

Rangified version of `lexicographical_compare_three_way`

Document #: P2794R1
Date: 2023-02-27
Project: Programming Language C++
Audience: SG9
LEWG
Reply-to: Ran Regev
<ran.regev@beyeonics.com>

1 Revision History

- R1
 - Added link to github implementation
 - Added code example
- R0
 - initial work

2 Motivation and Scope

This document adds the wording for `ranges::lexicographical_compare_three_way`

3 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 glue the ranges together to this comparison's needs.
 - Returning not only the comparison result but also the iterators to the ranges where the decision was made (return a result-struct). We couldn't find any useful implementation for these iterators and therefore decided to drop the idea.
- 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` such as (the yet to be defined) `ranges::sort_three_way()` should benefit from the additional information that can be found in the return value of `ranges::lexicographical_compare_three_way`, and even use it to indicate the user that the function ended in a specific state. E.g. `sort_three_way()` may report that the resulted sorted range is sorted from smallest to largest (or largest to smallest), all element are equal or even that the given range is unsortable.

4 Code Example

- In [\[GitHub\]](#) branch P2022/master one can build and run [\[Tests\]](#) to experiment with the function

5 Proposed Wording

5.1 Add to [algorithm.syn]

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

template <typename T, typename... U>
concept same-as-one-of = (same_as<T, U> or ...); // exposition only

template<
    input_iterator I1,
    input_iterator I2,
    class Comp,
    class Proj1,
    class Proj2
>
using lexicographical_compare_three_way_result_t =
    invoke_result_t<
        Comp,
        typename projected<I1, Proj1>::value_type,
        typename projected<I2, Proj2>::value_type
    >; // exposition-only

constexpr bool is_lexicographical_compare_three_way_result_ordering =
    same-as-one-of<
        lexicographical_compare_three_way_result_t<
            I1, I2, Comp, Proj1, Proj2
        >,
        strong_ordering, weak_ordering, 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_lexicographical_compare_three_way_result_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))
    ),
    strong_ordering
>;

template<
    ranges::input_range R1,
    ranges::input_range R2,
    class Comp = compare_three_way,
    class Proj1 = identity,
    class Proj2 = identity
>
requires
    is_lexicographical_compare_three_way_result_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
    >;

```

5.2 Add to §27.8.12 [alg.three.way]

```

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

template <typename T, typename... U>
concept same-as-one-of = (same_as<T, U> or ...); // exposition only

template<
    input_iterator I1,
    input_iterator I2,
    class Comp,
    class Proj1,
    class Proj2
>
using lexicographical_compare_three_way_result_t =
    invoke_result_t<
        Comp,
        typename projected<I1, Proj1>::value_type,
        typename projected<I2, Proj2>::value_type
    >; // exposition-only

constexpr bool is_lexicographical_compare_three_way_result_ordering =

```

```

    same-as-one-of<
        lexicographical-compare-three-way-result-t<
            I1, I2, Comp, Proj1, Proj2
        >,
        strong_ordering, weak_ordering, 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_lexicographical_compare_three_way_result_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))
        ),
        strong_ordering
    >;

template<
    ranges::input_range R1,
    ranges::input_range R2,
    class Comp = compare_three_way,
    class Proj1 = identity,
    class Proj2 = identity
>
requires
    is_lexicographical_compare_three_way_result_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 $\text{distance}(\text{first1}, s1)$ and $\text{distance}(\text{first2}, s2)$. Let $E(n)$ be $\text{comp}(\text{proj1}((\text{first1} + n)), \text{proj2}((\text{first2} + n)))$.
- ² — Returns: $E(i)$, where i is the smallest integer in $[0, N)$ such that $E(i) \neq 0$ is true, or $(\text{distance}(\text{first1}, s1) \leq \text{distance}(\text{first2}, s2))$ if no such integer exists.
- ³ — Complexity: At most N applications of comp , proj1 , proj2 .

6 Acknowledgements

Alex Dathskovsky <calebxyz@gmail.com>
 Avi Korzac <Avi.Korzac@beyeonics.com>
 Lee-or Saar <Leeor.Saar@beyeonics.com>
 Mor Elmaliach <Mor.Elmaliach@beyeonics.com>
 Yaron Meister <Yaron.Meister@beyeonics.com>

7 References

[GitHub] Ran Regev. implementation. <https://github.com/regevrn/II PapersFork/tree/P2022/master>

[Tests] Regev and Ran. tests. <https://github.com/regevrn/II PapersFork/tree/P2022/master/P2022/tests>