]

The name rbegin denotes a customization point object (20.1.4.2.1.6). The expression ::std2::rbegin(E) for some subexpression $E$ is expression-equivalent to:
21.4.4.6 rend

The name rend denotes a customization point object (20.1.4.2.1.6). The expression ::std2::rend(E) for some subexpression E is expression-equivalent to:

1 — ::std2::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]

2 — Otherwise, DECAY_COPY((E).rend()) if it is a valid expression and its type S meets the syntactic requirements of Sentinel<S, decltype(::std2::rbegin(E))>. If Sentinel is not satisfied, the program is ill-formed.

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::rbegin(E))> with overload resolution performed in a context that includes the following declaration:

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

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

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

5 — Otherwise, ::std2::rend(E) is ill-formed.

2 [Note: Whenever ::std2::rend(E) is a valid expression, its type satisfies Iterator. — end note]
21.4.4.7 crbegin

The name crbegin denotes a customization point object (20.1.4.2.1.6). The expression ::std2::crbegin(E) for some subexpression E of type T is expression-equivalent to:

1. ::std2::rbegin(static_cast<const T&>(E)) if E is an lvalue.
2. Otherwise, ::std2::rbegin(static_cast<const T&&>(E)).

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

[Note: Whenever ::std2::crbegin(E) is a valid expression, its type satisfies Iterator. —end note]

21.4.4.8 crend

The name crend denotes a customization point object (20.1.4.2.1.6). The expression ::std2::crend(E) for some subexpression E of type T is expression-equivalent to:

1. ::std2::rend(static_cast<const T&>(E)) if E is an lvalue.
2. Otherwise, ::std2::rend(static_cast<const T&&>(E)).

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

[Note: Whenever ::std2::crend(E) is a valid expression, the types of ::std2::crend(E) and ::std2::crbegin(E) satisfy Sentinel. —end note]

21.4.5 Range primitives

In addition to being available via inclusion of the <std2/range> header, the customization point objects in 21.4.5 are available when <std2/iterator> is included.

21.4.5.1 size

The name size denotes a customization point object (20.1.4.2.1.6). The expression ::std2::size(E) for some subexpression E of type T is expression-equivalent to:

1. DECAY_COPY(extent_v<T>) if T is an array type (6.7.2).
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.
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&k) = delete;

   and does not include a declaration of ::std2::size, and disable_sized_range<remove_cvref_t<T>> is false.

4. Otherwise, DECAY_COPY(::std2::end(E) - ::std2::begin(E)), except that E is only evaluated once, if it is a valid expression and the types I and S of ::std2::begin(E) and ::std2::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.

5. Otherwise, ::std2::size(E) is ill-formed.

[Note: Whenever ::std2::size(E) is a valid expression, its type satisfies Integral. —end note]
empty

The name `empty` denotes a customization point object (20.1.4.2.1.6). The expression `::std2::empty(E)` for some subexpression `E` is expression-equivalent to:

1. `bool((E).empty())` if it is a valid expression.
2. Otherwise, `::std2::size(E) == 0` if it is a valid expression.
3. Otherwise, `bool(::std2::begin(E) == ::std2::end(E))`, except that `E` is only evaluated once, if it is a valid expression and the type of `::std2::begin(E)` satisfies `ForwardIterator`.
4. Otherwise, `::std2::empty(E)` is ill-formed.

[Note: Whenever `::std2::empty(E)` is a valid expression, it has type `bool`. — end note]

data

The name `data` denotes a customization point object (20.1.4.2.1.6). The expression `::std2::data(E)` for some subexpression `E` is expression-equivalent to:

1. `::std2::data(static_cast<const T&>(E))` if `E` is an lvalue.
2. Otherwise, `::std2::begin(E)` if it is a valid expression of pointer to object type.
3. Otherwise, `::std2::data(E)` is ill-formed.

[Note: Whenever `::std2::data(E)` is a valid expression, it has pointer to object type. — end note]

cdata

The name `cdata` denotes a customization point object (20.1.4.2.1.6). The expression `::std2::cdata(E)` for some subexpression `E` of type `T` is expression-equivalent to:

1. `::std2::data(static_cast<const T&>(E))` if `E` is an lvalue.

[Note: This deprecated usage exists so that `::std2::cdata(E)` has behavior consistent with `::std2::data(E)` when `E` is an rvalue. — end note]

2. Use of `::std2::cdata(E)` with rvalue `E` is deprecated. [Note: This deprecated usage exists so that `::std2::cdata(E)` has behavior consistent with `::std2::data(E)` when `E` is an rvalue. — end note]

3. [Note: Whenever `::std2::cdata(E)` is a valid expression, it has pointer to object type. — end note]

Range requirements

[Editor's note: As in P0896.]

Dangling wrapper

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 std2 { inline namespace v1 {

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::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>>.

21.4.7.1.1 dangling operations [std2.dangling.wrap.ops]

21.4.7.1.1.1 dangling constructors [std2.dangling.wrap.op.const]

constexpr dangling() requires DefaultConstructible<T>;

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

cnvexpr dangling(T t);

2 Effects: Constructs a dangling, initializing value with t.

21.4.7.1.1.2 dangling::get_unsafe [std2.dangling.wrap.op.get]

constexpr T get_unsafe() const;

1 Returns: value.
Part II

Changes to P0789 R2

[P0789]
10 Ranges library

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

[Editor’s note: Change section “Header <experimental/ranges/range> synopsis” [range.synopsis], as follows:]

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
  // ... as before

  enum class subrange_kind : bool { unsized, sized };
  // 10.7.2.1:
  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
}}}

10.6 Range requirements

10.6.2 Ranges

[Editor’s note: The equivalent change should be made in P0896 also.]

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 bool Range =
        range-impl<T>;

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

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

\[ \text{ranges::begin}(t), \text{ranges::end}(t) \] denotes a range.
Both `ranges::begin(tE)` and `ranges::end(tE)` are amortized constant time and non-modifying.

\[ \text{Note: } \text{`ranges::begin(tE)` and `ranges::end(tE)` do not require implicit expression variations (20.3.1.1).} \text{- end note} \]

If `iterator_t<T>` the type of `ranges::begin(E)` satisfies `ForwardIterator`, `ranges::begin(tE)` is equality preserving.

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

\[ \text{The expressions `ranges::begin(E)` and `ranges::begin(static_cast<T&>(E))` are expression-equivalent.} \]

\[ \text{The expressions `ranges::end(E)` and `ranges::end(static_cast<T&>(E))` are expression-equivalent.} \]

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

### 10.6.11 Viewable ranges

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

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

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

### 10.7 Range utilities

#### 10.7.2 Sub-ranges

```
namespace std { namespace experimental { namespace ranges { inline namespace v1 {
    // ... as before

    template <class T, class U>
    concept bool not_same_as = // exposition only
        !Same<remove_cvref_t<T>, remove_cvref_t<U>>;

    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 <pair_like_convertible_to<I, S> not_same_as<subrange> PairLike>
  requires pair_like_convertible_to<PairLike, I, S>
constexpr subrange(PairLike&& r) requires !StoreSize;

constexpr subrange(PairLike&& r, difference_type_t<I> n)
  requires K == subrange_kind::sized;

template <Range R>
  requires ConvertibleTo<iterator_t<R>, I> && ConvertibleTo<sentinel_t<R>, S>
constexpr subrange(R&& r) requires !StoreSize || SizedRange<R>;

template <pair_like_convertible_from<const I&, const S&> 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>
```cpp
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
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>>>>;
}}}
```

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

1. If `SizedSentinel<S, I>` is satisfied, `subrange_kind::sized`

2. Otherwise, `subrange_kind::unsized`.

10.7.2.1.1 `subrange` constructors

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

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

```cpp
constexpr subrange(I i, S s, difference_type_t<I> n) requires K == subrange_kind::sized;
```
2 \textit{Requires:} n == \textbf{distance}(i, s).

3 \textit{Effects:} Initializes \texttt{begin\_} with \texttt{i}, \texttt{end\_} with \texttt{s}. If \texttt{StoreSize} is \texttt{true}, initializes \texttt{size\_} with \texttt{n}.

4 template <ConvertibleTo\texttt{I} X, ConvertibleTo\texttt{S} Y, subrange\_kind Z> constexpr subrange(subrange\texttt{<X, Y, Z>} r) requires !\texttt{StoreSize} || Z == subrange\_kind::\texttt{sized};

4 \textit{Effects:} Equivalent to:

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

4.2 — Otherwise, subrange\tt{(r.begin\texttt{()}, r.end\texttt{()})}.

5 template <ConvertibleTo\texttt{I} X, ConvertibleTo\texttt{S} Y, subrange\_kind Z> constexpr subrange(subrange\texttt{<X, Y, Z>} r, difference\_type\_t<\texttt{I}> n) requires K == subrange\_kind::\texttt{sized};

5 \textit{Effects:} Equivalent to \texttt{subrange(r.begin\texttt{()}, r.end\texttt{()}, n)}.

6 template <\texttt{not\texttt{-same\texttt{-as}}<subrange>} R> requires \texttt{forwarding\texttt{-range}<R>} && ConvertibleTo\texttt{<iterator\_t<R>, I>} && ConvertibleTo\texttt{<sentinel\_t<R>, S>} constexpr subrange\texttt{(R&& r)} requires !\texttt{StoreSize} || \texttt{SizedRange<R>};

6 \textit{Effects:} Equivalent to:

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

6.2 — Otherwise, subrange\tt{(ranges::begin\texttt{(r)}, ranges::end\texttt{(r)})}.

7 template <\texttt{forwarding\texttt{-range}} R> requires ConvertibleTo\texttt{<iterator\_t<R>, I>} && ConvertibleTo\texttt{<sentinel\_t<R>, S>} constexpr subrange\texttt{(R&& r, difference\_type\_t<\texttt{I}> n)} requires K == subrange\_kind::\texttt{sized};

7 \textit{Effects:} Equivalent to \texttt{subrange(ranges::begin\texttt{(r)}, ranges::end\texttt{(r)}, n)}.

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

8 \textit{Effects:} Equivalent to:

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

9 template <\texttt{pair\texttt{-like\texttt{-convertible\texttt{-to}}<\texttt{I}, S>}} PairLike>
constexpr subrange(PairLike&& r, difference\_type\_t<\texttt{I}> n) requires K == subrange\_kind::\texttt{sized};

9 \textit{Effects:} Equivalent to:

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

10 template <\texttt{Range} R> requires ConvertibleTo\texttt{<iterator\_t<R>, I>} && ConvertibleTo\texttt{<sentinel\_t<R>, S>} constexpr subrange\texttt{(R&& r)} requires !\texttt{StoreSize} || \texttt{SizedRange<R>};

10 \textit{Effects:} Equivalent to:

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

10.2 — Otherwise, subrange\tt{(ranges::begin\texttt{(r)}, ranges::end\texttt{(r)})}.
10.7.2.1.2 subrange operators

```cpp
template <pair_like_convertible_from<const I&, const S&> not_same_as subrange> PairLike
    requires pair_like_convertible_from<PairLike, const I&, const S&>
constexpr operator PairLike() const;
```

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

10.7.2.1.3 subrange accessors

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

Effects: Equivalent to: `return begin_;`

```cpp
castexpr S end() const;
```

Effects: Equivalent to: `return end_;`

```cpp
castexpr bool empty() const;
```

Effects: Equivalent to: `return begin_ == end_;`

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

Effects: Equivalent to:

1. If `StoreSize` is true, return `size_`;
2. Otherwise, return `end_ - begin_`;

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

Effects: Equivalent to:

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

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

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

Effects: Equivalent to:

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

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

Effects: Equivalent to:

1. If `StoreSize` is true,
   ```cpp
   size_ -= n - ranges::advance(begin_, n, end_);
   return *this;
   ```
2. Otherwise,
   ```cpp
   ranges::advance(begin_, n, end_);
   return *this;
   ```
### 10.7.2.1.4 subrange non-member functions [ranges.subrange.nonmember]

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

1. **Effects:** Equivalent to:
   ```cpp```
   return r.begin();
   ```cpp```

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

2. **Effects:** Equivalent to:
   ```cpp```
   return r.end();
   ```cpp```

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

### 10.8 Range adaptors [ranges.adaptors]

#### 10.8.4 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`. 
11 Algorithms library

[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.]

11.1 General

[Editor's note: Change the <experimental/ranges/algorithm> synopsis as follows:]

```cpp
#include <initializer_list>

namespace std { namespace experimental { namespace ranges { inline namespace v1 {
    // ... 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
}}}}
```

11.4 Mutating sequence operations

11.4.11 Rotate

```cpp
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 \( \text{first} + i \) into position \( \text{first} + (i + (\text{last} - \text{middle})) \% (\text{last} - \text{first}) \).
2   Returns: \( \{\text{first} + (\text{last} - \text{middle}), \text{last}\} \).
3   Remarks: This is a left rotate.
4   Requires: \([\text{first},\text{middle})\) and \([\text{middle},\text{last})\) shall be valid ranges.
5   Complexity: At most last \(-\) first swaps.

11.5  Sorting and related operations [alg.sorting]
11.5.3  Binary search [alg.binary.search]
11.5.3.3  equal_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 \([\text{first},\text{last})\) shall be partitioned with respect to the expressions \( \text{invoke}(\text{comp}, \text{invoke}(\text{proj}, e), \text{value}) \) and \( !\text{invoke}(\text{comp}, \text{value}, \text{invoke}(\text{proj}, e)) \). Also, for all elements \( e \) of \([\text{first}, \text{last})\), \( \text{invoke}(\text{comp}, \text{value}, \text{invoke}(\text{proj}, e), \text{value}) \) shall imply \( !\text{invoke}(\text{comp}, \text{value}, \text{invoke}(\text{proj}, e)) \).
2   Returns:
    \( \{\text{lower_bound}(\text{first}, \text{last}, \text{value}, \text{comp}, \text{proj}), \text{upper_bound}(\text{first}, \text{last}, \text{value}, \text{comp}, \text{proj})\} \)
3   Complexity: At most \( 2 \star \log_2(\text{last} - \text{first}) + O(1) \) 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


Index

C++ Standard, 3
Ranges TS, 3
<string_view>, 5
Index of library names

advance
  subrange, 18

basic_string_view
  begin, 5
  end, 5
begin
  subrange, 18, 19

dangling, 10
  dangling, 11
  get_unsafe, 11

empty
  subrange, 18
end
  subrange, 18, 19
equal_range, 21

get
  subrange, 19
get_unsafe
  dangling, 11

next
  subrange, 18

operator PairLike
  subrange, 18

prev
  subrange, 18

rotate, 20

size
  subrange, 18
subrange
  subrange, 16, 17