Project:	ISO JTC1/SC22/WG21: Programming Language C++
Doc No:	WG21 **P3329R0**
Date:	2024-11-13
Reply to:	Nicolai Josuttis (nico@josuttis.de)
Co-authors:	
Audience:	SG9, LEWG, LWG
Issues:	
Previous:	

Healing the Filter View, Rev 0

For filter views, several basic and common use cases cause unexpected compile-time errors and fatal runtime errors with undefined behavior. Because almost all of these effects are not obvious, ordinary programmers get confused and frustrated by error messages and behavior they do not understand or, even worse, are not aware that their code is highly fragile or even broken.

As a consequence, the filter view and the view library as a whole is considered to be unusable and dangerous for many companies and projects and more and more banned from being used.

Most of the broken use cases are a consequence of the fact that filter views cache begin(). You would expect that internal caching causes no harm, because it should be transparent for an API. However, this is not the case for filter views. Instead, the caching of filter views has dramatic consequences for its behavior. You might assume, then, that there is a compelling reason for non-transparent caching. However, it turns out that the only reason is to avoid possible bad performance of some special and rare use cases.

This paper suggests to “heal” the filter view so that it works as expected, is simple to use, even by ordinary programmers in safety critical environments in natural and obvious ways.

The Current Problems of the Filter View in a Nutshell

The current design of the filter view leads to the following problems, which are discussed below in detail:

Problem	Description	Category	Effect
Read iterations do not work with const	Cannot use a filter view when it is const	Compile-time errors	Highly confused and frustrated programmers. Compromises the basic paradigm of const-ness.
empty() is not stateless	Calling empty() may change/compromise behavior	Runtime errors	Calling `if (v.empty())` may change program behavior.
Read iterations are not stateless	Printing elements may change/compromise behavior	Runtime errors	Adding a print statement may change program behavior.
Concurrent reads are not thread safe	Concurrent Read Access may be Broken	Runtime errors	Undefined behavior when two threads read using a filter view.
Modifications between iterations are not safe	Broken Use Cases when Modifying Elements between Using the Filter View	Runtime errors	Wrong/invalid elements are processed.
Modifications via filter iterators are not safe, when breaking predicates	Broken Use Cases when Writing with the Filter View	Runtime errors	A key use case of filtering results in undefined behavior. Wrong/invalid elements may be processed or even abnormal program termination.

Filters cache different things	Filters on vectors and lists are not consistent (and break different use cases)	Runtime errors	Switching between vectors and lists and other subtle changes in the underlying ranges and pipelines change/break behavior.
A copy might have a different state	Pass-by-value might change state	Runtime errors	Switching between call-by-value and call-by-reference might change behavior.

What this Paper proposes

The current filter view design, which causes all these drawbacks and traps, is not necessary. Looking at the motivation of the current design and possible other options, it turns out that it is surprisingly easy to heal the filter view. With a different design, all the problems described here would no longer occur.

Of course, there is no fix without any drawbacks. However, the proposed alternative design approach heals the damage without introducing significant new damage: We propose that the use cases with bad performance no longer compile instead of breaking useful basic use cases. That means: By changing the current design we can ensure that all problems described here are gone without creating new runtime problems. Most code will suddenly simply work and no longer cause surprising or undefined behavior. This fix is even possible without breaking binary compatibility.

Note that this does not mean that there can no longer be and problem when using the filter view. By nature, filter views can create trouble for some non-trivial use cases. The goal is to change the design to ensure that basic and simple use cases just work fine and behave as expected.

Tony Table

The following table lists the most important basic use cases broken due to the current design of the filter view and those that benefit from the current design:

	Read access with `const`	**Pure read on stable range**	**Range modified between iterations**	**Modify on iteration Predicate kept**	**Modify on iteration Predicate broken**
one iteration from begin to end	**Error** (compile-time)	✓	✓	✓	often works but undefined behavior
`empty()` and **one iteration**	**Error** (compile-time)	✓ (fast)	**broken** (runtime)	✓ (fast)	often works but undefined behavior
multiple iterations from begin to end	**Error** (compile-time)	✓ (fast)	**broken** (runtime)	✓ (fast)	**broken** (runtime)
going **back and forth**	**Error** (compile-time)	✓	**broken** (runtime)	✓	**broken** (runtime)
concurrent two iterations or **`empty()`** + **iteration**	**Error** (compile-time)	**broken** (runtime)	**broken** (runtime)	**broken** (runtime)	**broken** (runtime)
one reverse iteration	**Error** (compile-time)	✓ (fast)	**broken** (runtime)	✓ (fast)	**broken** (runtime)
two reverse iterations	**Error** (compile-time)	✓ (fast)	**broken** (runtime)	✓ (fast)	**broken** (runtime)

Note:

- The cells marked with "(fast") benefit from the current design.
- All **blue broken** entries are **unnecessarily broken** due to the current design of the filter view.
- Only all **red broken** entries are naturally broken and cannot be avoided with any useful design.

The suggested fix is to disable caching and make filter view iterators forward iterators only.

With this fix, we get the following behavior instead:

	Read access with `const`	**Pure read on stable range**	**Range modified between iterations**	**Modify on iteration Predicate kept**	**Modify on iteration Predicate broken**
one iteration from begin to end	✓	✓	✓	✓	✓
`empty()` and **one iteration**	✓	✓ (slower)	✓	✓ (slower)	✓
multiple iterations from begin to end	✓	✓ (slower)	✓	✓ (slower)	✓
going **back and forth**	✓	✓	✓	✓	**broken** (runtime)
concurrent two iterations or **`empty()` + iteration**	✓	✓	✓	**broken** (runtime)	**broken** (runtime)
one reverse iteration	**Error** (compile-time)				
two reverse iterations	**Error** (compile-time)				

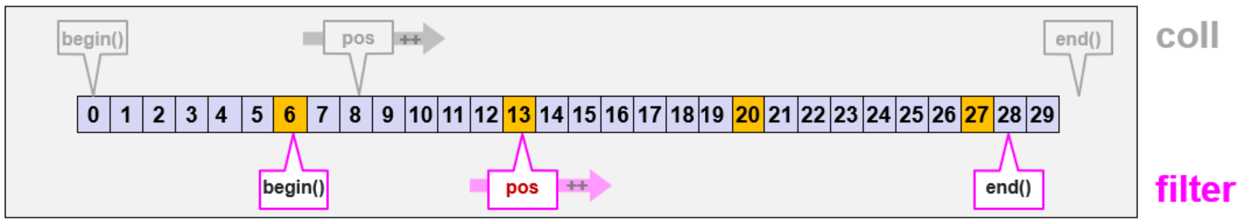
So a few use cases get slightly worse performance. However, note that they still have the same complexity and that there are simple workaround to get back the current performance with caching in use cases that are affected.

The key message is that several basic use cases that were broken suddenly compile and simply work.

The Current Design of the Filter View

The current design of filter views is driven by the goal to have good performance (at least, never have really bad (quadratic) performance).

The problem of possible bad performance lies in the nature of the filter view. Common basic member functions, that are cheap for containers, might be expensive for filter views. And in some situations that can have dramatic consequences.
Consider we have the basic use case of applying a filter on a simple (maybe sequential collection) of elements:



To understand how expensive an iteration over all elements is, we have to take the following costs into account:

The cost of begin()

begin() has to find the first element that matches the filter predicate. However, this first element can be at the end or even not exist at all. begin() therefore has to iterate from the beginning over all elements looking at their values until a first match is found or the end is reached.

This means that **begin() may have liner complexity in the worst case**. With use cases where usually no matching element is found, doubling the number of elements means that begin() has to iterate over a doubled number of elements.

Whether this really is a problem depends on details. If usually many elements fit (such as we only look for elements with odd values which pretty likely always exist sooner or later at the beginning), begin() always roughly takes the same amount of time independent from the number of elements.

The cost of ++

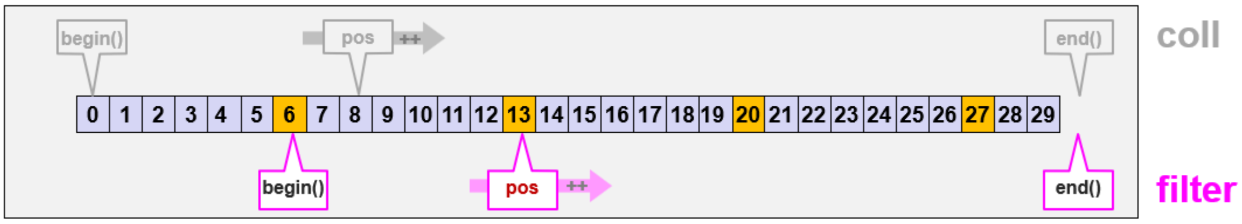
Once we found the beginning, an iteration iterates to the next element calling ++ for the iterator. This has the same problem and therefore the same complexity like calling begin(): Internally, we have to call ++ multiple times for the underlying range to find the next matching element. The less likely it is that elements fit, the more expensive ++ becomes.
Again we have **a linear complexity in the worst case**.

Note that the position of matching elements cannot be computed without looking at each and every value in front of it. For this reason, filter view iterators cannot support random access. They can only be bidirectional iterators or worse.

The cost of end()

We iterate over ranges until an iterator reached the end(), which is the position behind the last matching element. Now you might assume that end() is also expensive because we have to find the position right behind the last matching element, which means to iterate either from the beginning over all elements or backward iterate from the end (if the underlying range supports that) to compute the end().

However, there is a trick, the filter view uses: It just takes the underlying end() as its end:



This means that **end() is super fast** (has constant complexity) **provided** end() of the underlying range is super fast.

However, this trick has the consequence that calling ++ for an iterator that refers to the last matching element might again result into multiple calls of ++ on the underlying range. But that is anyway the case for ++.

The cost of an iteration on a filter view

Now let's look how expensive full iterations over all elements of filter views are. We have to take into account both the costs of calling begin(), end() and ++ as well as how often we call these operations.

In practice, there are two different basic ways to iterate over the elements of a range:

- We can use a range-based `for` loop:

  ```
  v = coll | std::views::filter(every7th);
  for (const auto& elem : v) {
    process(elem);
  }
  ```

  This code is effectively iterating over all element and iterator using a loop such as the following:

  ```
  v = coll | std::views::filter(every7th);
  auto end = v.end();
  for (auto pos = v.begin(); pos != end; ++pos) {
    process(elem);
  }
  ```

  In this scenario we call:

  - begin() once
  - end() once
  - ++ once for each element of the underlying range

  Therefore, **the cost of iterating over all elements of the filter is as expensive as iterating over all elements of the underlying range** (combined with a check for each element whether it fits).

- We can use a manual loop using iterators
  Pretty often, code that manually iterates does not call end() only once. Instead, it looks as follows:

  ```
  v = coll | std::views::filter(every7th);
  for (auto pos = v.begin(); pos != v.end(); ++pos) {
    process(elem);
  }
  ```

In this scenario we call:

- begin() once
- end() multiple times
- ++ once for each element of the underlying range

Therefore, **the cost of iterating over all elements of the filter is as expensive as iterating over all elements of the underlying range plus the cost of calling end() on the underlying range multiple times.**

If end() of the underlying range is cheap, there is no problem. And if each end() delegates to a cheap underlying end() there can't be a problem.
However, there are use cases where calling end() is not cheap.
And when iterating over the elements of a filter view, we can get into trouble if use cases call begin() multiple times.

Calling begin() Multiple Times

An simple common use case where we call begin() more than once demonstrates the following code:

```
v = coll | std::views::filter(every7th);
if (v.empty()) return;    // skip processing if the range is empty
…
for (const auto& elem : v) {
  process(elem);
}
```

Before we later process the elements in the range-base **for** loop, we check whether the range we process is empty. For filter views, calling empty() is more or less the same as calling begin(): empty() has to find the begin() and see whether it is the end(). Therefore, in this scenario we call:

- begin() twice
- end() once
- ++ once for each element of the underlying range

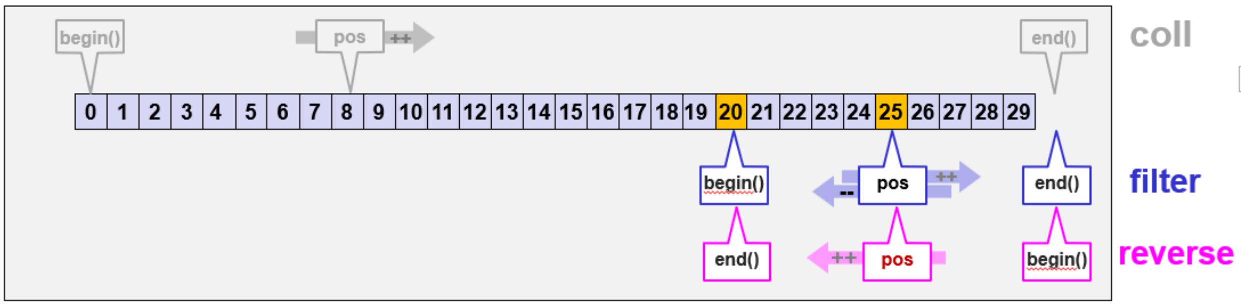
In the worst case scenario (the last element is the first matching element) this can double the time it takes to run this code, because we iterate twice instead of once to the last element. But note that we still have linear complexity. So this use case is worth looking at but doesn't kill scalability.

The cost of an reverse iteration on a filter view

The use case that really causes trouble is a reverse iteration over a filter view. Consider we have a use case like this:

```
v = coll | std::views::filter(someElemsAtTheEnd) | std::views::reverse;
```

where *someElemsAtTheEnd* for example selects two elements close to the end of the underlying range:



Note that a reverse iteration internally maps `begin()` to `end()`, `end()` to `begin()`, and `++` to `--` (with a small adjustment, because begin() refers directly to the first element but end() refers behind it).
Now, when performing a reverse iteration with an approach that calls end() on the reverse view multiple times:

```
        for (auto pos = v.begin(); pos != v.end(); ++pos) {
            process(elem);
         }
```

we effectively call begin() of the underlying filter view multiple times. For the filter view, we get:

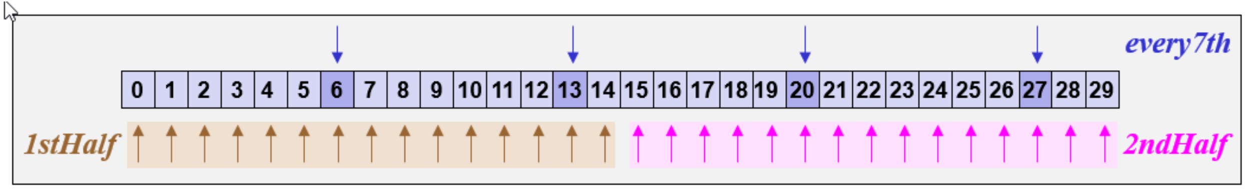
- begin() multiple times
- end() once
- -- once for each element of the underlying range

In this case, the cost of a reverse iteration over the elements of the filter view can grow quadratically provided the first matching element comes late. Calling end() multiple times means we call begin() multiple times, and each call of begin() calls ++ multiple times to find the first element.

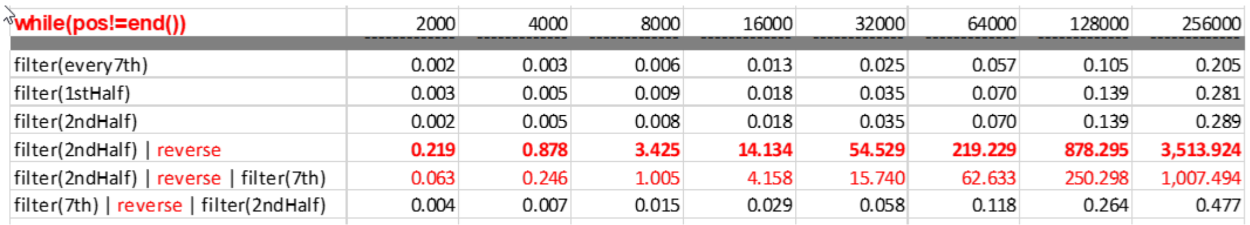
Note that this is not a problem when using the range-based `for` loop, because it calls begin() only once. However, having a pipeline with additional views after the reverse view can have the same effect.

The real costs of a naïve implemented filter view

Consider we have a couple naively implemented filters we can apply to a range:



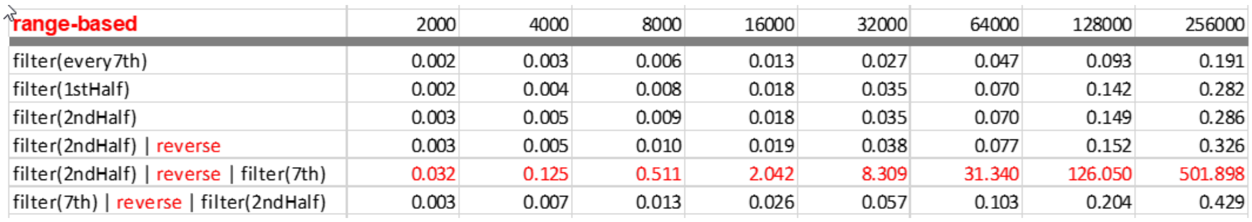
Using the manual loop approach calling end() multiple times, we can measure the following:



while(pos!=end())	2000	4000	8000	16000	32000	64000	128000	256000
filter(every7th)	0.002	0.003	0.006	0.013	0.025	0.057	0.105	0.205
filter(1stHalf)	0.003	0.005	0.009	0.018	0.035	0.070	0.139	0.281
filter(2ndHalf)	0.002	0.005	0.008	0.018	0.035	0.070	0.139	0.289
filter(2ndHalf) | reverse	**0.219**	**0.878**	**3.425**	**14.134**	**54.529**	**219.229**	**878.295**	**3,513.924**
filter(2ndHalf) | reverse | filter(7th)	0.063	0.246	1.005	4.158	15.740	62.633	250.298	1,007.494
filter(7th) | reverse | filter(2ndHalf)	0.004	0.007	0.015	0.029	0.058	0.118	0.264	0.477

In this table we double the number of elements in each column. You can clearly see how the running time grows by a factor of 4 in the scenarios where we have a reverse view involved. In total, we reach factors of 100 for 2,000 elements and 10,000 for 250,000 elements, which simply means that a reverse iteration on a filter view does not scale with multiple calls of end().

Using the range-based `for` loop approach (calling end() only once), we can measure the following:



range-based	2000	4000	8000	16000	32000	64000	128000	256000
filter(every7th)	0.002	0.003	0.006	0.013	0.027	0.047	0.093	0.191
filter(1stHalf)	0.002	0.004	0.008	0.018	0.035	0.070	0.142	0.282
filter(2ndHalf)	0.003	0.005	0.009	0.018	0.035	0.070	0.149	0.286
filter(2ndHalf) | reverse	0.003	0.005	0.010	0.019	0.038	0.077	0.152	0.326
filter(2ndHalf) | reverse | filter(7th)	0.032	0.125	0.511	2.042	8.309	31.340	126.050	501.898
filter(7th) | reverse | filter(2ndHalf)	0.003	0.007	0.013	0.026	0.057	0.103	0.204	0.429

Again, we double the number of elements in each column. You can clearly see that when using the range-based `for` loop, a reverse iteration over a filter is not a problem at all. However, the running time grows by a factor of 4 in the scenario, where we have a filter behind the reverse iteration over a filter. In this case, we reach factors of 10 for 2,000 elements and 1.000 for 250.000 elements, which is not as bad as before, but again means that a filter on a reverse iteration on a filter view does not scale because it grows quadratically.

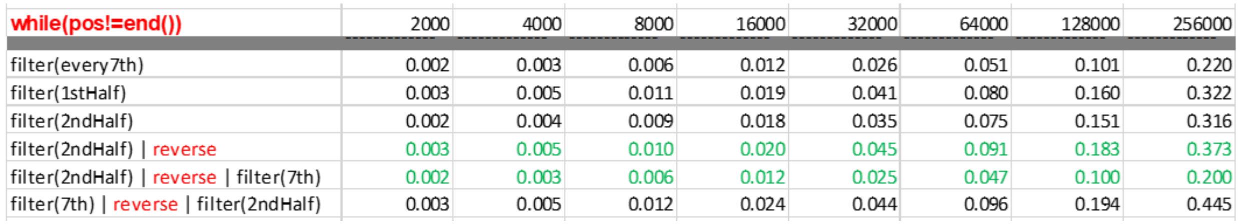
Filter views cache begin()

As a consequence of the described problems the design of the standardized C++ filter views requires that filter views cache begin().

The effect is that only the first call of begin() (or empty(), respectively) is expensive. Calling begin() (or empty()) ***multiple*** times is not a performance problem then.

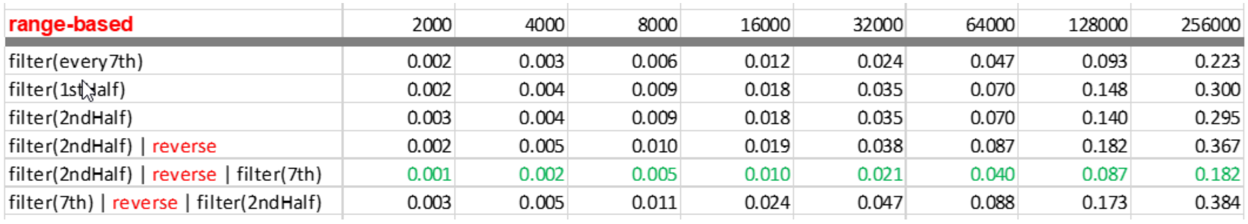
In fact, for the use cases measures above, we get the following performance:

When calling end() multiple times:



while(pos!=end())	2000	4000	8000	16000	32000	64000	128000	256000
filter(every7th)	0.002	0.003	0.006	0.012	0.026	0.051	0.101	0.220
filter(1stHalf)	0.003	0.005	0.011	0.019	0.041	0.080	0.160	0.322
filter(2ndHalf)	0.002	0.004	0.009	0.018	0.035	0.075	0.151	0.316
filter(2ndHalf) | reverse	0.003	0.005	0.010	0.020	0.045	0.091	0.183	0.373
filter(2ndHalf) | reverse | filter(7th)	0.002	0.003	0.006	0.012	0.025	0.047	0.100	0.200
filter(7th) | reverse | filter(2ndHalf)	0.003	0.005	0.012	0.024	0.044	0.096	0.194	0.445

When using the range-based **`for`** loop and/or calling end() once:



range-based	2000	4000	8000	16000	32000	64000	128000	256000
filter(every7th)	0.002	0.003	0.006	0.012	0.024	0.047	0.093	0.223
filter(1stHalf)	0.002	0.004	0.009	0.018	0.035	0.070	0.148	0.300
filter(2ndHalf)	0.003	0.004	0.009	0.018	0.035	0.070	0.140	0.295
filter(2ndHalf) | reverse	0.002	0.005	0.010	0.019	0.038	0.087	0.182	0.367
filter(2ndHalf) | reverse | filter(7th)	0.001	0.002	0.005	0.010	0.021	0.040	0.087	0.182
filter(7th) | reverse | filter(2ndHalf)	0.003	0.005	0.011	0.024	0.047	0.088	0.173	0.384

The bad performance of reverse iterations on filter views is gone.

However, the requirement to cache begin() comes with a price, because the required caching is not transparent. As a consequence, the current design with caching has an immediate dramatic effect on several basic use cases for filtering. The most important fatal consequence were already listed above as the The Current Problems of the Filter View in a Nutshell. The following sections describe these consequences in detail.

Broken Use Cases when Reading with the Filter View

Let us go through various typical basic use cases that are unexpectedly broken by the filter view due to their current design.

In this section, we start with the fatal consequences when using the filter view only to read elements.

Cannot use a filter view when it is `const`

Category:	**Compile-time errors**
Effect:	Highly confused and frustrated programmers. Compromises the basic paradigm of **`const`**-ness.

Consider the following code(see https://www.godbolt.org/z/PPMjYTsan for the full example):

We first provide a generic print function just performing a read iteration over all elements of a passed range.

```
template<typename CollT>
void print(const CollT& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << '\n';     // do something read-only with each element
  }
}
```

Now let us use this print() function for some basic use cases with views:

```
print(coll | std::views::take(3));            // OK
print(coll | std::views::drop(3));            // OK
print(coll | std::views::transform(square)); // OK

print(coll | std::views::filter(isEven));     // compile-time ERROR

for (const auto& elem : coll | std::views::filter(isEven)) {   // OK !!!
  std::cout << elem << '\n';     // do something read-only with each element
}
```

While most of the C++ standard views applied to collections can be passed to print() without any problem, applying the filter view suddenly does not work. The error message is not helpful.

Even more confusing is the fact that in general you can apply the filter to the underlying collection coll (as the raw call of the range-based **`for`** loop at the end demonstrates). It is just the combination of using a filter view ***and*** the call of print() what is broken.

This behavior is in no way intuitive. For programmers, it is obvious that this code should compile and work. The fact that this is not the case this is a very early experience that brings programmers trying out views into deep trouble. Their usual conclusion is that there is something wrong with themselves or C++. They give up, try ugly workarounds or hate C++ even more. I had many trainings and tutorials where a reaction was “ah, now I understand why this didn’t work”.

The problem is that we pass the filter view to a generic function that makes the view **`const`**. But even for pure read access, filter views do not allow to make the view const, because begin() modifies the view when caching the found position of the first matching element.

This is such a fatal and unexpected break of the principle of const-ness that nobody understands what is going on. When I motivate and explain the design, programmers cannot believe that the standard committee comes up with such a huge break of a basic feature of C++. The standard committee looks like simply compromising and giving up **`const`**, one of key principles of C++.

This would only be acceptable for a very good reason and with obvious warning signs in the resulting code. But both is not the case. As we saw, the reason for this huge breakage is to get better performance in some very rare use cases (see The Current Design of the Filter View). Instead of simply stopping these use cases to compile, we prefer to compromise the basic principle of **`const`**.

Once, programmers understand this trap, they have to use workarounds if they do not want to give up using filter views. The possible (proposed) workarounds are as follows:

- Either the caller has to do something special. For example:

  ```
  auto collEven = coll | std::views::filter(isEven);   // init the filter view
  print(std::ranges::subrange{collEven});               // and pass it as subrange
  ```

  Note that this code needs multiple statements. The following statements do not compile:

  ```
  print(std::ranges::subrange(coll | std::views::filter(isEven)));   // ERROR
  ```

  And the following statements with a created temporary view **`collEven`** also don't compile:

  ```
  auto collEven = coll | std::views::filter(isEven);   // init the filter view
  print(collEven);                                                   // ERROR
  print(std::views::all(collEven));                                  // ERROR
  ```

- Or the generic function for a range has to be rewritten or overloaded. One possible change is to use universal/forwarding references :

  ```
  template<typename CollT>
  void print(CollT&& coll)           // note: universal reference !!!
  {
    for (const auto& elem : coll) {
      std::cout << elem << '\n';     // do something read-only with each element value
    }
  }
  ```

The consequence of these changes and workarounds are fatal:
First, we can no longer teach the effect of **`const`** without saying "when using views the principles of **`const`** are broken". This raises basic questions of how to teach writing generic code. Should we give up teaching **`const T&`** or **`const auto&`** ? The promise so far was that universal references were just for perfect forwarding (that's why the standard committee named them forwarding references; it turns out, now how wrong and bad this change of the established name "universal reference" was).

Even worse, in their frustration, programmers start to use even more ugly workarounds. Here is a remarkable one:

```
template<typename CollT>
CollT& makeIterable(const CollT& v)
{
  return *const_cast<std::remove_cvref_t<CollT*>>(&v);
}

template<typename CollT>
void print(const CollT& coll)
{
  for (const auto& elem : makeIterable(coll)) {
    std::cout << elem << '\n';  // do something with each element value
  }
}
```

Yes, bad design leads to bad code. Let me clearly say that this code is the fault of the C++ standard committee not the fault of 4 million programmers! We cannot expect that 4 million programmers have to deal with bad design in a useful way.

Concurrent Read Access may be Broken

Category:	**Runtime errors (undefined behavior)**
Effect:	Undefined behavior when two threads read concurrently

Programmers are used to know that for ranges concurrent reads are safe. We have designed all containers that way and still guarantee that for containers in general. For 20 years, several generic code for ranges was written with that guarantee in mind.

However, for views, basic read operations such as begin(), empty(), front() of the filter view are no longer thread safe. The effect is that concurrent read iterations result in undefined behavior.

Consider the following example (full code at https://www.godbolt.org/z/bGE6rvEc5):

```
std::vector<int> coll{1, 2, 3, 4, 5};
…
auto not0 = [] (const auto& val) { return val != 0; };
auto rg = coll | std::views::filter(not0);

// accumulate values in separate thread:
std::jthread t1{[&]{
  std::cout << std::accumulate(rg.begin(), rg.end(),
                               0L) << '\n';
}};
```

So far, this code is fine.

However, when adding the following check after the thread was started:

```
if (!rg.empty()) {    // OOPS: causes undefined behavior
  …
}
```

we suddenly get undefined behavior, because we have concurrent calls of begin()/empty() which is data race for the filter view.

Of course, problems like this can occur way more indirectly. This especially means that a call such as empty() or a read iteration are suddenly safety-critical for generic code that might be used in multiple threads.

That means: **The current design of the filter view makes concurrent reads and calls of empty() bad for generic code.**

As an alternative consequence, using filter views in code that could (indirectly) run in multiple threads is highly dangerous and should be avoided.

Again, we start to see silly workarounds for this problem. This is one of the worst I had to discuss with programmers:

```
void process(auto&& rg) {
  (void)rg.empty();     // force caching (DO NOT REMOVE)

  // separate thread to read access the elements:
  std::jthread t1{[&] {
    for (const auto& elem : rg) {   // OK: concurrent begin() works fine now
      ...
    }
  }};

  if (!rg.empty()) {   // OK: concurrent begin() works fine now
    ...
  }
}
```

Yes, the call of empty() forces the initialization of the cache, which is therefore a very subtle way to avoid that the following code results into undefined behavior.

Again: Bad design forces bad code. This code is the fault of the filter view designers not the fault of any programmer using it.

Note that this “silly workaround with calling empty()” demonstrates another serious trap caused by the bad design of the filter view. For filter views, empty() is not stateless, which is a recipe for huge confusion and runtime errors (see Calling empty() may change/compromise the behavior of a program).

The filter view does not cache `end()`

There is another a bit of surprising consequence of the current design. However, note that this consequence is not really a problem we propose to fix. However, it often comes up when teaching the filter view because caching begin() to heal the issue of a late first matching element raises the expectation that the filter view also makes an early last element cheap. But this is not the case.

Category:	Performance issues
Effect:	Use cases with early last match do not have the expected performance.

While the filter view does cache begin() to make a second begin() cheap, surprisingly, the filter view does not cache end().

A program such as this demonstrates the effect (for the full program including some numbers, see https://www.godbolt.org/z/vWqoe8Gqq):

```
std::vector<int> rg{0, 1, 2, 3, 4, 5, 6,… 99};

// 2 iterations where begin() is the last element:
auto vLast = vec | std::views::filter(equalTo99);
for (const auto& elem : vLast) {      // slow (iterates over all elements)
}
for (const auto& elem : vLast) {      // fast (iteration starts from last element)
}

// 2 iterations where begin() is the first element and not other elements match:
auto vFirst = rg | std::views::filter(equalTo0);
for (const auto& elem : vFirst) {     // slow (iterates over all elements)
}
for (const auto& elem : vFirst) {     // slow (iterates again over all elements)
}
```

We explained in The cost of end() why the filter view does not cache end() as the “position behind the last element”. Instead, it just uses the end of the underlying range as its end(). For an early last match, this has the effect that iterating over all elements multiple times does not benefit from caching at all.

This consequence is surprising for programmers, because while begin() is typically only called once, end() may be called multiple times, when iterations are implemented as follows:

```
for (auto pos = rg.begin(), pos != rg.end(); ++pos) {
  …
}
```

So, caching the end() looks like being at least as important as caching begin(), but is not done at all. A use case where typically only early elements fit does therefore not benefit from caching. Multiple iterations all have linear complexity.

Broken Use Cases when Writing with the Filter View

As second part of the consequences of the current filter view design let us look at the use cases where filters are used to modify some of the elements in a filter view.

Using views for modifications is in principle allowed and supported (not only for filter views). However, the current design creates serious traps for the programmers when filter views are used.

Cannot use a filter to „heal" broken elements

Category:	**Runtime errors (undefined behavior)**
Effect:	A key use case of filtering results in undefined behavior. Wrong/invalid elements may be processed or even abnormal program termination.

One key use case to use a filter when iterating over elements of a collection is to "heal" broken elements. That is, for each element that does not have a valid state, we modify it so that it gets back a valid state.

Consider the following example (thanks to Patrice Roy for this example):

```
// as a shaman:
for (auto& m : monsters | std::views::filter(isDead)) {
  m.resurrect();    // undefined behavior: because no longer dead
  m.burn();         // OK (provided it is still dead)
}
```

This code is undefined behavior. You cannot filter for dead monsters to bring them back to live. This code compiles and might even work, but it is not allowed to do so.

Instead of just making this example work, the standard intentionally places the burden on the programmer making this a runtime error. According to [range.filter.iterator] **the C++ standard states:**

> 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.

Why? What are the problems that might break?

Well, we can get into serious trouble if we iterate more than once from begin to end.

Consider the following code (for the full program, see https://www.godbolt.org/z/W7TKGjsYE):

```
std::vector<int> coll{1, 2, 3, 4};

// a simple filter view that ignores odd elements of coll:
auto isEven = [] (auto&& i) { return i % 2 == 0; };
auto collEven = coll | std::views::filter(isEven);

print(coll);        // 1 2 3 4

// increment even elements (works but UB):
for (int& i : collEven) {
  i += 1;           // formally undefined behavior, but works
}
print(coll);        // OK: 1 3 3 5

// increment even elements (broken):
for (int& i : collEven) {
  i += 1;           // undefined behavior and broken
}
```

```
print(coll);        // OOPS: 1 4 3 5  (yes, initial first odd element was incremented again)
```

Because filter views cache begin(), a second use of the same filter yields an element a begin() where the predicate no longer applies to. Yes, the caching implemented for filter views is not transparent. It compromises several intuitive use cases.

The usual recommendation here is to apply views and pipelines ad-hoc to the underlying range claiming this would always work:

```
// increment even elements (broken):
auto isEven = [] (auto&& i) { return i % 2 == 0; };
for (int& i : coll | std::views::filter(isEven)) {
  i += 1;              // undefined behavior and broken
}
```

However, the recommendation to heal by applying filter views ad-hoc has its limits. Consider the following simple program (see https://www.godbolt.org/z/PWhYhedYr for the complete example):

```
std::vector coll{1, 2, 3, 4, 5, 6, 7, 8};

// increment even elements (broken):
auto isEven = [] (auto&& i) { return i % 2 == 0; };
for (int& i : coll | std::views::filter(isEven) | std::views::reverse) {
  i += 1;              // undefined behavior and broken
}
print(coll);        // OOPS: 1 2 3 5 5 6 7 9
```

Initializing `coll` here with one value less even causes an abnormal program termination due to resulting in an endless loop overwriting memory of other objects (see https://www.godbolt.org/z/nG5nPMje4):

```
std::vector coll{1, 2, 3, 4, 5, 6, 7};

// increment even elements (endless loop resulting in a CORE DUMP):
auto isEven = [] (auto&& i) { return i % 2 == 0; };
for (int& i : coll | std::views::filter(isEven) | std::views::reverse) {
  i += 1;              // UNDEFINED BEHAVIOR => FATAL RUNTIME ERROR
}
```

The fact that a filter cannot be used to modify elements under some condition is bad enough. **A key use case of filters is broken.** But a design that creates such fragile code where it sometimes works, and sometimes not causing severe runtime errors is in no way acceptable. Instead of helping programmers to detect or avoid such a problem by disabling that this code compiles, we expect that 4 million programmers just have to know and deal with this trap.
There is no excuse for such a bad design for a library simple to use and provided for the mass.

All programmers, projects, and companies, where I taught this behavior, feel like the C++ standard committee is shooting ordinary programmers in the foot. This design is in no way acceptable. Especially, as we start to claim that we want to make C++ safer to use.

What is really painful is that there are other design options that do not pay such a high price. Without caching, this problem simply would not exist.

Just to be clear: As Barry Revzin points out in [https://brevzin.github.io/c++/2023/04/25/mutating-filter/], in general, multi-pass processing of elements while mutating elements can always result into severe issues and undefined behavior. However, ***trivial*** modifying use cases such as modifying elements during a single iteration (i.e. when used like input iterators) or between iterations should just work fine.

Broken Use Cases when Modifying Elements between Using the Filter View

The filter view *refers* to an underlying range. Naturally, this can cause serious trouble such as using he view when the underlying range is no longer valid. However, the current design with non-transparent caching also creates highly unexpected runtime errors even when we change data ***before*** or ***between*** using a filter view. Let us look into a couple of examples for this.

Calling `empty()` may change/compromise behavior

Category:	**Runtime errors (undefined behavior)**
Effect:	Calling `if (v.empty())` may change program behavior.

One very surprising consequence of the current filter view design is that a just a call of `empty()` can change the **behavior** of a program. **This change can result into a state so that the filter view does no longer work correctly.**

Here is a simple example (see https://www.godbolt.org/z/Yv8fcM9d8 for the full example):

```
std::list coll{2, 3, 4, 5};

// view for all odd elements:
auto odd = [] (auto v) { return v % 2 != 0; };
auto vOdd = coll | std::views::filter(odd);
…

// insert a new element at the front:
coll.push_front(1);            // coll is now: 1 2 3 4 5

print(vOdd);                   // OK:  1 3 5
```

If we add a call of `empty()` for the view, the output of the program changes:

```
std::list coll{2, 3, 4, 5};

// view for all odd elements:
auto even = [] (auto v) { return v%2 == 0; };
auto vEven = coll | std::views::filter(even);
if (vEven.empty()) {             // OOPS: may change program state
    return;
}
…

// insert a new element at the front:
coll.push_front(1);            // coll is now: 1 2 3 4 5

print(vOdd);                   // OOPS:  3 5
```

The fact that empty() is not stateless, is in no way intuitive or explainable. It is a real trap, caused by the current filter view design. Again, the design of the filter view compromises well established intuitive behavior and design established for decades.

Note that we even standardized with C++17, that empty() should be `[[nodiscard]]`, because it is obviously stateless. That attribute helped to detect when programmers accidentally assumed that empty() removes all element ("empty the range"). Now we compromise our own clarification, which demonstrates how bad this design is.

Printing elements may change/compromise behavior

Category:	**Runtime errors (undefined behavior)**
Effect:	Calling `if (v.empty())` may change program behavior.

In similar ways just adding a print statement (or a different read) can change the behavior of a program and compromise the state of a filter view.

See https://www.godbolt.org/z/v67vMP91r for a corresponding C++23 example:

```
// view for all even elements:
auto vColl = coll | std::views::filter([] (auto v) { return v%2 == 0; });
…
if (…) {
  std::println("vColl: {}", vColl);   // OOPS: println() changes program state
}
…

// insert a new element at the front:
coll.insert(coll.begin(), 0);
…

std::println("vColl: {}\n", vColl);   // OOPS: output depends on if println() was called
```

In this code the conditional call of println() changes what happens later in the program. The reason is that a print caches begin() and performing a non-transparent caching may have a huge impact on the behavior of a program.

Just to make clear how bad this design is:

If programmers see a problem and add a print statement to better understand the problem, the program changes its behavior.

It is an incredible huge surprise when a print statement changes the state and behavior of a program. Adding a debug output or adding an early check to return on empty() must be transparent for the state of a program. Any design that violates these basics is a trap and recipe for a severe mess.

This bad design is not only weird. It makes code incredibly fragile because for programmers code works fine until it doesn't because a subtle change has unexpected fatal consequences. **This makes printing elements and calling empty() a fragile call !!! Nobody, really nobody expects this. This design breaks one of the most important rules of good library design: Don't surprise the programmer.**

Filters on vectors and lists are not consistent (and break different use cases)

Category:	Runtime errors (undefined behavior)
Effect:	Switching from vectors to lists and other subtle changes in the underlying ranges and pipelines change/break behavior.

Surprisingly, the issue described before might depend on the container category.
Here is a simple example (see https://www.godbolt.org/z/cjxKv8crx for the full example):

```
std::list coll{2, 3, 4, 5};

// view for all odd elements:
auto odd = [] (auto v) { return v % 2 != 0; };
auto vOdd = coll | std::views::filter(odd);
if (vEven.empty()) return;      // OOPS: may change program state
…

// insert a new element at the front:
coll.insert(coll.begin(), 1);   // coll is now: 1 2 3 4 5
…

print(vOdd);                    // OK:  1 3 5
```

If we **switch from a list to a deque or vector**, the output of the program changes:

```
std::vector coll{2, 3, 4, 5};
coll.reserve(100);

// view for all odd elements:
auto odd = [] (auto v) { return v % 2 != 0; };
auto vOdd = coll | std::views::filter(odd);
if (vEven.empty()) return;      // OOPS: may change program state
…

// insert a new element at the front:
coll.insert(coll.begin(), 1);   // coll is now: 1 2 3 4 5
…

print(vOdd);                    // OOPS:  0 1 3 5
```

The reason for this behavior is, that the filter view caches different things depending on the category of the underlying range:

- The filter view caches an offset for random-access containers (so that for filters a reallocation of the underlying range does not invalidate the filter).
- The filter view caches an iterator in all other cases.

As the example demonstrates, the effect can be that switching from one container to another has unexpected functional consequences. Again we cause runtime errors that nobody sees coming. Nobody expects that a working filter suddenly does no longer work just because a slightly different underlying range is used

Pass-by-value might change state/behavior

Category:	Runtime errors (undefined behavior)
Effect:	A key use case of results in undefined behavior. Wrong/invalid elements may be processed.

A view shall be cheap to copy. As a consequence, it is fine to pass it by value. However, it should simply not matter whether we pass by value or by reference (this is one of the basics of good regular behavior). But for the filter view this is not the case.

When filters cache iterators (e.g. operating on a list) a copy of the view loses any cached begin(). Only when a filter on a random-access container is copied, the cached offset is kept alive with the copy.

This leads to the following effect **(see** https://www.godbolt.org/z/8WfxbsahT **for the full example**:

```
void printByVal(auto coll);      // just print the elements
void printByVRef(auto&& coll);   // just print the elements
…

std::set coll{1, 2, 3, 4};

auto even = [] (auto v) { return v%2 == 0; };
auto collEven = coll | std::views::filter(even);
…

if (!collEven.empty()) {
  coll.insert(0);              // coll is: 0 1 2 3 4
}
…

printByVal(collEven);          // OK, prints 0 2 4
printByRef(collEven);          // OOPS, prints 2 4
printByVal(collEven);          // OK, prints 0 2 4
printByRef(collEven);          // OOPS, prints 2 4
```

As this example demonstrates, switching between call-by-value and call-by-reference may change the behavior of a program when passing a view.

Again, this is an unexpected runtime error. Programmers have to be aware that it might matter for views whether they are passed by value or by reference.

You could argue that in all these situations the problem is that an underlying range gets modified between different uses of the view. However, the point is that programmers do not see how fragile their code is as long as it works. And suddenly with a simple change the code is broken although there is absolutely no intuitive assumption or awareness that something like this can happen.

Good design protects programmers from doing bad things instead of providing traps only avoidable with expert knowledge.

And again, if there would be a good reason for this mess, fine. But there isn't.

Summary of the Design and Its Consequences

The Current Situation of Filter Views

For filter views, we currently have the following situation:

- Filter views cache begin() to avoid that you may get bad runtime when performing a reverse iteration over the elements of a filter view. Note that a couple of additional preconditions have to be met to get this bad performance (e.g. that there is usually no element that fits a filter early in the collection).
- The resulting caching of begin() causes several severe problems for basic real-world use cases. Even simple use cases of filter views cause confusing compile-time errors and severe runtime errors.
- No doubt, by nature, there are use cases where filter views in principle cannot work correctly. However, these failures are on non-trivial use cases and more or less intuitive. All problems described here are not natural for filter views and can easily be avoided by a better design.

Taking into account that the use cases with bad performance for reverse iterations are very rare and the price is a huge amount of confusion and several runtime errors in very simple standard use cases, the current design is simply an unnecessary trap the C++ standard committee provides for ordinary programmers.

Why are these problems so severe?

What make these problems so severe is that almost all of them result in subtle runtime errors. That means that code that looks like being correct (both because there is no reason to assume that something is broken and because all test code works fine) might suddenly break when:

- Inserting a filter
- Adding a check on empty()
- Adding a print statement
- Switching from one underlying container type to another
- Switching between call-by-value and call-by-reference
- Using other initial values
- Assigning other values (or using other slightly different modifications)
- Using multiple threads

What are the consequences for using filter views in general?

As a consequence of the current design, programmers have to know about all of these non-obvious limitations and have to follow important non-intuitive recommendations when using filter views (or know very exactly what they (can) do):

- Give up using `const` in generic code for a range even if the code only reads.
- Use universal references in generic code even if is only reads.
- Do never apply a filter view early on an underlying collection. Apply the filter view (and a pipeline using a filter view) to an underlying range always ad-hoc, right when performing the iteration.
- Iterate only once with an applied filter view.
- Never put a reverse view behind a filter view (unless you only read and elements are never modified).
- Never use a filter view to select elements to “heal” them or modify them in other ways.
- Never use a filter view in multi-threaded code.

A resulting simple advice is something like this:

> **Use filter views only ad-hoc in a single threaded environment and let their iterators act as if they are const input iterators (can only iterate once to read) on elements that never change.**

Companies, projects, and programmers simplify this consequences by **not allowing to use filters views** and views at all.

These are dramatical restrictions. And they are simply no necessary. They are a consequence of an incredible bad design.

Which Design Alternatives Exist?

C++ is a language that cares for good performance. We usually try to avoid bad performance. One typical way is to deal with possible bad performance is to ensure that bad performance results in a compile-time error. That is why vectors do not have `push_front()`, lists have no index operator and, yes, filter views do not provide a `size()` member function.

However, we cannot disable the basic operations begin() and end() to iterate over a filter view. Thus, we better do something, when begin() gets quadratic behavior in some use cases. Ideally, we avoid the resulting quadratic complexity. But if the price for avoiding quadratic complexity is too high, we should simply disable problematic use cases to compile provided this happens rarely and does not compromise basic filter use cases.

With that in mind let us look what design options we have for filter views:

Alternative A: Cache begin()

This is the current design.

As a consequence we **break** several basic patterns of C++ well established over decades:

a) You can read iterate if the range is const
b) A read iteration does not change state
c) empty() doesn't have side effects
d) Concurrent read iterations are safe
e) A copy of a range has the same state
f) Modifications between iterations are safe
g) Modifications via iterations are safe

Only a) results in a compile-time error (the error message is hard to understand and it breaks the key C++ principle of const-ness). All other consequences may cause severe runtime errors with non-intuitive behavior and more or less fatal undefined behavior.

Breakage g) especially means that you cannot you a filter view for one of its main purpose: Filtering broken elements to heal them.

If you look at how rare the use cases are that cause this mess, choosing this option is obviously a huge design mistake (I still struggle to have a compelling answer, when attendees of my C++ trainings ask me how this design could be standardized).

Alternative B: begin() is initialized during construction

The consequence of this design would be that a declaration of a filter view may have linear complexity. In addition, you could not modify elements after the declaration.

Both are significant drawbacks, we should avoid.

Alternative C: Cache begin(), but make it transparent and const by using mutable

With this approach each call of begin() still is expensive, but for another reason. To be thread-safe, you would need a mutex that is locked for each call of begin() and empty(). This again makes both the first and additional calls of begin() and empty() significant slower.

This is a significant drawback, we should avoid.

Alternative D: Do nothing (i.e.: do not cache begin() at all)

This would heal all of the broken compile-time and runtime errors in the use cases described above.

However, this approach can result in the discussed quadratic complexity. We could argue that programmers then should do something else, which they always can do. However, we can support this approach in many ways:

- We can let the reverse view cache end(). In fact, it is not clear why we currently solve the problem of a reverse iteration inside the filter view with the effect that each many basic use cases of filtering are compromised.

- We could offer a view that inside a pipeline caches begin() in a way the subrange view does. That would look similar to the following:

```
        v = coll | std::views::cacheBegin
                 | std::views::filter(every7th) | std::views::reverse;
```

As a drawback to the current design, with this approach an empty() check early before we iterate would double the time to call begin() when we really start to iterate later.

And for reverse iterations, this would put the burden of avoiding quadratic complexity to the programmer. However, this is definitely way better than to put incredible burden on each and every ordinary programmers due to the non-intuitive breakage of several trivial and basic use cases, patterns, and key principles of C++.

Alternative E: Do not cache begin() but let filter iterators only be forward iterators

Again, this would heal all of the broken compile-time and runtime errors in the use cases described above and also an empty() check early before we iterate would double the time to call begin() when we really start to iterate later.

However, reverse iterations of filter views would longer compile. This also removes the possibility of bad performance in this scenario, but you would have to modify your code to make it compile (which is possible with no big effort).

A drawback of this approach is that code that calling begin() again and again for other reasons than reverse will still result into quadratic complexity. The question is how likely it is to happen accidentally. If it is clear that begin() may have linear complexity, a limited number of multiple calls of begin() are linear but create wors performance. Again simple workarounds are easy.

Alternative F: Do not cache begin() but let filter iterators only be input iterators

Like for alternative E:

- This would heal all of the broken compile-time and runtime errors in the use cases described above.
- An empty() check early before we iterate would double the time to call begin() when we really start to iterate later.
- This would cause compile-time errors on reverse iterations.

In addition, this could also create compile-time errors when we iterate multiple times over all elements. However, it depends whether the iterator category is checked by concepts applied to the passed range we iterate over in generic code.

However, there are many basic use cases that require forward iterators to support multiple iterations or having multiple independent iterators into a range at any given time (for example, to support min_element() as discussed in https://brevzin.github.io/c++/2023/04/25/mutating-filter/).

Comparing the drawbacks with the possible benefits of only supporting input iterators, I see no real benefit in this alternative compared to providing forward iterators. One reason is that multiple iterations do still have linear complexity.

Proposed Fix

This paper proposes alternative E:

- **Remove caching and**
- **Restrict filter view iterators to be at most forward iterators.**

That way, we heal several broken basic use cases causing confusion with unexpected compile-time errors and bad or undefined runtime errors and make filter views act way more intuitive (empty() is stateless, read iterations are const and can happen concurrently, modifications work fine and as expected in all trivial use cases).

The drawback is that reverse iterations are no longer supported. However, code with reverse iterations is incredible fragile and could cause severe damage with small modifications so that this fix is probably good.

How about backward compatibility to C++20?

C++ is out now for 4 years and we might already have some code using filter views that benefit from caching. If we apply alternative E or F. this means:

- Code that performs the empty() check at the beginning might become slower in some situations, but still have linear complexity.
- Code that does reverse iterations on a filter view will no longer compile.

Note that **binary backward compatibility is not a problem.** Implementations caring for binary compatibility could simply keep the member for the cached value of begin() without using it.

What are the consequence of this fix?

The proposed fix has some consequences.

Changing the requirements for begin() of ranges concepts

The most important consequence of this fix is that, because a filter view still is a view, we have to relax the requirements for range concepts.

begin() should only require to be linear

begin() can no longer require that is has amortized constant complexity. It should only require linear complexity.

Is this a problem? Note that calling begin() usually is only one single call in a complete iteration having the same cost as ++. In a loop that iterates from element to element the search for the first element is nothing different than the search of any other next element. If we use two iterators or perform two iterations, any call of looking for the next element is doubled. No longer caching begin() means that we only no longer lose time for the first search of the next element done with begin(). We still have the same complexity as each ++ has its cost multiple time with multiple iterations.

The special problem was that we call a linear begin() with each and every ++. We only need that when comparing against the end() after each ++ which is required when performing a reverse iteration on a filter view. That use cases causing quadratic complexity no longer compile and can therefore no longer occur.

Relaxing the requirement for begin() therefore in practice seems to fall in the same category as relaxing the requirements for views a couple of times in the past. We changes the complexity of begin() and destructors and more in the past.

Also note that it looks like filter view currently violates the requirements of ranges at all because begin() is required to be non-modifying, which currently is clearly not the case for filter views as it modifies the member for the cached begin().

Therefore, this fix might also heal the current specification that a filter view is not a range at all.

Yes, we should teach the resulting complexity when teaching filter views, which is nothing new. However, we not only have to teach all the fatal consequences of caching.

end() should still be amortized constant (or only constant)

Note that this change does not mean that end() relaxes its requirements. In fact, end() should still require amortized constant complexity.

Well, I see no reason not even to require constant complexity. This would require that end() always delegates to a cheap underlying end() or is cached, which as we often compare against end() sounds reasonable.

Consequence for the reverse view

The reverse view might anyway better in general cache its end(), which is the underlying begin(). This, however, could be a binary break. Or we make it undefined behavior if this is note that case.

We might for this reason stay with amortized constant complexity for end() for ranges in general.

Summary

The current design of the filter view is severely broken. It compromises multiple basic pattern of C++ to improve the performance of some rare use cases, causes non-intuitive compile-time errors and fatal non-intuitive runtime errors on several basic use cases.

As a result, C++ looks like giving up the principle of `const` for read access:

- empty() suddenly might modify
- a read suddenly might modify
- a `const` suddenly disables a read
- A read suddenly is no longer thread-safe

Such a design kills the credibility of the C++ programming language and the C++ standard committee at all.

This has the consequence of frustrating programmers, causing fatal runtime errors and that views are banned from being used by several companies and projects. This compromises any claim of C++ that we care for safety.

All these consequence are a pity, because views have the power to be a very intuitive first class library for the mass. But with the current design it is in no way acceptable to be used by any serious C++ programmer unless they know all of the pitfalls and traps, which is simply impossible.

The proposed fix would heal the filter view so that is just works in simple intuitive use-cases without easily getting into the worst case quadratic complexity.

The proposed fix makes filter views usable in practice.

FAQ

Should we Fix Other Views the Same Way?

This paper concentrates on the filter view because it is a key view (filtering elements is a common use case when processing elements of ranges) and certainly special.

Other views have the same problem and I am happy if we fix the whole design of the views library to make it simple and robust for basic use cases and make the design consistent, KISS, and not breaking several basic established patterns of range processing. However, nothing is as important as healing the filter view.

However, note that if we do similar fixes on other views, additional consequences might be necessary. For example, the drop view might no longer be a random access view although the underlying range is.

Is this the only thing we have to fix for filter views?

For filter views const behaves weird in a different sense: const sometimes is propagated to elements and sometimes it is not. While a const container and array always has const elements, this principle is sometimes broken for filter views (it depends on the value category of the underlying range). This behavior has to do with is reference semantics and there are some arguments for this confusing behavior. No question this could also be healed, the current runtime issues are far more severe.

So, in this document, we propose only healing the obvious bad design that causes real harm on programmers and the reputation of C++.

Why didn't you bring this fix up earlier?

I voted the ranges library into the C++20 standard, not being aware of this bad design. I trusted the experts that they will never come up with a library for the mass that compromises key principles of C++ and I am not aware that this key design decision was brought up to the mass (the way standardization is currently organized, nobody can follow everything).

When I started to understand what we standardized, I could not believe and said that. I simply wanted to understand the motivation. It was strange, instead of explaining the possible options and clearly explaining the design arguments I got comments such as "this design is in the nature of filter views" and "stop to talk bad about C++ and teach this design right".

I did and more and more companies and programmers could not believe how bad this design is. And the better I could explain the motivation (which definitely took a while) the worse that reaction was.

Finally, it took me 4 years to come up with the right examples and the right motivation for this design, which hopefully now demonstrates fair and precise how severe the problem is as well as how unnecessary it is (thanks to Berry Revzin and Jonathan Müller, which finally took their time to explain/discuss details).

I still might be wrong and don't see the picture as a whole. But there is no doubt that the use cases I describe here are real and convincing to reveal that there is a severe issue. And these use cases would all be fixed.

Isn't it too late to fix it now?

Definitely not. In the past I did fight for fixing std::thread, atomics and the range-based `for` loop. All of them were used widely. Filter views are still brand new and the fix does only in rare use cases have consequences. Some use cases will then result into compile-time errors (which is good because they are extremely fragile). I some situations the running time will suffer, but not switching from linear to quadratic complexity. There are easy workarounds for these consequences.

Note also that the fix doesn't break binary compatibility.

In any case, everything is way better than leaving the door open for a huge mass of confusion, inconsistencies, and fatal runtime errors.

Proposed Wording

(All against N????)

In 26.4.2 Ranges [range.range]

Modify concept std::ranges::range as follows:

3 Given an expression t such that decltype((t)) is T&, T models range only if

(3.1) — [ranges::begin(t), ranges::end(t)) denotes a range (25.3.1),

(3.2) — ~~both~~ ranges::begin(t) ~~and ranges::end(t) are~~ ~~amortized constant time~~ is linear and non-modifying,

(3.2) —ranges::end(t) is amortized constant time and non-modifying,

…

In 26.7.8.2 Class template filter_view [range.filter.view]

Modify the definition of class filter_view as follows:

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

constexpr *iterator* begin() **const**;

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.~~

In 26.7.8.3 Class filter_view::*iterator* [range.filter.iterator]

strike:

~~1 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.~~

Modify as follows:

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

(2.1) — ~~If V models bidirectional_range, then iterator_concept denotes bidirectional_iterator_tag.~~

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

(2.3) — Otherwise, iterator_concept denotes input_iterator_tag.

In Annex C Compatibility add a section for Clause 26 ranges:

C.1.13 Clause 26: Ranges library [diff.cpp20.range.filter]

Affected subclause: 26.7.8

Change: …

Rationale: …

Effect on original feature: …

Feature Test Macro

…

Acknowledgements

Thanks to a lot of people who helped ad gave support again and again to come to finally get this proposal done. Special thanks go to Berry Revzin and Jonathan Müller which finally took their time to explain the motivation and details of the current situation.

Rev0:

First initial version.

References

A video about these issues of the filter view is available from my keynote at “using std::cpp 2024” at https://youtu.be/WOd8h1MB_nc.

https://brevzin.github.io/c++/2023/04/25/mutating-filter/