Project: ISO JTC1/SC22/WG21: Programming Language C++
Doc No: WG21 **P3725R1**
Date: 2025-06-19
Reply to: Nicolai Josuttis (nico@josuttis.de)
Co-authors:
Audience: SG9, LEWG, LWG
Issues:
Previous: http://wg21.link/**p3725r0**

Filter View Extensions for Input Ranges, Rev 1

Several basic use cases of the current filter view are broken or risky. 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.

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” it is “only” not working correct but does not overwrite (still UB).
- With a predicate “`s.size() < 5`” it 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 is that it 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 solutions for these use cases, which is easy to teach and easy to follow. By using **`std::views::input_filter()`** to create the filter view all broken basic use cases work fine and/or safe.

UC2 and UC3 just work:

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

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

,

UC2 (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::input_filter(large)
                 | std::views::reverse
                 | std::views::as_rvalue
                 | std::ranges::to<std::vector>();  // compile-time ERROR
```

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()`**.

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

> Modification of the element a filter_view::*iterator* denotes is permitted, but results in undefined behavior if the resulting value does not satisfy the filter predicate.

to allow all modifications on input iterators:

Modification of the element a filter_view::*iterator* denotes is permitted, but results in undefined behavior if **the underlying range is a forward_range and** the resulting value does not satisfy the filter predicate.

c) Provide a new range adaptor object **std::views::input_filter,** which forces the underlying range internally to be used as input_range using the **to_input** view.

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 (see a note on this in the wording).

Does the input_filter view 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 const_filter view ?

An adaptor const_filter() might also make sense for different reasons. However, const behavior is a different topic (and slightly more complicated due to owning views and deep const issues). This proposal is what is necessary to give ordinary programmers 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 N5008)

In **25.7.8.1 Overview [range.filter.overview]**

Add a new paragraph:

> The name **views::input_filter** denotes a range adaptor object (25.7.2). Given subexpressions E and P, the expression `views::input_filter(E, P)` is expression-equivalent to `filter_view(to_input_view(E), P)`.

in **25.7.8.2 Class template filter_view [range.filter.view]**

in the class structure replace:

```
constexpr iterator begin();

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

by

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

constexpr auto end() {
  if constexpr (common_range<V>)
    return iterator {*this, ranges::end(base_)};
  else
    return sentinel {*this};
}
constexpr auto end() const requires (input_range<const V> && !forward_range<const V>) {
  return sentinel {*this};
}
```

In **24.7.4.2 Class template filter_view [range.filter.view]**

Inroduce an exposition only const_iterator type:

```
// 25.7.8.3, class filter_view::iterator
class iterator ;          // exposition only
class const_iterator ;    // exposition only
```

Before the definition of

constexpr *iterator* begin();

add a second member function:

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

using the same effects clause.

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

Replace

> Modification of the element a filter_view::*iterator* denotes is permitted, but results in undefined behavior if the resulting value does not satisfy the filter predicate.

by:

> Modification of the element a filter_view::*iterator* denotes is permitted, but results in undefined behavior if **the underlying range is a forward_range and** the resulting value does not satisfy the filter predicate.

Add a new section **25.7.8.3 Class filter_view::const_*iterator* [range.filter.const.iterator]**

Introduce a new exposition-only type const_iterator according to iterator (or extent iterator to have a const and non-const version).

Note: This is a new exposition only type. Because it is exposition only, implementers can implement it as they .like. However, the usual trick to have a generic iterator type with a Boolean NTTP to signal constness might not be appropriate here because converting the existing exposition-only class iterator into a class template would be a binary breaking change.

Feature Test Macro

Add a new ranges or filter_view feature test macro.

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 Barry Revzin, Jonathan Müller, Ville Voutilainen, Peter Dimov, Hui Xie, Herb Sutter, Tristan Brindle, which finally took their time to explain the motivation, discuss options, corrected me, and proposed details of the current situation.

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