`views::enumerate`

Document #: P2164R4
Date: 2021-04-26
Project: Programming Language C++
Audience: LEWG
Reply-to: Corentin Jabot <corentin.jabot@gmail.com>

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

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

```
std::vector days{"Mon", "Tue",
  "Wed", "Thu", "Fri", "Sat", "Sun"};

int idx = 0;
for(const auto & d : days) {
    print("{} {} \n", idx, d);
    idx++;
}
```

After

```
#include <ranges>

std::vector days{"Mon", "Tue",
  "Wed", "Thu", "Fri", "Sat", "Sun"};

for(const auto & [idx, d]
      : std::views::enumerate(days)) {
    print("{} {} \n", idx, d);
}
```

Motivation

The impossibility to extract an index from a range-based for loop leads to the use of non-range based for loop, 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 `index` 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

The result is ~~a simple aggregate~~ the simple type that would satisfy the `common_-reference` requierements

Following the trend of using meaningful names instead of returning pairs or tuples, this proposal uses structured with named public members.

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

This design was previously discussed by LEWGI in Belfast in the context of P1894R0 [**?**].

Unfortunately, to satisfy the `indirectly_readable` requierements, the `value_type` and `reference_-type` must have a common reference.

This adds some complexity to the design. Below would be the minimum implementation of the value & reference types:

```
template <typename count_type, typename T>
struct result {

    count_type index;
    T value;

    result()  requires std::is_default_constructible_v<T> = default;

    template <typename U>
    requires std::constructible_from<T, U>
    result(count_type index, U&& value)
    noexcept(std::is_nothrow_constructible_v<T, U>)
    :  index(index), value(std::forward<U>(value)) {}

    result(const result  & other) requires std::copyable<T> = default;

    template <typename U>
    requires std::constructible_from<T, U>
    result(enumerate_result_tcount_type, U> && other)
    noexcept(std::is_nothrow_constructible_v<T, U>)
    :  index(other.index), value(std::forward<enumerate_result_tcount_type, U>>(other).value) {};

    template <typename U>
    requires std::constructible_from<T, U>
    result(const enumerate_result_tcount_type, U> & other)
    noexcept(std::is_nothrow_constructible_v<T, U>)
    :  index(other.index), value(other.value) {};

    result  & operator=(const result & other) requires std::assignable_from<T, T> = default;

    template <typename U>
    requires std::convertible_to<U, T>
    result  & operator=(const enumerate_result_tcount_type, U> & other)
    noexcept(noexcept(value  = std::forward<enumerate_result_tcount_type, U>>(other).value)) {
        index  = other.index;
        value  = other.value;
        return *this;
    }

    template <typename U>
    requires std::equality_comparable_with<T,U>
    constexpr bool operator==(const enumerate_result_tcount_type, U> &other) const noexcept {
        return other.index == index && other.value = value;
    }

    template <typename U>
    requires std::three_way_comparable_with<T,U>
    constexpr auto operator<=>(const enumerate_result_tcount_type, U> &other) const noexcept {
        std:: tie (index, value) <=> std:: tie (other.index, other.value);
    }

    template <typename U>
    friend void swap(result & a, enumerate_result_tcount_type, U>& b);
};
```

I think it's very important here not to try to recreate a `pair`. This type should not be a generalpurpose type with different names than pair. As such it has fewer constructors (doesn't need to be constructed from literals), no deduction guides, etc. Missing at the moment is support for `tuple_size`, `tuple_element`, `get`. Ideally convertibility to `tuple` is desired. This could be dealt with something like P2165R0 [**?**] (a paper which needs a redesign to support types like `array`...).

Despite that newly discovered complexity, I remained convince that it's worth the effort to get these names, and several people have expressed to me that they want names ”if we can get them”.

count_type

`count_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

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

� 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]

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

```
namespace std::ranges {


    template <typename count_type, typename T>
    struct enumerate_result_t {

        count_type index;
        T value;

        enumerate_result_t() requires std::is_default_constructible_v<T> = default;

        template <typename U>
        requires std::constructible_from<T, U>
        enumerate_result_t(count_type index, U&& value) noexcept(std::is_nothrow_constructible_v<T, U>);

        enumerate_result_t(const result & other)
        requires std::copyable<T> = default;

        template <typename U>
        requires std::constructible_from<T, U>
        enumerate_result_t(enumerate_result_tcount_type, U> && other)
        noexcept(std::is_nothrow_constructible_v<T, U>);

        template <typename U>
        requires std::constructible_from<T, U>
        enumerate_result_t(const enumerate_result_t<count_type, U> & other)
        noexcept(std::is_nothrow_constructible_v<T, U>);

        enumerate_result_t & operator=(const enumerate_result_t & other)
        requires std::assignable_from<T, T> = default;

        template <typename U>
        requires std::convertible_to<U, T>
        enumerate_result_t & operator=(const enumerate_result_tcount_type, U> & other)
        noexcept(noexcept(value  = std::forward<enumerate_result_t<count_type, U>>(other).value));

        template <typename U>
        requires std::equality_comparable_with<T,U>
        constexpr bool operator==(const enumerate_result_tcount_type, U> &other) const noexcept;

        template <typename U>
        requires std::three_way_comparable_with<T,U>
        constexpr auto operator<=>(const enumerate_result_tcount_type, U> &other) const noexcept;

        template <typename U>
        friend void swap(result & a, enumerate_result_tcount_type, U>& b);
    };

    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() 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 count_type = see below;

        iterator_t<Base> current_ = iterator_t<Base>();
        count_type pos_ = 0;


      public:
        using iterator_category = typename iterator_traits<iterator_t<Base>>::iterator_category;

        using reference = enumerate_result_t<count_type, range_reference_t<Base>>;
        using value_type = enumerate_result_t<count_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>;
    };
}
```

`iterator::count_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>>`

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

Effects: Initializes `current_` with `move(current)` and *pos* with `static_cast<count_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_;`

References

[N4878] Richard Smith *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4878