Rangified version of lexicographical_compare_three_way
Document #: P2022R3
Date: 2023-12-13
Project: Programming Language C++
Audience: SG9, LEWG
Reply-to:
Alex Dathskovsky <calebxyz@gmail.com>
Ran Regev <ran.regev@beyeonics.com>

Revision History

R3

- Moved the concept `same_as_any_of` to [concept.same_as_any_of]
- Revisit the the Range Interface

R2

- Fixed wording per mailing list comments

R1

- Added link to github implementation
- Added code example

R0

- initial work

Motivation and Scope

This document adds the wording for ranges::lexicographical_compare_three_way.

Design Decisions

- We explored the following directions and decided to drop them:
    - Having restrictions on the relation between the ranges. We found it unnecessary as the comp predicate glues the ranges together for the needs of this comparison.

    - Returning not only the comparison result but also the iterators to the ranges where the decision was made (returning a result-struct). One can use std::ranges::mismatch [alg.mismatch] for this purpose.
- The chosen direction is as follows:
    - Follow the way `std::lexicographical_compare_three_way` is declared.
    - The Comp function is restricted to return one of the comparison categories, and nothing else. Therefore
        - There is no reason to restrict the relation between the compared ranges in any way.
        - Functions built on top of `ranges::lexicographical_compare_three_way` may restrict their input parameters if required.
        - Functions built on top of `ranges::lexicographical_compare_three_way`, maybe `ranges::sort_three_way()` or alike may stand to leverage the comprehensive information embedded within the return value of `ranges::lexicographical_compare_three_way`. This enriched data can be harnessed to communicate specific outcomes to users. For example, `sort_three_way()` could relay details such as the resultant sorted range being ordered from the smallest to the largest (or vice versa), indicating uniformity among all elements, or signaling an unsortable state within the given range. By tapping into this returned information, these functions can provide users with clear and detailed insights into the operation's conclusion.

Code Example

In [GitHub] branch P2022/master one can build and run [Tests] to experiment with the function

Proposed Wording

Add to [concepts.syn]

```
template<class T, class U>
    concept same_as_any_of = see below;
```

Add to [concept.same_as_any_of]

```
template<
     typename T,
     typename... Us
>
concept same_as_any_of = (same_as<T, Us> or ...); // exposition-only
```

Add to [algorithm.syn]

```
namespace std::ranges {
template<
       input_iterator I1,
       input_iterator I2,
       class Comp,
       class Proj1,
       class Proj2
>
using three-way-order =
       invoke_result_t<
              Comp,
              typename projected<I1, Proj1>::value_type,
              typename projected<I2, Proj2>::value_type
       >; // exposition-only

template<
       std::input_iterator I1,
       std::input_iterator I2,
       class Comp,
       class Proj1,
       class Proj2
>
constexpr bool is-three-way-ordering =
       std::same_as_any_of<
              three-way-order<I1, I2, Comp, Proj1, Proj2>,
              std::strong_ordering,
              std::weak_ordering,
              std::partial_ordering
       >; // exposition-only

template<
       input_iterator I1, sentinel_for<I1> S1,
       input_iterator I2, sentinel_for<I2> S2,
       class Comp = std::compare_three_way,
       class Proj1 = identity,
       class Proj2 = identity
>
requires
       is-three-way-ordering<I1, I2, Comp, Proj1, Proj2>
constexpr auto
lexicographical_compare_three_way(
       I1 first1,
       S1 last1,
       I2 first2,
       S2 last2,
       Comp comp = {},
       Proj1 proj1 = {},
       Proj2 proj2 = {}
) -> common_comparison_category_t<
       decltype(comp(proj1(*first1), proj2(*first2))),
       strong_ordering
>
```

```
template<
	input_range R1,
	input_range R2,
	class Comp = compare_three_way,
	class Proj1 = identity,
	class Proj2 = identity
>
requires
	is-three-way-ordering<iterator_t<R1>, iterator_t<R2>, Comp, Proj1, Proj2>
constexpr auto
ranges::lexicographical_compare_three_way(
	R1&& r1,
	R2&& r2,
	Comp comp = {},
	Proj1 proj1 = {},
	Proj2 proj2 = {}
	) -> common_comparison_category_t<
		decltype(comp(proj1(*ranges::begin(r1)), proj2(*ranges::begin(r2)))),
		strong_ordering
		>;
```

Add to [alg.three.way]

```
template<class InputIterator1, class InputIterator2>
constexpr auto
	lexicographical_compare_three_way(
		InputIterator1 b1,
		InputIterator1 e1,
		InputIterator2 b2,
		InputIterator2 e2
	);

using three-way-order =
	invoke_result_t<
		Comp,
		class projected<I1, Proj1>::value_type,
		class projected<I2, Proj2>::value_type
	>; // exposition-only

template<
	input_iterator I1,
	sentinel_for S1,
	input_iterator I2,
	sentinel_for S2,
	class Comp = std::compare_three_way,
	class Proj1 = identity,
	class Proj2 = identity
>
constexpr bool is-three-way-ordering =
	same-as-any-of<
		lexicographical-compare-three-way-result-t<I1, I2, Comp, Proj1, Proj2>,
		std::strong_ordering,
		std::weak_ordering,
```

```
            std::partial_ordering
      >; //exposition-only

template<
      input_iterator I1,
      sentinel_for S1,
      input_iterator I2,
      sentinel_for S2,
      class Comp = compare_three_way,
      class Proj1 = identity,
      class Proj2 = identity
>
requires
      is-three-way-ordering<I1, I2, Comp, Proj1, Proj2>
constexpr auto
ranges::lexicographical_compare_three_way(
      I1 first1,
      S1 last1,
      I2 first2,
      S2 last2,
      Comp comp = {},
      Proj1 proj1 = {},
      Proj2 proj2 = {}
) -> common_comparison_category_t<
      decltype(comp(proj1(*first1), proj2(*first2))),
      std::strong_ordering
      >;

template<
      input_range R1,
      input_range R2,
      class Comp = compare_three_way,
      class Proj1 = identity,
      class Proj2 = identity
>
requires
      is-three-way-ordering<iterator_t<R1>, iterator_t<R2>, Comp, Proj1, Proj2>
constexpr auto
ranges::lexicographical_compare_three_way(
      R1&& r1,
      R2&& r2,
      Comp comp = {},
      Proj1 proj1 = {},
      Proj2 proj2 = {}
      ) -> common_comparison_category_t<
            decltype(comp(proj1(*ranges::begin(r1)), proj2(*ranges::begin(r2)))),
            strong_ordering
            >;
```

[1] Let N be the minimum integer between `distance(first1,s1)` and `distance(first2,s2)`. Let E(n) be `comp(proj1((first1 + n)), proj2((first2 + n)))`.

[2] — Returns: E(i), where i is the smallest integer in [0, N) such that E(i) != 0 is true, or `distance(first1,s1) <=> distance(first2, s2)` if no such integer exists.
[3] — Complexity: At most N applications of comp, proj1, proj2.

Acknowledgements

Lee-or Saar <Leeor.Saar@beyeonics.com>
Mor Elmaliach <Mor.Elmaliach@beyeonics.com>
Yaron Meister <Yaron.Meister@beyeonics.com>
Ronen Friedman <friedman.ronen@gmail.com>

References

[GitHub] implementation. https://github.com/regevran/IlPapersFork/tree/P2022/master
[Tests] tests. https://github.com/regevran/IlPapersFork/tree/P2022/master/P2022/tests