find_last

Wording in this paper applies to N4800.

Contents

Contents

0.1 Revisions .......................... i

25 Algorithms library .................. 2
  25.4 Header <algorithm> synopsis ...................... 2
  25.5 Non-modifying sequence operations .................. 3
  25.6 Acknowledgements .......................... 6
0.1 Revisions

0.1.1 Changes from R1

— Change \texttt{find\_backward()} to \texttt{find\_last()}.

— Wording.

0.1.2 Changes from R0

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

— Drop std-namespace overloads.

— Drop \texttt{find\_not()} and \texttt{find\_not\_backward()}.
# 25 Algorithms library

## 25.4 Header `<algorithm>` synopsis

```cpp
#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<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
        requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>  
        constexpr I find(I first, S 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(R&& r, const T& value, Proj proj = {});

        template<InputIterator I, Sentinel<I> S, class Proj = identity,
                  IndirectUnaryPredicate<projected<I, Proj>> Pred>
        constexpr I find_if(I first, S 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(R&& r, Pred pred, Proj proj = {});

        template<InputIterator I, Sentinel<I> S, class Proj = identity,
                  IndirectUnaryPredicate<projected<I, Proj>> Pred>
        constexpr I find_if_not(I first, S 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(R&& r, Pred pred, Proj proj = {});
    }
}
```
constexpr safe_iterator_t<R>
    find_if_not(R&& r, Pred pred, Proj proj = {});
}

// 25.5.6, find last
namespace ranges {
    template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>  
    constexpr I find_last(I first, S last, const T& value, Proj proj = {});
    template<ForwardRange R, class T, class Proj = identity>  
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T*>  
    constexpr safe_iterator_t<R>
        find_last(R&& r, const T& value, Proj proj = {});
    template<ForwardIterator I, Sentinel<I> S, class Proj = identity,  
        IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_last_if(I first, S last, Pred pred, Proj proj = {});
    template<ForwardRange R, class Proj = identity,  
        IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
        find_last_if(R&& r, Pred pred, Proj proj = {});
    template<ForwardIterator I, Sentinel<I> S, class Proj = identity,  
        IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_last_if_not(I first, S last, Pred pred, Proj proj = {});
    template<ForwardRange R, class Proj = identity,  
        IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R>
        find_last_if_not(R&& r, Pred pred, Proj proj = {});
}

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

namespace ranges {
    template<InputIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>'
    constexpr I find(I first, S 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(R&& r, const T& value, Proj proj = {});
    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_if(I first, S 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(R&& r, Pred pred, Proj proj = {});
    template<InputIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_if_not(I first, S 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(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)  
    — invoke(proj, *i) == value for ranges::find,
(1.5)  
    — invoke(pred, invoke(proj, *i)) != false for ranges::find_if,
(1.6)  
    — invoke(pred, invoke(proj, *i)) == false 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 any projection.

25.5.6 Find last

namespace ranges {
    template<ForwardIterator I, Sentinel<I> S, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<I, Proj>, const T*>'
    constexpr I find_last(I first, S last, const T& value, Proj proj = {});
    template<ForwardRange R, class T, class Proj = identity>
    requires IndirectRelation<ranges::equal_to<>, projected<iterator_t<R>, Proj>, const T>'
    constexpr safe_iterator_t<R> find_last(R&& r, const T& value, Proj proj = {});
    template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
             IndirectUnaryPredicate<projected<I, Proj>> Pred>
    constexpr I find_last_if(I first, S last, Pred pred, Proj proj = {});
    template<ForwardRange R, class Proj = identity,
             IndirectUnaryPredicate<projected<iterator_t<R>, Proj>> Pred>
    constexpr safe_iterator_t<R> find_last_if(R&& r, Pred pred, Proj proj = {});
}
constexpr safe_iterator_t<R>
find_last_if(R&& r, Pred pred, Proj proj = {});

template<ForwardIterator I, Sentinel<I> S, class Proj = identity,
IndirectUnaryPredicate<projected<I, Proj>> Pred>
constexpr I find_last_if_not(I first, S last, Pred pred, Proj proj = {});

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

Let $E$ be:

1. $\text{invoke}(\text{proj}, *i) == \text{value}$ for \text{ranges::find}\_\text{last},
2. $\text{invoke}((\text{pred}, \text{invoke}(\text{proj}, *i)) != \text{false}$ for \text{ranges::find}\_\text{last}\_\text{if},
3. $\text{invoke}(\text{pred}, \text{invoke}(\text{proj}, *i)) == \text{false}$ for \text{ranges::find}\_\text{last}\_\text{if}\_\text{not}.

Returns: The last iterator $i$ in the range $[\text{first, last})$ for which $E$ is true. Returns last if no such iterator is found.

Complexity: At most $\text{last - first}$ applications of the corresponding predicate and any projection.
25.6 Acknowledgements

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