views::enumerate

Document #: P2164R5
Date: 2021-06-15
Project: Programming Language C++
Audience: LEWG, SG-9
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

R5

Instead of adding complexity to enumerate_result, we assume changes made by P2165R2 [2]. P2165R2 [2] 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).

For simplicity, consistency with zip and carthesian_product and to avoid enumerate_result propagating, the reference type of enumerate_view is enumerate_result and its value type is tuple.

P2165R2 [2] ensures a common reference exists 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.

1
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

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>std::vector days{&quot;Mon&quot;, &quot;Tue&quot;, &quot;Wed&quot;, &quot;Thu&quot;, &quot;Fri&quot;, &quot;Sat&quot;, &quot;Sun&quot;};</code></td>
<td><code>#include &lt;ranges&gt;</code></td>
</tr>
<tr>
<td><code>int idx = 0;</code></td>
<td><code>std::vector days{&quot;Mon&quot;, &quot;Tue&quot;, &quot;Wed&quot;, &quot;Thu&quot;, &quot;Fri&quot;, &quot;Sat&quot;, &quot;Sun&quot;};</code></td>
</tr>
<tr>
<td><code>for(const auto &amp; d : days) {</code></td>
<td><code>for(const auto &amp; e : std::views::enumerate(days)) {</code></td>
</tr>
<tr>
<td>print(&quot;{} {} \n&quot;, idx, d);</td>
<td>print(&quot;{} {} \n&quot;, e.index, e.value);</td>
</tr>
<tr>
<td>idx++;</td>
<td>}</td>
</tr>
</tbody>
</table>

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**

**The reference type is a simple aggregate with name members**

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

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

This design was previously discussed by LEWGI in Belfast in the context of P1894R0 [4], and many people have expressed a desire for such struct with names. Using this struct for both the reference type and the value type would add significant complexity, as the value and reference type need to share a common_reference (see P2164R4 [1]).

Instead, we propose that the reference type is `enumerate_result<index, range_reference_t<Base>>` and the value type is `tuple<index, range_value_t<Base>>`.

With this design, only `get`, `tuple_element`, `tuple_size` need to be implemented for `enumerate_result`, and `enumerate_result` remains a simple aggregate.

This design works nicely with `ranges::to` as it will create a container based on the value type:

```cpp
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>.
```

This gives us some consistency: `enumerate`'s value type is a tuple, similar to that of `zip`, `cartesian_product`, while retaining the ease of use and added benefits of a struct with named members while iterating over an `enumerate_view`.

**Why not just always return a tuple/pair and rely on structure binding?**

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

```cpp
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.

And we can. The proposed design in R5 is not involved. Keep in mind that zipping the view with `iota` does not actually work (see also P2214R0 [3]), 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 if we pursue P2165R2 [2].

Granted, P2165R2 [2] and this paper justify each other, and P2165R2 [2] is not a trivial amount of work. However, P2165 offers further benefits besides enabling a slightly nicer enumerate, so if we think P2165 is generally useful, we can pursue this paper. If we don't, we can quickly respecify enumerate in terms of zip and some index_view, for which we have usage experience.

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: TODO: ranges synopsis]
Enumerate view

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

namespace std::ranges {

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

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

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

    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\langle V\rangle)
{ return iterator\langle false\rangle\langle ranges::begin(base_), 0 \rangle; }

constexpr auto begin() const requires simple_view\langle V\rangle
{ return iterator\langle true\rangle\langle ranges::begin(base_), 0 \rangle; }

constexpr auto end()
{ return sentinel\langle false\rangle\langle end(base_) \rangle; }

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

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

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

constexpr auto size()
requires sized_range\langle V\rangle
{ return ranges::size(base_); }

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

constexpr V base() const & requires copy_constructible\langle V\rangle { return base_; }
constexpr V base() && { return move(base_); }


namespace std {

template<class R>
enumerate_view(R&&) -> enumerate_view<views::all_t\langle R\rangle>;

}

}
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.

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

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

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

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.

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 = typename iterator_traits<iterator_t<Base>>::iterator_category;
  using reference = enumerate_result<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>;
};
)
}

iterator::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>>

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_t<!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:

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

**Effects:** Equivalent to:

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

**References**


