`views::enumerate`

Document #:	P2164R7
Date:	2022-10-15
Programming Language C++	
Audience:	LEWG, SG-9
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

A struct with 2 members, how hard can it be?!

Abstract

We propose a view `enumerate` whose value type is a struct with 2 members `index` and `value` representing respectively the position and value of the elements in the adapted range.

Revisions

R7

- This version asks LEWG to choose between `tuple` or `enumerate_result` as the reference and value types of `enumerate_view`. The approach presented in previous revisions of having a value type and a reference type of different types proved not workable. We need to pick one of the two options. Wording is provided for both.
- Add missing `iter_move` and `iter_swap` functions.
- Add the markup for freestanding
- Add feature test macro
- Formatting fixes.

R6

Wording changes:

- Add `enumerate_result` to the list of `tuple-like` types
- Because `enumerate_view::iterator::operator*` returns values, `enumerate_view::iterator` cannot be a `Cpp17ForwardIterator`. Change `iterator_category` and add `iterator_concept` accordingly.

R5

Instead of adding complexity to `enumerate_result`, we assume changes made by P2165R2 [1]. P2165R2 [1] makes `pair` constructible from *pair-like* objects, and associative containers deduction guides work with ranges of *pair-like* objects. With these changes, `enumerate_result` can remain a simple aggregate. We just need to implement the tuple protocol for it (`get`, `tuple_element`, `tuple_size`).

P2165R2 [1] ensures a common reference exists between `enumerate_result` and `std::tuple` as long as one exists between each element.

`count_type` is renamed to `index_type`. I am not sure why I ever chosed `count_type` as the initial name.

R4

This revision is intended to illustrate the effort necessary to support named fields for `index` and `value`. In previous revisions, the value and reference types were identical, a regrettable blunder that made the wording and implementation efforts smaller than they are. `reference` and `value_type` types however needs to be different, if only to make the `ranges::to` presented in this very paper.

If that direction is acceptable, better wording will be provided to account for these new `reference` and `value_type` types.

This revision also gets rid of the `const index` value as LEWG strongly agreed that it was a terrible idea to begin with, one that would make composition with other views cumbersome.

R3

- Typos and minor wording improvements

R2, following mailing list reviews

- Make `value_type` different from `reference` to match other views
- Remove inconsistencies between the wording and the description
- Add relevant includes and namespaces to the examples

R1

- Fix the index type

Tony tables

Before	After

<pre>std::vector days{"Mon", "Tue",
  "Wed", "Thu", "Fri", "Sat", "Sun"};
 	
int idx = 0;
for(const auto & d : days) {
    print("{} {} \n", idx, d);
    idx++;
}</pre>
	<pre>#include <ranges>
	
	std::vector days{"Mon", "Tue",
	  "Wed", "Thu", "Fri", "Sat", "Sun"};
	
	for(const auto & e : std::views::enumerate(days)) {
	    print("{} {} \n", e.index, e.value);
	}</pre>

Motivation

The impossibility to extract an index from a range-based for loop leads to the use of non-range-based `for` loops, or the introduction of a variable in the outer scope. This is both more verbose and error-prone: in the example above, the type of `idx` is incorrect.

`enumerate` is a library solution solving this problem, enabling the use of range-based for loops in more cases.

It also composes nicely with other range facilities: The following creates a map from a vector using the position of each element as key.

```
my_vector | views::enumerate | ranges::to<map>;
```

This feature exists in some form in Python, Rust, Go (backed into the language), and in many C++ libraries: ranges-v3, folly, boost::ranges (`indexed`).

The existence of this feature or lack thereof is the subject of recurring StackOverflow questions.

Design

`std::tuple` vs aggregate with name members

Following the trend of using meaningful names instead of returning pairs or tuples, one option is to use a struct with named public members.

```
struct enumerate_result {
    count index;
    T value;
};
```

This design was previously discussed by LEWGI in Belfast in the context of P1894R0 [3], and many people have expressed a desire for such struct with names.

```
std::vector<double> v;
enumerate(view) | to<std::vector>(); // std::vector<std::tuple<std::size_t, double>>.
enumerate(view) | to<std::map>();    // std::map<std::size_t, double>.
```

Why not just always return a `tuple` and rely on structure binding?

If a range reference type is convertible to the index type, it is error-prone whether one should write

```
for(auto && [value, index] : view | std::views::enumerate)
for(auto && [index, value] : view | std::views::enumerate)
```

Having named members avoids this issue. The feedback I keep getting is ”we should use a struct if we can”. Which is consistent with previous LEWG guidelines to avoid using pair when a more meaningful type is possible.

Why use a tuple?

The drawback of using a distinct type is that

```
auto vec = enumerate(view) | ranges::to<std::vector>();
```

would produce a `vector<enumerate_result<std::size_t, range_value_t<decltype(view)>>`

where ideally, I think it should produce a tuple.

Why not `enumerate_result` as reference type and `tuple` as value type?

This was the approached proposed by the previous revision of the paper and my preferred solution. Best of both world. It only has a smal drawback: it doesn't work.

Many algorithms end up requiring `invocable<F&, iter_value_t<I>&> && invocable<F&, iter_reference_t<I>>` (where F is a function), which would require `std::tuple<std::size_t, Foo>&` to be convertible to `enumerate_result<std::size_t, Foo>`.

In practice, this means that many algorithms are not utilisable if reference and values are not the same type

```
std::ranges:::find(enumerate(/*...*/), [](auto const& p) { // constraints not satisfied.
    return /*...*/;
})
```

This is simply not acceptable.

Tomasz also observed that it would interract pourly with `as_const`.

```
for (auto const& p : enumerate(/*...*/)) {
    something(p.value); // OK
}

for (auto const& p : enumerate(/*...*/) | views|as_const) {
    something(p.value); // KO decltype(p) is tuple<std::size_t, const Foo&>
}
```

Which is... not great. The unfortunate `invocable<F&, iter_value_t<I>&>` constraints exists as some algorithms (not find) may constructs value types out of the elements of the range.

Where do we go from here?

We need to choose between using `tuple` or `enumerate_result`, and that type would be used for both the value type and the reference type.

POLL: Does LEWG prefer using `enumerate_result` (Option 1) rather than a `tuple` (Option 2) as the value and reference type of `enumerate_view::iterator`?

The wording provides both options.

Why not `zip(iota, view)`?

Zipping the view with iota does not actually work (see also P2214R0 [2]), and a custom `index_view` would need to be used as the first range composed with `zip`, so a custom `enumerate` view with appropriately named members is not adding a lot of work.

`enumerate` as presented here is slightly less work for the compiler, but both solutions generate similar assembly.

index_type

`index_type` is defined as follow:

- `ranges::range_size_t<Base>` if `Base` models `ranges::sized_range`
- Otherwise, `make_unsigned_t<ranges::range_difference_t<Base>>`

This is consistent with ranges-v3 and allows the view to support both sized and non-sized ranges.

Performance

An optimizing compiler can generate the same machine code for `views::enumerate` as it would for an equivalent `for` loop. Compiler Explorer [Editor's note: This implementation is a prototype not fully reflective of the proposed design] .

Implementation

This proposal has been implemented (Github) There exist an implementation in ranges-v3 (where the enumerate view uses zip_with and a pair value type).

Proposal

We propose a view `enumerate` whose value type is a struct with 2 members `index` and `value` representing respectively the position and value of the elements in the adapted range.

Wording

[Editor's note: This wording covers 2 options, depending LEWG wishes.
In blue, Option 1 adds `enumerate_result` as the value and reference types of `enumerate_view`.
In brown, Option 2 uses std::tuple instead.
Common wording in blue-green.]

� Header <ranges> synopsis [ranges.syn]

```
template<class R>
using keys_view = elements_view<R, 0>;                                    // freestanding
template<class R>
using values_view = elements_view<R, 1>;                                  // freestanding

namespace views {
    template<size_t N>
    inline constexpr unspecified elements = unspecified;                   // freestanding
    inline constexpr auto keys = elements<0>;                               // freestanding
    inline constexpr auto values = elements<1>;                             // freestanding
}


template <typename Index, class Value>
requires is-integer-like<Index>
struct enumerate_result; // freestanding

template<size_t I, class Index, class Value>
constexpr tuple_element_t<I, enumerate_result<Index, Value>>&
get(enumerate_result<Index, Value>&) noexcept;  // freestanding

template<size_t I, class Index, class Value>
constexpr tuple_element_t<I, enumerate_result<Index, Value>>&&
get(enumerate_result<Index, Value>&&) noexcept; // freestanding

template<size_t I, class Index, class Value>
constexpr const tuple_element_t<I, enumerate_result<Index, Value>>&
get(const enumerate_result<Index, Value>&) noexcept; // freestanding

template<size_t I, class Index, class Value>
constexpr const tuple_element_t<I, enumerate_result<Index, Value>>&&
get(const enumerate_result<Index, Value>&&) noexcept; // freestanding

template <input_range View>
requires view<View>
class enumerate_view; // freestanding

namespace views { inline constexpr unspecified enumerate = unspecified; }      // freestanding
```

```
// ??, zip view
template<input_range... Views>
requires (view<Views> && ...) && (sizeof...(Views) > 0)
class zip_view;                                                        // freestanding

template<class... Views>
inline constexpr bool enable_borrowed_range<zip_view<Views...>> =     // freestanding
(enable_borrowed_range<Views> && ...);

namespace views { inline constexpr unspecified zip = unspecified; }   // freestanding

[...]

}

namespace std {
    namespace views = ranges::views;                                   // freestanding

    template<class T> struct tuple_size;                               // freestanding
    template<size_t I, class T> struct tuple_element;                  // freestanding

    template<class I, class S, ranges::subrange_kind K>
    struct tuple_size<ranges::subrange<I, S, K>>                       // freestanding
    : integral_constant<size_t, 2> {};
    template<class I, class S, ranges::subrange_kind K>
    struct tuple_element<0, ranges::subrange<I, S, K>> {               // freestanding
        using type = I;                                                // freestanding
    };
    template<class I, class S, ranges::subrange_kind K>
    struct tuple_element<1, ranges::subrange<I, S, K>> {               // freestanding
        using type = S;                                                // freestanding
    };
    template<class I, class S, ranges::subrange_kind K>
    struct tuple_element<0, const ranges::subrange<I, S, K>> {         // freestanding
        using type = I;                                                // freestanding
    };
    template<class I, class S, ranges::subrange_kind K>
    struct tuple_element<1, const ranges::subrange<I, S, K>> {         // freestanding
        using type = S;                                                // freestanding
    };



    template<class Index, class Value>
    struct tuple_size<ranges::enumerate_result<Index, Value>> : integral_constant<size_t, 2> { };

    template<size_t I, class Index, class Value>
    struct tuple_element<I, ranges::enumerate_result<Index, Value>> {
        using type =  see below;
```

```
    };

    template<template<class> class TQual, template<class> class UQual,
    typename Index, typename Type, typename Type2>
    struct basic_common_reference<ranges::enumerate_result<Index, Type>,
    ranges::enumerate_result<Index, Type2>, TQual, UQual> {
        using type = ranges::enumerate_result<Index, std::common_reference_t<Type, Type2>>;
    };


    struct from_range_t { explicit from_range_t() = default; };                  // freestanding
    inline constexpr from_range_t from_range{};                                  // freestanding
}
```

� Concept *tuple-like* [tuple.like]

```
template <typename T>
concept tuple-like = see below;          // exposition only
```

A type T models and satisfies the exposition-only concept *tuple-like* if `std::remove_cvref_t<T>` is a specialization of `array`, `pair`, `tuple`, `ranges::enumerate_result`, or `ranges::subrange`.

[Editor's note: Add the following new section before [range.zip]]

� Enumerate view [range.enumerate]

� Overview [range.enumerate.overview]

`enumerate_view` presents a `view` with a value type that represents both the position and value of the adapted `view`'s value-type.

The name `views::enumerate` denotes a range adaptor object. Given the subexpressions `E` the expression `views::enumerate(E)` is expression-equivalent to `enumerate_view{E}`.

[*Example:*

```
vector<int> vec{ 1, 2, 3 };
for (auto [index, value] : enumerate(vec) )
    cout << index << ":" << value ' '; // prints: 0:1 1:2 2:3
```

— *end example*]

[Editor's note: The following wording (in blue) is for Option 1: Use a distinct type for enumerate result]

� Class template `enumerate_result` [range.enumerate.result]

```
namespace std {

namespace ranges {
```

```
    template <integral Index, class Value>
    struct enumerate_result {
        Index index;
        Value value;

        constexpr bool operator==(const enumerate_result &) const = default;

        template<typename OtherValue>
        explicit(see below) operator enumerate_result<Index, OtherValue>() const &;
        template<typename OtherValue>
        explicit(see below) operator enumerate_result<Index, OtherValue>() &;
        template<typename OtherValue>
        explicit(see below) operator enumerate_result<Index, OtherValue>() const &&;
        template<typename OtherValue>
        explicit(see below) operator enumerate_result<Index, OtherValue>() &&;
    };
}


}
```

```
template<typename OtherValue>
explicit(see below) operator enumerate_result<Index, OtherValue>() const &;
template<typename OtherValue>
explicit(see below) operator enumerate_result<Index, OtherValue>() &;
template<typename OtherValue>
explicit(see below) operator enumerate_result<Index, OtherValue>() const &&;
template<typename OtherValue>
explicit(see below) operator enumerate_result<Index, OtherValue>() &&;
```

Let ValueRef be the type `COPYCV(decltype(*this), Value)&` for &-qualified overload, and `COPYCV(decltype(*this), Value)` overloads. ([[meta.trans.other]]).

Constraints:

- `std::is_constructible_v<OtherValue, ValueRef>` is `true`, and
- `std::reference_constructs_from_temporary_v<OtherValue, ValueRef>` is `false`.

Returns: `{index, OtherValue(std::forward<ValueRef>(value))}`.

Remarks:

The expression inside `explicit` is equivalent to:

`!std::is_convertible_v<ValueRef, OtherValue>`.

� Tuple protocol for `enumerate_result` [range.enumerate.result.tuple]

```
template<size_t I, class Index, class Value>
struct tuple_element<I, ranges::enumerate_result<Index, Value>> {
    using type = see below;
```

```
};
```

Mandates: `I < 2`.

Type: The type `Index` if `I` is `0`, otherwise the type `Value`.

```
template<size_t I, class Index, class Value>
constexpr tuple_element_t<I, enumerate_result<Index, Value>>&
get(enumerate_result<Index, Value>& r) noexcept;

template<size_t I, class Index, class Value>
constexpr tuple_element_t<I, enumerate_result<Index, Value>>&&
get(enumerate_result<Index, Value>&& r) noexcept;

template<size_t I, class Index, class Value>
constexpr const tuple_element_t<I, enumerate_result<Index, Value>>&
get(const enumerate_result<Index, Value>& r) noexcept;

template<size_t I, class Index, class Value>
constexpr const tuple_element_t<I, enumerate_result<Index, Value>>&&
get(const enumerate_result<Index, Value>&& r) noexcept;
```

Mandates: `I < 2`. *Returns:*

- if I is 0, returns a reference to `r.index`.
- if I is 1, returns a reference to `r.value`.

� Class template `enumerate_view` [range.enumerate.view]

```
template<input_range V>
requires view<V>
class enumerate_view : public view_interface<enumerate_view<V>> {

  private:
    V base_ = {};

    template <bool Const>
    class iterator; // exposition only
    template <bool Const>
    struct sentinel; // exposition only

   public:

    constexpr enumerate_view() = default;
    constexpr enumerate_view(V base);

    constexpr auto begin() requires (!simple_view<V>)
    { return iterator<false>(ranges::begin(base_), 0); }

    constexpr auto begin()\textbf{} const requires simple_view<V>
    { return iterator<true>(ranges::begin(base_), 0); }
```

```
        constexpr auto end()
        { return sentinel<false>{end(base_)}; }

        constexpr auto end()
        requires common_range<V> && sized_range<V>
        { return iterator<false>{ranges::end(base_),
                 static_cast<range_difference_t<V>>(size()) }; }

        constexpr auto end() const
        requires range<const V>
        { return sentinel<true>{ranges::end(base_)}; }

        constexpr auto end() const
        requires common_range<const V> && sized_range<V>
        { return iterator<true>{ranges::end(base_),
                 static_cast<range_difference_t<V>>(size())}; }

        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 V base() const & requires copy_constructible<V> { return base_; }
        constexpr V base() && { return move(base_); }
    };
    template<class R>
    enumerate_view(R&&) -> enumerate_view<views::all_t<R>>;
}
```

```
    constexpr enumerate_view(V base);
```

Effects: Initializes *base_* with `move(base)`.

� Class `enumerate_view::`*`iterator`* [range.enumerate.iterator]

```
namespace std::ranges {
    template<input_range V>
    requires view<V>
    template<bool Const>
    class enumerate_view<V>::iterator {

        using Base = conditional_t<Const, const V, V>;
        using index_type = see below;

        iterator_t<Base> current_ = iterator_t<Base>();
        index_type  pos_ = 0;
```

```
public:
  using iterator_category = input_iterator_tag;
  using iterator_concept  = see below;

  [Editor's note: The following wording (in blue) is for Option 1]


  using reference = enumerate_result<index_type, range_reference_t<Base>>;
  using value_type = enumerate_result<index_type, range_value_t<Base>>;

  [Editor's note: The following wording (in brown) is for Option 2]

  using reference = tuple<index_type, range_reference_t<Base>>;
  using value_type = tuple<index_type, range_value_t<Base>>;

  using difference_type = range_difference_t<Base>;

  iterator() = default;
  constexpr explicit iterator(iterator_t<Base> current, range_difference_t<Base> pos);
  constexpr iterator(iterator<!Const> i)
  requires Const && convertible_to<iterator_t<V>, iterator_t<Base>>;


  constexpr iterator_t<Base> base() const&
  requires copyable<iterator_t<Base>>;
  constexpr iterator_t<Base> base() &&;

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

  constexpr iterator& operator++();
  constexpr void operator++(int) requires (!forward_range<Base>);
  constexpr iterator operator++(int) requires forward_range<Base>;

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

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

  constexpr decltype(auto) operator[](difference_type n) const
  requires random_access_range<Base>
  { return reference{static_cast<difference_type>(pos_ + n), *(current_ + n) }; }


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

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

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

        friend constexpr decltype(auto) iter_move(const iterator& i)
        noexcept(noexcept(ranges::iter_move(i.current_))) {
            return enumerate_result<index_type,
                        range_rvalue_reference_t<Base>>{pos, ranges::iter_move(i.current_)};
        }

        friend constexpr void iter_swap(const iterator& x, const iterator& y)
        noexcept(noexcept(ranges::iter_swap(x.current_, y.current_)))
        requires indirectly_swappable<iterator_t<Base>> {
            std::swap(x.pos_, y.pos_);
            ranges::iter_swap(x.current_, y.current_);
        }
    };

    };
}
```

`iterator::iterator_concept` is defined as follows:

- If *Base* models `random_access_range`, then `iterator_concept` denotes `random_access_iterator_tag`.
- Otherwise, if *Base* models `bidirectional_range`, then `iterator_concept` denotes `bidirectional_iterator_tag`.
- Otherwise, if *Base* models `forward_range`, then `iterator_concept` `iterator_concept` denotes `forward_iterator_tag`.
- Otherwise, `iterator_concept` denotes `input_iterator_tag`.

`iterator::index_type` is defined as follow:

- `ranges::range_size_t<Base>` if *Base* models `ranges::sized_range`
- Otherwise, `make-unsigned-like-t<ranges::range_difference_t<Base>>`

```
constexpr explicit iterator(iterator_t<Base> current, range_difference_t<Base> pos = 0);
```

Effects: Initializes `current_` with `move(current)` and *pos* with `static_cast<index_type>(pos)`.

```
constexpr iterator(iterator<!Const> i)
requires Const && convertible_to<iterator_t<V>, iterator_t<Base>>;
```

Effects: Initializes *current_* with `move(i.current_)` and *pos* with `i.pos`.

```
constexpr iterator_t<Base> base() const&
requires copyable<iterator_t<Base>>;
```

Effects: Equivalent to: `return current_;`

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

Effects: Equivalent to: `return move(current_);`

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

Effects: Equivalent to:

```
++pos_;
++current_;
return *this;
```

```
constexpr void operator++(int) requires (!forward_range<Base>);
```

Effects: Equivalent to:

```
++pos_;
++current_;
```

```
constexpr iterator operator++(int) requires forward_range<Base>;
```

Effects: Equivalent to:

```
auto temp = *this;
++pos;
++current_;
return temp;
```

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

Effects: Equivalent to:

```
--pos_;
--current_;
```

```
return *this;
```

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

Effects: Equivalent to:

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

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

Effects: Equivalent to:

```
current_ += n;
pos_ += n;
return *this;
```

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

Effects: Equivalent to:

```
current_ -= n;
pos_ -= n;
return *this;
```

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

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

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

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

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

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

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

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

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

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

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

Effects: Equivalent to: `return x.current_ <=> y.current_;`

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

Effects: Equivalent to: `return iterator{x} += y;`

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

Effects: Equivalent to: `return y + x;`

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

Effects: Equivalent to: `return iterator{x} -= y;`

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

Effects: Equivalent to: `return x.current_ - y.current_;`

� Class template `enumerate_view::sentinel` [range.enumerate.sentinel]

```
namespace std::ranges {
    template<input_range V, size_t N>
    requires view<V>
    template<bool Const>
    class enumerate_view<V, N>::sentinel {                 // exposition only
        private:
        using Base = conditional_t<Const, const V, V>;      // exposition only
        sentinel_t<Base> end_ = sentinel_t<Base>();         // exposition only
        public:
        sentinel() = default;
        constexpr explicit sentinel(sentinel_t<Base> end);
        constexpr sentinel(sentinel<!Const> other)
        requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;

        constexpr sentinel_t<Base> base() const;

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

        friend constexpr range_difference_t<Base>
        operator-(const iterator<Const>& x, const sentinel& y)
        requires sized_sentinel_for<sentinel_t<Base>, iterator_t<Base>>;
```

```
        friend constexpr range_difference_t<Base>
        operator-(const sentinel& x, const iterator<Const>& y)
        requires sized_sentinel_for<sentinel_t<Base>, iterator_t<Base>>;
    };
}
```

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

Effects: Initializes *end_* with `end`.

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

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

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

Effects: Equivalent to: `return end_;`

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

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

```
friend constexpr range_difference_t<Base>
operator-(const iterator<Const>& x, const sentinel& y)
requires sized_sentinel_for<sentinel_t<Base>, iterator_t<Base>>;
```

Effects: Equivalent to: `return x.current_ - y.end_;`

```
friend constexpr range_difference_t<Base>
operator-(const sentinel& x, const iterator<Const>& y)
requires sized_sentinel_for<sentinel_t<Base>, iterator_t<Base>>;
```

Effects: Equivalent to: `return x.end_ - y.current_;`

Feature test macro

[Editor's note: define `__cpp_lib_ranges_enumerate` set to the date of adoption in `<version>` and `<ranges>`] .

Acknowledgments

Thanks a lot to Tomasz Kamiński for finding defects in the design proposed in earlier revisions, as well as his invaluable wording feedbacks. Thanks to Barry Revzin and Christopher Di Bella for their inputs on the design.

References

[1] Corentin Jabot. P2165R2: Compatibility between tuple, pair and tuple-like objects. https://wg21.link/p2165r2, 6 2021.

[2] Barry Revzin, Conor Hoekstra, and Tim Song. P2214R0: A plan for c++23 ranges. https://wg21.link/p2214r0, 10 2020.

[3] Andrew Tomazos. P1894R0: Proposal of std::upto, std::indices and std::enumerate. https://wg21.link/p1894r0, 10 2019.

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4885