`find_backward`

Document Number: P1223R1
Date: 2018-10-02
Reply to: Zach Laine whatwasthataddress@gmail.com
Audience: LEWG

1 Revisions

1.1 Changes from R0

- Base synopsis on The One Ranges Proposal (P0896R4).
- Drop `std`-namespace overloads.
- Drop `find_not()` and `find_not_backward()`.

2 Introduction

Sometimes you need to search backward. This is often awkward to do with `find` and `make_reverse_iterator`. We should have first-class algorithms to turn this:

```
while (it-- != first) {
    if (*it == x) {
        // Use it here...
    }
}
```

into this:

```
auto it = std::find_backward(first, it, x);
// Use it here...
```

3 Motivation and Scope

Consider how finding the last element that is equal to ‘x‘ in a range is typically done (for all the examples below, we assume a valid range of elements `[first, last)`, and an iterator `it` within that range):

```
while (it-- != first) {
    if (*it == x) {
        // Use it here...
    }
}
```

Raw loops are icky though. Perhaps we should do a bit of extra work to allow the use of `find()`:

```
auto rfirst = std::make_reverse_iterator(it);
auto rlast = std::make_reverse_iterator(first);
auto it = std::find(rfirst, rlast, x);
// Use it here...
```

That seems nicer in that there is no raw loop, but it requires an unpleasant amount of typing (and an associated lack of clarity).

Consider this instead:

```
auto it = std::find_backward(first, it, x);
// Use it here...
```

That's better! It's a lot less verbose.

Let's consider for a moment the lack of clarity of the `make_reverse_iterator()` code. In a typical use of `find()`, I search forward from the element I start from, including the element itself:

```
auto it = std::find(it, last, x); // Includes examination of *it.
```

However, using finding in reverse in the middle of a range leaves out the element pointed to by the current iterator:

```
auto it = std::find( // Skips *it entirely.
    std::make_reverse_iterator(first),
    std::make_reverse_iterator(it),
    x);
```

That leads to code like this:

```
auto it = std::find( // Includes *it again!
    std::make_reverse_iterator(first),
    std::make_reverse_iterator(std::next(it)),
    x);
```

Though this looks like an off-by-one error. is is correct. Moreover, even though the use of `next()` is correct, it gets lost in noise of the rest of the code, since it is so verbose. Use `find_backward()` makes things clearer:

```
// Search, but don't include *it.
auto it_1 = std::find_backward(first, it, x);

// Search, and include *it.
auto it_2 = std::find_backward(first, stds::next(it), x);
```

The use of `next()` may at first appear like a mistake, until the reader takes a moment to think things through. In the `reverse_iterator` version, this correctness is a lot harder to readily grasp.

4 Proposed Design

4.1 Design

This paper proposes to introduce only the `std::range` overloads of the functions `find_backward()`, `find_if_backward()`, `find_if_not_backward()`. The following synopsis has interface details. Note that the iterator-based overloads do not take an iterator-sentinel pair; this is not suitable for an algorithm that operates in reverse.

4.1.1 `flat_set` Synopsis

```
namespace std { namespace ranges {

    template<InputIterator I, class T, class Proj = identity>
    requires IndirectRelation<
        ranges::equal_to<>,
        projected<I, Proj>,
        const T*>
    constexpr I
    find_backward(I first, I last, const T& value, Proj proj = {});

    template<InputRange R, class T, class Proj = identity>
    requires IndirectRelation<
        ranges::equal_to<>,
        projected<iterator_t<R>, Proj>,
        const T*>
    constexpr safe_iterator_t<R>
    find_backward(R&& r, const T& value, Proj proj = {});

    template<
```

```
        InputIterator I,
        class Proj = identity,
        IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_if_backward(I first, I last, Pred pred, Proj proj = {});

    template<
        InputRange R,
        class Proj = identity,
        IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
    find_if_backward(R&& r, Pred pred, Proj proj = {});

    template<
        InputIterator I,
        class Proj = identity,
        IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I
    find_if_not_backward(I first, I last, Pred pred, Proj proj = {});

    template<
        InputRange R,
        class Proj = identity,
        IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
    find_if_not_backward(R&& r, Pred pred, Proj proj = {});

}}
```

5 Acknowledgements

Thanks to Alisdair Meredith and Marshall Clow for encouraging this submission.