Document Number: P1223R5
Date: 2022-06-17
Reply to: Zach Laine
whatwasthataddress@gmail.com
Audience: LEWG, LWG

`find_last`

Wording in this paper applies to N4820.

Contents

0.1 Revisions

0.1.1 Changes from R3

— Update wording based on LWG review: reintroduce motivation section; add feature test macro; use final concept names; safe_iterator -> borrowed_iterator; IndirectRelation -> indirect_binary_predicate; add whitespace; change the wording to reflect the way `find()` is specified post boolean-testable paper.

— Remove the word any in wording based on LWG 2020-12-11 review.

— add design considerations to address return value api issue raised in LWG review.

— add results of SG9 and LEWG polling on api change.

— replaced outdated `sentinel<I>` with `sentinel_for<I>`.

— update wording for new api replacing iterator returns with range return values.

0.1.2 Changes from R2

— Update wording based on Cologne meeting LWG small group review.

0.1.3 Changes from R1

— Change `find_backward()` to `find_last()`.

— Wording.

0.1.4 Changes from R0

— Base synopsis on The One Ranges Proposal (P0896R4).

— Drop `std`-namespace overloads.

— Drop `find_not()` and `find_not_backward()`.

0.2 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):

```
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_last(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_last()` makes things clearer:

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

// Search, and include *it.
auto it_2 = std::find_last(first, std::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.

0.3 Design Considerations

During LWG review it was noted that the original design does not account for the principle of preserving useful information. Specifically, with sentinal-based ranges `find_last()` needs to calculate the end of the range, but the api as specified does not return the end. At the time [1], a small group had consensus to ask LEWG to change the return to include the end.

To incorporate the calculation of the end of the range would require changing the signature from:

```
//current wording
template<forward_iterator I, sentinel<I> S, class T, class Proj = identity>
  requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*>
  constexpr I find_last(I first, S last, const T& value, Proj proj = {});

template<forward_range R, class T, class Proj = identity>
  requires indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>
  constexpr borrowed_iterator_t<R>
    find_last(R&& r, const T& value, Proj proj = {});
```

to:

```
// possible new wording
template<forward_iterator I, sentinel<I> S, class T, class Proj = identity>
  requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*>
  constexpr subrange<I> find_last(I first, S last, const T& value, Proj proj = {});

template<forward_range R, class T, class Proj = identity>
  requires indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>
  constexpr borrowed_subrange_t<R>
    find_last(R&& r, const T& value, Proj proj = {});
```

where the subrange would be `[i, last)` when found and `[last, last)` when not found.

But there's a downside to this design. The api is not inconsistent with the other find algorithms `find and find_first`. Also, the usage in a typical use case where only the found iterator is used the user has to discard the extra return information. This was discussed at some length on the LEWG mailing list with only moderate support for the proposed change [2].

Still, the author feels the change is an improvement and should be made.

SG9 discussed the paper on 2022-04-11 and was weakly in favor of the proposed change [3].

Poll: We recommend making the change proposed in section 0.3 of P1223R3 for `find_last`, (making `find_last` returning a subrange) for C++23.

SF	F	N	A	SA
1	4	5	1	0

Outcome: Weak consensus in favor

LEWG mailing list review as of 2022-04-15 had 5 in favor and no against.

0.4 Feature-Test Macro

In addition to the wording that follows, add a new feature-test macro `__cpp_lib_find_last`, with value `xxxxxx`. This macro should also be defined in `<algorithm>`.

1) Dec 2020 LWG review notes https://wiki.edg.com/bin/view/Wg21fall2020/P1223-2020-12-11
2) LEWG list discussion 2021-12 http://lists.isocpp.org/lib-ext/2021/12/21613.php
3) SG9 results https://github.com/cplusplus/papers/issues/149#issuecomment-1097082737

25 Algorithms library [algorithms]

25.4 Header <algorithm> synopsis [algorithm.syn]

```
#include <initializer_list>

namespace std {
  // 25.5, non-modifying sequence operations

  // 25.5.5, find
  template<class InputIterator, class T>
    constexpr InputIterator find(InputIterator first, InputIterator last,
                                 const T& value);
  template<class ExecutionPolicy, class ForwardIterator, class T>
    ForwardIterator find(ExecutionPolicy&& exec, // see ??
                         ForwardIterator first, ForwardIterator last,
                         const T& value);
  template<class InputIterator, class Predicate>
    constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                    Predicate pred);
  template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    ForwardIterator find_if(ExecutionPolicy&& exec, // see ??
                            ForwardIterator first, ForwardIterator last,
                            Predicate pred);
  template<class InputIterator, class Predicate>
    constexpr InputIterator find_if_not(InputIterator first, InputIterator last,
                                        Predicate pred);
  template<class ExecutionPolicy, class ForwardIterator, class Predicate>
    ForwardIterator find_if_not(ExecutionPolicy&& exec, // see ??
                                ForwardIterator first, ForwardIterator last,
                                Predicate pred);

  namespace ranges {
    template<input_iterator I, sentinel_for<I> S, class T, class Proj = identity>
      requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*>
      constexpr I find(I first, S last, const T& value, Proj proj = {});
    template<input_range R, class T, class Proj = identity>
      requires indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>
      constexpr borrowed_iterator_t<R>
        find(R&& r, const T& value, Proj proj = {});
    template<input_iterator I, sentinel_for<I> S, class Proj = identity,
             indirect_unary_predicate<projected<I, Proj>> Pred>
      constexpr I find_if(I first, S last, Pred pred, Proj proj = {});
    template<input_range R, class Proj = identity,
             indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
      constexpr borrowed_iterator_t<R>
        find_if(R&& r, Pred pred, Proj proj = {});
    template<input_iterator I, sentinel_for<I> S, class Proj = identity,
             indirect_unary_predicate<projected<I, Proj>> Pred>
      constexpr I find_if_not(I first, S last, Pred pred, Proj proj = {});
    template<input_range R, class Proj = identity,
             indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
```

```
      constexpr borrowed_iterator_t<R>
        find_if_not(R&& r, Pred pred, Proj proj = {});
  }


  // 25.5.6, find last
  namespace ranges {
    template<forward_iterator I, sentinel_for<I> S, class T, class Proj = identity>
      requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*>
      constexpr subrange<I> find_last(I first, S last, const T& value, Proj proj = {});
    template<forward_range R, class T, class Proj = identity>
      requires indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>
      constexpr borrowed_subrange_t<R>
        find_last(R&& r, const T& value, Proj proj = {});
    template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
             indirect_unary_predicate<projected<I, Proj>> Pred>
      constexpr subrange<I> find_last_if(I first, S last, Pred pred, Proj proj = {});
    template<forward_range R, class Proj = identity,
             indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
      constexpr borrowed_subrange_t<R>
        find_last_if(R&& r, Pred pred, Proj proj = {});
    template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
             indirect_unary_predicate<projected<I, Proj>> Pred>
      constexpr subrange<I> find_last_if_not(I first, S last, Pred pred, Proj proj = {});
    template<forward_range R, class Proj = identity,
             indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
      constexpr borrowed_subrange_t<R>
        find_last_if_not(R&& r, Pred pred, Proj proj = {});
  }

}
```

25.5 Non-modifying sequence operations [alg.nonmodifying]

25.5.5 Find [alg.find]

```
template<class InputIterator, class T>
  constexpr InputIterator find(InputIterator first, InputIterator last,
                               const T& value);
template<class ExecutionPolicy, class ForwardIterator, class T>
  ForwardIterator find(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                       const T& value);

template<class InputIterator, class Predicate>
  constexpr InputIterator find_if(InputIterator first, InputIterator last,
                                  Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if(ExecutionPolicy&& exec, ForwardIterator first, ForwardIterator last,
                          Predicate pred);

template<class InputIterator, class Predicate>
  constexpr InputIterator find_if_not(InputIterator first, InputIterator last,
                                      Predicate pred);
template<class ExecutionPolicy, class ForwardIterator, class Predicate>
  ForwardIterator find_if_not(ExecutionPolicy&& exec,
                              ForwardIterator first, ForwardIterator last,
```

```
                                Predicate pred);

template<InputIterator I, sentinel_for<I> S, class T, class Proj = identity>
 requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*>
 constexpr I ranges::find(I first, S last, const T& value, Proj proj = {});
template<input_range R, class T, class Proj = identity>
 requires indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>
 constexpr borrowed_iterator_t<R>
   ranges::find(R&& r, const T& value, Proj proj = {});

template<InputIterator I, sentinel_for<I> S, class Proj = identity,
         indirect_unary_predicate<projected<I, Proj>> Pred>
  constexpr I ranges::find_if(I first, S last, Pred pred, Proj proj = {});
template<input_range R, class Proj = identity,
         indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
 constexpr borrowed_iterator_t<R>
   ranges::find_if(R&& r, Pred pred, Proj proj = {});

template<InputIterator I, sentinel_for<I> S, class Proj = identity,
         indirect_unary_predicate<projected<I, Proj>> Pred>
 constexpr I ranges::find_if_not(I first, S last, Pred pred, Proj proj = {});
template<input_range R, class Proj = identity,
         indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
 constexpr borrowed_iterator_t<R>
   ranges::find_if_not(R&& r, Pred pred, Proj proj = {});
```

1 Let E be:

(1.1) — `*i == value` for `find`,

(1.2) — `pred(*i) != false` for `find_if`,

(1.3) — `pred(*i) == false` for `find_if_not`,

(1.4) — `bool(invoke(proj, *i) == value)` for `ranges::find`;

(1.5) — `bool(invoke(pred, invoke(proj, *i)))` for `ranges::find_if`;

(1.6) — `bool(!invoke(pred, invoke(proj, *i)))` for `ranges::find_if_not`.

2 *Returns:* The first iterator `i` in the range `[first,last)` for which E is `true`. Returns `last` if no such iterator is found.

3 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

25.5.6 Find last [alg.find.last]

```
template<forward_iterator I, sentinel_for<I> S, class T, class Proj = identity>
 requires indirect_binary_predicate<ranges::equal_to, projected<I, Proj>, const T*>
 constexpr subrange<I> ranges::find_last(I first, S last, const T& value, Proj proj = {});
template<forward_range R, class T, class Proj = identity>
 requires indirect_binary_predicate<ranges::equal_to, projected<iterator_t<R>, Proj>, const T*>
 constexpr borrowed_subrange_t<R>
   ranges::find_last(R&& r, const T& value, Proj proj = {});
template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
         indirect_unary_predicate<projected<I, Proj>> Pred>
 constexpr subrange<I> ranges::find_last_if(I first, S last, Pred pred, Proj proj = {});
template<forward_range R, class Proj = identity,
         indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
 constexpr borrowed_subrange_t<R>
```

```
  ranges::find_last_if(R&& r, Pred pred, Proj proj = {});
template<forward_iterator I, sentinel_for<I> S, class Proj = identity,
         indirect_unary_predicate<projected<I, Proj>> Pred>
 constexpr subrange<I> ranges::find_last_if_not(I first, S last, Pred pred, Proj proj = {});
template<forward_range R, class Proj = identity,
         indirect_unary_predicate<projected<iterator_t<R>, Proj>> Pred>
 constexpr borrowed_subrange_t<R>
  ranges::find_last_if_not(R&& r, Pred pred, Proj proj = {});
```

1 Let E be:

(1.1) — `bool(invoke(proj, *i) == value)` for `ranges::find_last`;

(1.2) — `bool(invoke(pred, invoke(proj, *i)))` for `ranges::find_last_if`;

(1.3) — `bool(!invoke(pred, invoke(proj, *i)))` for `ranges::find_last_if_not`.

2 *Returns:* Let `i` be the last iterator in the range `[first,last)` for which E is `true`. Returns `{i, last}`, or `{last, last}` if no such iterator is found.

3 *Complexity:* At most `last - first` applications of the corresponding predicate and projection.

25.6 Acknowledgements

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

Thanks to Jeff Garland for helping shepherding the paper through final LEWG and LWG reviews.