Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P3725R3**
Date: 2026-03-24
Reply to: Nicolai Josuttis (nico@josuttis.de)
Co-authors:
Audience: SG9, LEWG, LWG
Issues:
Previous: P3725R2

Filter View Extensions for Safer Use, Rev 3

Several basic use cases of the current filter view are broken or risky or non-intuitive. This paper proposes a workaround to give ordinary programmers of that want to filter data in ranges an easy option to use filter views more intuitive and with less risks.

What is proposed here follows the following SG9 vote;

> SG9 discussed this in Kona on Tuesday:
> https://wiki.edg.com/bin/view/Wg21kona2025/NotesSG9DE251
>
> **We recommend that the proposed changes a) and b) in**
> **P3725R1 "Filter View Extensions for Input Ranges"**
> **are accepted as DRs against C++20, partially resolving DE 251.**
>
> Attendance: 7
> Consensus in favor.
>
> | SF | F | N | A | SA |
> |---|---|---|---|---|
> | 0 | 4 | 2 | 1 | 0 |

And was requested by national bodies for C++26: **AT9-249**, **RU-250**, **DE-251**.

Motivation

Status Quo

Using filter views is both risky and non-intuitive. Let us look at some typical use cases.

UC1) Possible core dumps

For example (see https://www.godbolt.org/z/qrMYb3G4d):

```
std::vector<std::string> coll1{"Amsterdam", "Berlin", "Cologne", "LA"};

// move long strings in reverse order to another container:
auto large = [](const auto& s) { return s.size() > 5; };
auto sub = coll1 | std::views::filter(large)
                 | std::views::reverse
                 | std::views::as_rvalue
                 | std::ranges::to<std::vector>();
```

This program has undefined behavior and results in a **core dump** due to **overwriting memory of other objects**.
Note that the core dump depends on the values and predicate used:

- With “Rome” instead of “Berlin” the example is “only” not working correctly but does not overwrite (still UB).
- With a predicate like “`s.size() < 5`” the example is always well defined and works fine.

UC2) Healing "broken" elements

For example (https://www.godbolt.org/z/nG3v5vMev, example by Patrice Roy):

```
// bring all dead monsters back to live:
auto dead = [] (const auto& m) { return m.isDead(); };
for (auto& m : monsters | std::views::filter(dead)) {
  m.bringBackToLive();  // undefined behavior
}
```

The loop to bring all dead monsters back to live works but is formally undefined behavior. The reason for this UB is that in general code like this can go wrong when after the filter there are other views like reverse (see the previous example).

UC3) No const iterations supported

For example (see https://www.godbolt.org/z/cjYbsn757):

```
void constIterate(const auto& coll);   // forward declaration

std::vector<std::string> coll3{"Amsterdam", "Berlin", "Cologne", "LA"};

auto large = [](const auto& s) { return s.size() > 5; };
constIterate(coll3 | std::views::filter(large));  // compile-time ERROR
```

All three use cases mean that ordinary programmers need good insights of the filter view to understand what is going on and how to avoid serious mistakes.

Proposed Fix

This paper provides a simple workaround for these use cases, which is easy to teach and easy to follow. By putting an **as_input view** (former to_input view) in front of all these broken use cases, they work fine or fail to compile.

UC2 and UC3 just work then:

```
// bring all dead monsters back to live:
auto dead = [] (const auto& m) { return m.isDead(); };
for (auto& m : monsters | std::views::as_input
                        | std::views::filter(dead)) {
  m.bringBackToLive();  // well defined and works now
}

auto large = [](const auto& s) { return s.size() > 5; };
constIterate(coll3 | std::views::as_input
                   | std::views::filter(large));  // OK now
```

,

UC1 (overwriting other memory) no longer compiles:

```
std::vector<std::string> coll1{"Amsterdam", "Berlin", "Cologne", "LA"};

// move long strings in reverse order to another container:
auto large = [](const auto& s) { return s.size() > 5; };
auto sub = coll1 | std::views::as_input
                 | std::views::filter(large)
                 | std::views::reverse
                 | std::views::as_rvalue
                 | std::ranges::to<std::vector>();  // compile-time ERROR now
```

Note that this compile-time error also occurs in more sophisticated use cases previously not broken and working fine. To be able to use filters here, programmers can still use **`std::views::filter()`** without the as_input view.

Note also that as_input is the new name of the to_input view as proposed for C++26 in P3828.

Proposed Changes

The proposed changes are pretty simple:

a) For filter views, add a new **const** member functions **begin()** and **end()** when it operates on an input ranges, which supports a const begin().

b) For filter view iterators, relax the wording of when a modification via the filter view iterator results in undefined behavior.

FAQ

Isn't a const begin() / end() wrong for an input range ?

Note that the proposed change introduces something new. So far, pure input ranges (such as istream_view or generator()) read data with begin() (as with ++) and therefore have no const begin() at all. So this change looks like there is something wrong.

However, from a semantic perspective, this proposal introduces a new special category of range here. Being an input range is only required to disable multi-pass problems, which makes the use of the filter view way safer. In this case, begin() is still a cheap const operation. We assume that this pattern might also be applied to other views in future.

Does this proposal break existing code ?

The proposes change is a pure extension to existing API's. The API is backward compatible. The binary API is backward compatible (note that the now templified types iterator and sentinel are exposition only).

Does the resulting view from `as_input | filter` have `empty()` ?

No, as we can iterate only once, having empty() makes no sense.

Note that this doesn't need a change. The member function **`empty()`** is provided via the base class **`view_interface`** and requires that we have a sized range or a forward range.

Shouldn't we also have a new filter view ?

A new adaptor like safe_filter() or const_filter() might also make sense for different reasons. However, the first step is to have a working workaround because both terminology and constness details might need further discussions.

This proposal is what is necessary to give ordinary programmers a workaround top use a safe and self-explanatory filter so that they can simply compose pipelines with filters and it just works for all basic use-cases. Therefore, this paper only introduces this change.
It will enable to teach composable view to basic programmers without the need to teach caching, universal references, broken predicates and so on.

Proposed Wording

(All against N5032)

Proposed wording for the filter view class

In **24.7.8.2 Class template filter_view [range.filter.view]**

Change the declaration:

```
// 25.7.8.3, class template filter_view::iterator
template<bool Const>
class iterator ;          // exposition only

// 25.7.8.4, class template filter_view::sentinel
template<bool Const>
class sentinel ;          // exposition only

…

constexpr iterator<false> begin();
```

and add:

```
constexpr iterator<true> begin() const
  requires (input_range<const V> && !forward_range<const V> &&
            indirect_unary_predicate<const Pred, iterator_t<const V>>);
```

and change:

```
constexpr auto end() {
  if constexpr (common_range<V>)
    return iterator<false> {*this, ranges::end(base_)};
  else
    return sentinel<false> {*this};
}
```

and add:

```
constexpr sentinel<true> end() const
  requires (input_range<const V> && !forward_range<const V> &&
            indirect_unary_predicate<const Pred, iterator_t<const V>>) {
    return sentinel<true> {*this};
}
```

At the definitions:

change:

constexpr *iterator<false>* begin();

3 *Preconditions*: *pred_*.has_value() is true.

4 *Returns*: {*this, ranges::find_if(*base_*, ref(**pred_*))}.

5 *Remarks*: In order to provide the amortized constant time complexity required by the range concept when filter_view models forward_range, this function caches the result within the filter_view for use on subsequent calls.

and add:

```
constexpr iterator<true> begin() const
  requires (input_range<const V> && !forward_range<const V> &&
            indirect_unary_predicate<const Pred, iterator_t<const V>>);
```

6 *Preconditions*: *pred_*.has_value() is true.

7 *Returns*: {*this, ranges::find_if(*base_*, ref(**pred_*))}.

[*Note*: This function does not cache the result within the filter_view —*end note*]

In **25.7.8.3 [range.filter.iterator]**

25.7.8.3 Class template filter_view::*iterator* [range.filter.iterator]

```
namespace std::ranges {
  template<input_range V, indirect_unary_predicate<iterator_t<V>> Pred>
    requires view<V> && is_object_v<Pred>
  template<bool Const>
  class filter_view<V, Pred>::iterator {
   private:
    using Parent = maybe-const<Const, filter_view>;      // exposition only
    using Base = maybe-const<Const, V>;                  // exposition only
    iterator_t<V Base> current_ = iterator_t<V Base>();  // exposition only
    filter_view Parent * parent_ = nullptr;              // exposition only
   public:
    using iterator_concept = see below;
    using iterator_category = see below;      // not always present
    using value_type = range_value_t<V Base>;
    using difference_type = range_difference_t<V Base>;

    iterator () requires default_initializable<iterator_t<V Base>> = default;
    constexpr iterator (filter_view& parent, iterator_t<V Base> current);
    constexpr iterator (iterator <!Const> i)
      requires Const && convertible_to<iterator_t<V>, iterator_t<Base>>;
    constexpr const iterator_t<V Base>& base() const & noexcept;
    constexpr iterator_t<V Base> base() &&;
    constexpr range_reference_t<V Base> operator*() const;
    constexpr iterator_t<V Base> operator->() const
      requires has-arrow <iterator_t<V Base>> && copyable<iterator_t<V Base>>;
    constexpr iterator & operator++();
    constexpr void operator++(int);
    constexpr iterator operator++(int) requires forward_range<V Base>;
    constexpr iterator & operator--() requires bidirectional_range<V Base>;
    constexpr iterator operator--(int) requires bidirectional_range<V Base>;
    friend constexpr bool operator==(const iterator & x, const iterator & y)
      requires equality_comparable<iterator_t<V Base>>;
    friend constexpr range_rvalue_reference_t<V Base> iter_move(const iterator & i)
      noexcept(noexcept(ranges::iter_move(i.current_)));
    friend constexpr void iter_swap(const iterator & x, const iterator & y)
      noexcept(noexcept(ranges::iter_swap(x.current_, y.current_)))
    requires indirectly_swappable<iterator_t<V Base>>;
  };
}
```

1 [Note 1: Modification of the element a filter_view::*iterator* denotes ~~is permitted, but results~~ **can result** in undefined behavior if **the underlying range is a forward_range and** the resulting value does not satisfy the filter predicate **when the predicate is next evaluated for that element ([concepts.equality]). – end note]**

2 *iterator* ::iterator_concept is defined as follows:

(2.1) — If `Const` is `true`, then `iterator_concept` denotes `input_iterator_tag`.

(~~2.1~~ **2.2**) — **Otherwise, if** ~~If~~ V models bidirectional_range, then iterator_concept denotes bidirectional_iterator_tag.

(~~2.2~~ **2.3**) — Otherwise, if V models forward_range, then iterator_concept denotes forward_iterator_tag.

(~~2.3~~ **2.4**) — Otherwise, iterator_concept denotes input_iterator_tag.

3 The member *typedef-name* iterator_category is defined if and only if ~~V~~ `Base` models forward_range. In that case, *iterator* ::iterator_category is defined as follows:

(3.1) — Let C denote the type iterator_traits<iterator_t<~~V~~ `Base`>>::iterator_category.

(3.2) — If C models derived_from<bidirectional_iterator_tag>, then iterator_category denotes bidirectional_iterator_tag.
(3.3) — Otherwise, if C models derived_from<forward_iterator_tag>, then iterator_category denotes forward_iterator_tag.
(3.4) — Otherwise, iterator_category denotes C.

constexpr *iterator* (filter_view& parent, iterator_t<~~v~~ **Base**> current);

4 *Effects*: Initializes *current_* with std::move(current) and *parent_* with addressof(parent).

constexpr const iterator_t<~~v~~ **Base**>& base() const & noexcept;

5 *Effects*: Equivalent to: return *current_*;

constexpr iterator_t<~~v~~ **Base**> base() &&;

6 *Effects*: Equivalent to: return std::move(*current_*);

constexpr range_reference_t<~~v~~ **Base**> operator*() const;

7 *Effects*: Equivalent to: return **current_*;

constexpr iterator_t<~~v~~ **Base**> operator->() const
requires *has-arrow* <iterator_t<~~v~~ **Base**>> && copyable<iterator_t<~~v~~ **Base**>>;

8 *Effects*: Equivalent to: return *current_*;

constexpr *iterator* & operator++();

9 *Effects*: Equivalent to:

```
current_ = ranges::find_if(std::move(++current_), ranges::end(parent_->base_),
ref(*parent_->pred_));
return *this;
```

constexpr void operator++(int);

10 *Effects*: Equivalent to ++*this.

constexpr *iterator* operator++(int) requires forward_range<~~v~~ **Base**>;

11 *Effects*: Equivalent to:

```
auto tmp = *this;
++*this;
return tmp;
```

constexpr *iterator* & operator--() requires bidirectional_range<~~v~~ **Base**>;

12 *Effects*: Equivalent to:

```
do
--current_;
while (!invoke(*parent_->pred_, *current_));
return *this;
```

constexpr *iterator* operator--(int) requires bidirectional_range<~~v~~ **Base**>;

13 *Effects*: Equivalent to:

```
auto tmp = *this;
--*this;
return tmp;
```

friend constexpr bool operator==(const *iterator* & x, const *iterator* & y)
requires equality_comparable<iterator_t<~~V~~ *Base*>>;

14 *Effects*: Equivalent to: return x.*current_* == y.*current_*;

friend constexpr range_rvalue_reference_t<~~V~~ *Base*> iter_move(const *iterator* & i)
noexcept(noexcept(ranges::iter_move(i.*current_*)));

15 *Effects*: Equivalent to: return ranges::iter_move(i.*current_*);

friend constexpr void iter_swap(const *iterator* & x, const *iterator* & y)
noexcept(noexcept(ranges::iter_swap(x.*current_*, y.*current_*)))
requires indirectly_swappable<iterator_t<~~V~~ *Base*>>;

16 *Effects*: Equivalent to ranges::iter_swap(x.*current_*, y.*current_*).

Before definition of base() add:

```
constexpr iterator (iterator <!Const> i)
  requires Const && convertible_to<iterator_t<V>, iterator_t<Base>>;
```

***Effects*: Initializes `parent_` with `i.parent_` and `current_` with `std::move(i.current_)`.**

In **25.7.8.4 [range.filter.sentinel]**

25.7.8.4 Class template filter_view::*sentinel* [range.filter.sentinel]

```
namespace std::ranges {
  template<input_range V, indirect_unary_predicate<iterator_t<V>> Pred>
    requires view<V> && is_object_v<Pred>
  template<bool Const>
  class filter_view<V, Pred>::sentinel {
   private:
    using Base = maybe-const <Const, V>;              // exposition only
    sentinel_t<~~V~~ Base> end_ = sentinel_t<~~V~~ Base>(); // exposition only
  public:
    sentinel () = default;

    constexpr sentinel (sentinel<!Const> other)
     requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;

    constexpr explicit sentinel (filter_view& parent);

    constexpr sentinel_t<~~V~~ Base> base() const;

    template<bool OtherConst>
     requires sentinel_for<sentinel_t<Base>, iterator_t<maybe-const <OtherConst, V>>>
    friend constexpr bool operator==(const iterator<OtherConst>& x, const sentinel & y);
  };
}
```

```
constexpr sentinel (sentinel <!Const> other)
 requires Const && convertible_to<sentinel_t<V>, sentinel_t<Base>>;
```

***Effects*: Initializes `end_` with `std::move(other.end_)`.**

```
constexpr explicit sentinel (filter_view& parent);
```

1 *Effects*: Initializes *end_* with ranges::end(parent.*base_*).

```
constexpr sentinel_t<~~V~~ Base> base() const;
```

2 *Effects*: Equivalent to: return *end_*;

```
template<bool OtherConst>
 requires sentinel_for<sentinel_t<Base>, iterator_t<maybe-const <OtherConst, V>>>
 friend constexpr bool operator==(const iterator<OtherConst>& x, const sentinel & y);
      Effects: Equivalent to: return x.current_ == y.end_;
```

Feature Test Macro

We propose a feature test macro because parts of the paper might be applied as a defect to earlier versions:

`__cpp_lib_ranges_filter` for the extensions to the filter view (including its iterator).

Acknowledgements

Thanks to a lot of people who helped and gave support again and again to finally get this proposal done. Special thanks go to Tristan Brindle, Peter Dimov, Hewill Kang, Inbal Levi, Jonathan Müller, Barry Revzin, Oliver Rosten, Herb Sutter, Ville Voutilainen, Hui Xie, LEWG and LWG who finally took their time to explain the motivation, discuss options, corrected me, and proposed details.

Rev3:

According to LWG feedback:

- Convert paragraph for undefined modifying filter iterators into a note, so that it is now still well defined to modify elements breaking the predicate as long as these elements are not reevaluated anymore.
- Let begin() const and end const() not return just auto.

Rev2:

According to SG9 and LEWG feedback and time constraints for C++26:

- Proposing only const begin()/end() support for filter views (no new filter adaptor).
- Detailed proposed wording (as discussed)

Rev1:

Small fixes due to SG9 feedback:

- end() const should always return sentinel only
- Constraints fixed
- const_iterator for const begin()

Rev0:

First initial version.

References

P3329R2 Nicolai Josuttis: Filter View Extensions for Safer Use, Rev2

P3828 Nicolai Josuttis: Rename the to_input view to as_input