1 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:

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

into this:

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

2 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):

```cpp
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()`:

```cpp
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 has major drawbacks. First, it requires an unpleasant amount of typing. Second, it is less efficient than forward-iterator `find()`, since `reverse_iterator` calls its base-iterator’s `operator--()` in most of its member functions before doing the work that the member function requires.

Consider this instead:

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

That’s better! It’s a lot less verbose and is more efficient too.

Another drawback is 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:

```cpp
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.

2.0.1 find_not()

One more thing. Consider this use of find():

std::vector<int> vec = {1, 1, 2};
auto it = std::find(vec.begin(), vec.end(), 1);

This gives us the first occurrence of 1 in vec. What if we want to find the first occurrence of any number besides 1 in vec? We have to write an unfortunate amount of code:

std::vector<int> vec = {1, 1, 2};
auto it = std::find_if(vec.begin(), vec.end(), [](int i) { return i != 1; });

With find_not() the code gets much more terse:

std::vector<int> vec = {1, 1, 2};
auto it = std::find_not(vec.begin(), vec.end(), 1);

The existing find variants are: find(), find_if(), and find_if_not(). It seems natural to also have find_not(), for the very reason that we have find_if_not() — to avoid having to write a lambda to wrap the negation of the find condition.

3 Proposed Design

3.1 Design

This paper proposes to introduce iterator-based and range-based overloads of the functions find_backward(), find_not_backward(), find_if_backward(), find_if_not_backward(), and find_not(). The following synopsis has interface details. Note that the iterator-based *_backward overloads in namespace ranges do not take an iterator-sentinel pair; this is not suitable for an algorithm that operates in reverse. find_not(), being the only forward-operating algorithm proposed, does have a sentinel-accepting overload.

3.1.1 flat_set Synopsis

namespace std {

    template<typename BidirectionalIterator, typename T>
    constexpr BidirectionalIterator find_backward(BidirectionalIterator first,
    BidirectionalIterator last,
    const T & value);
template<typename BidirectionalIterator, typename T>
constexpr BidirectionalIterator find_not_backward(BidirectionalIterator first,
BidirectionalIterator last,
const T & value);

template<typename BidirectionalIterator, typename Pred>
constexpr BidirectionalIterator find_if_backward(BidirectionalIterator first,
BidirectionalIterator last,
Pred p);

namespace ranges {

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

template<BidirectionalRange Rng, class T, class Proj = identity>
constexpr requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<Rng>, Proj>, const T *>
safe_iterator_t<Rng>
find_not_backward(Rng && rng, const T & value, Proj proj = Proj{});

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

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

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

4 Acknowledgements

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