Taming the C++ Filter View

NICOLAI JOSUTTIS

Cppcon The C++ Conference

2024 September 15 - 20

September 17, 2024
16:45 - 17:45 MDT

C++

josuttis | eckstein IT communication

Nicolai M. Josuttis

- **Independent consultant**
  - Continuously learning since 1962
- **C++:**
  - since 1990
  - ISO Standard Committee since 1997
- **Other Topics:**
  - Systems Architect
  - Technical Manager
  - SOA
  - X and OSF/Motif

C++

josuttis | eckstein IT communication

C++20

Views

C++20: Views

C++20

```
template<typename T>
void print(const T& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector<int> coll1{0, 8, 15, 47, 11, 42, 1};
std::set<int> coll2{0, 8, 15, 47, 11, 42, 1};
print(coll1);
print(coll2);
```

Output:

```
0 8 15 47 11 42 1
0 1 8 11 15 42 47
```

C++20: Views

C++20

```
void print(const auto& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

std::vector<int> coll1{0, 8, 15, 47, 11, 42, 1};
std::set<int> coll2{0, 8, 15, 47, 11, 42, 1};
```

Output:

```
0 8 15 47 11 42 1
0 1 8 11 15 42 47
```


C++20: Views

C++20

```
void print(const std::ranges::input_range auto& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

std::vector<int> coll1{0, 8, 15, 47, 11, 42, 1};
std::set<int> coll2{0, 8, 15, 47, 11, 42, 1};
print(coll1);
print(coll2);
print(std::views::take(coll1, 3));      // print first three elements
print(std::views::take(coll2, 3));      // print first three elements
print(coll1 | std::views::take(3));     // print first three elements
print(coll2 | std::views::take(3));     // print first three elements

print(coll2 | std::views::take(3)
            | std::views::transform([](auto v){
                                      return std::to_string(v) + 's';
                                    }));
```

Output:

```
0 8 15 47 11 42 1
0 1 8 11 15 42 47
0 8 15
0 1 8
0 8 15
0 1 8
0s 1s 8s
```

Example of Pipeline of Range Adaptors

C++20

```
int main()
{
  std::map<std::string, int> composers {
    {"Bach", 1685},
    {"Mozart", 1756},
    {"Beethoven", 1770},
    {"Tchaikovsky", 1840},
    {"Chopin", 1810},
    {"Vivaldi", 1678},
  };

  // iterate over the names of the first 3 composers since 1700:
  namespace vws = std::views;
  for (const auto& elem : composers
   | vws::filter([](const auto& y) {   // since 1700
                   return y.second >= 1700;
                 })
   | vws::take(3)                      // first 3
   | vws::keys                         // names only
                                 ) {
    std::cout << "- " << elem << '\n';
  }
}
```

Output:

```
- Beethoven
- Chopin
- Mozart
```


Using a Pipeline of Views

C++20

```
// view 4th to 11th value that are multiples of 3 with suffix "s":
auto v = std::views::iota(1)                          // generates 1, 2, 3, ...
         | std::views::filter([] (auto val) {         // multiples of 3 only
                                return val % 3 == 0;
                              })
         | std::views::drop(3)                        // skip first 3
         | std::views::take(8)                        // take next 8
         | std::views::transform([] (auto val) {      // append "s"
                                   return std::to_string(val) + "s";
                                 });

for (const auto& elem : v) {
  std::cout << elem << '\n';
}
```

Output:

```
12s
15s
18s
...
33s
```

Using a Pipeline of Views

C++20

```
// view 4th to 11th value that are multiples of 3 with suffix "s":
auto v = std::views::iota(1)                          // generates 1, 2, 3, ...
         | std::views::filter([] (auto val) {         // multiples of 3 only
                                return val % 3 == 0;
                              })
         | std::views::drop(3)                        // skip first 3
         | std::views::take(8)                        // take next 8
         | std::views::transform([] (auto val) {      // append "s"
                                   return std::to_string(val) + "s";
                                 });

for (const auto& elem : v) {
  std:
}
```

Output:

```
12s
```

Type of v is:

```
std::ranges::transform_view<
 std::ranges::take_view<
  std::ranges::drop_view<
   std::ranges::filter_view<
    std::ranges::iota_view<int, std::unreachable_sentinel_t>,
    typeOf1stLambda>>>,
 typeOf2ndLambda>
```


Using a Pipeline of Views

C++20

```
// view 4th to 11th value that are multiples of 3 with suffix "s":
auto v = std::views::iota(1)                          // generates 1, 2, 3, ...
         | std::views::filter([] (auto val) {         // multiples of 3 only
                                return val % 3 == 0;
                              })
         | std::views::drop(3)                        // skip first 3
         | std::views::take(8)                        // take next 8
         | std::views::transform([] (auto val) {      // append "s"
                                   return std::to_string(val) + "s";
                                 });

for (const auto& elem : v) {
  std::cout << elem << '\n';
}
```

Output:

```
12s
15s
18s
...
33s
```

Using a Pipeline to Modify Elements

C++20

```
auto vNotEmpty = std::views::filter([] (const auto& rg) {
                                      return !rg.empty();
                                    });
```

```
std::vector<std::string> words{tic", "tac", "toe", "", "ok"};
print(words);

// modify all non-empty words:
for (int i = 0; std::string& s : words | vNotEmpty) {
  s = std::to_string(++i) + ':' + s;
}
print(words);
```

Output:

```
tic tac toe  ok
1:tic 2:tac 3:toe  4:ok
```


C++20

How Views Operate

C++20: How Views Operate

C++20

```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);
```

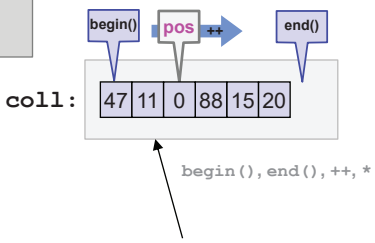
Output:
```
47 11 0 88 15 20
47
11
0
...
```

print():

```
for (auto pos = coll.begin(); pos != coll.end(); ++pos) {
  std::cout << *pos << '\n';
}
```

print() step by step:

```
auto pos = coll.begin();
if (pos != coll.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != coll.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != coll.end())
  std::cout << *pos << '\n';
  ++pos;
...
```



C++

josuttis | eckstein IT communication

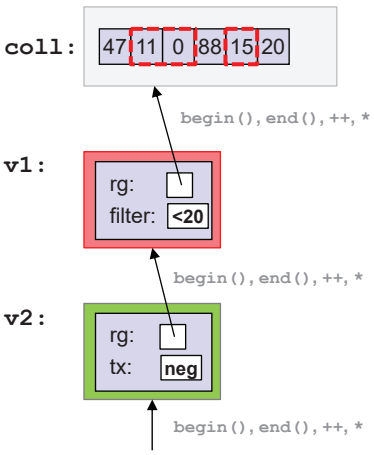
C++20: How Views Operate

C++20

```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

Output:
```
47 11 0 88 15 20
-11 0 -15
```



C++

josuttis | eckstein IT communication

C++20: How Views Operate

C++20

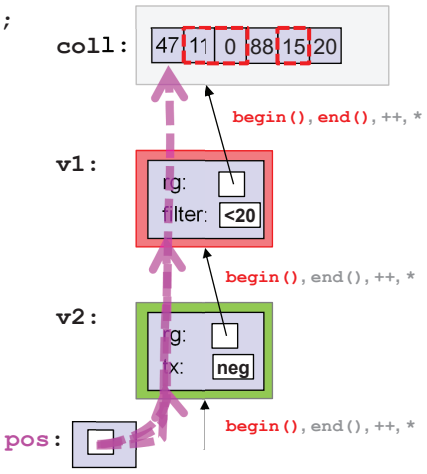
```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

Output:
```
47 11 0 88 15 20
-11 0 -15
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
```



C++20: How Views Operate

C++20

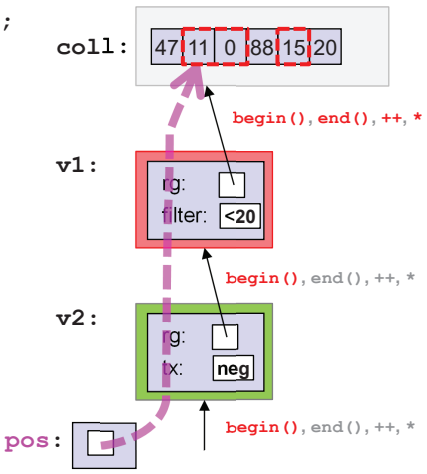
```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

Output:
```
47 11 0 88 15 20
-11 0 -15
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
```



C++20: How Views Operate

C++20

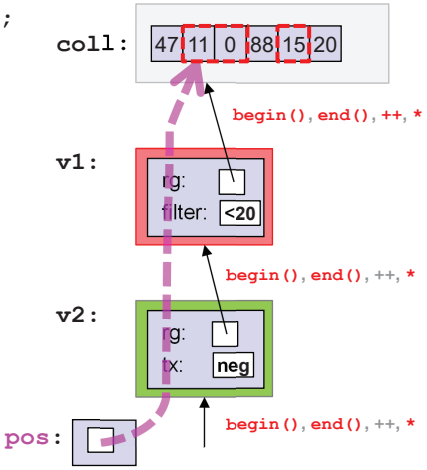
```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

Output:
```
47 11 0 88 15 20
-11 0 -15
-11
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
if (pos != v2.end())
  std::cout << *pos << '\n';
```



C++20: How Views Operate

C++20

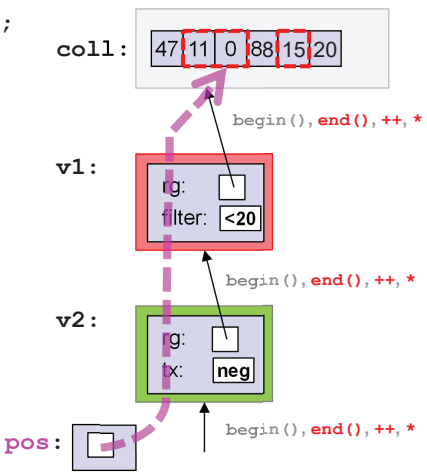
```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

Output:
```
47 11 0 88 15 20
-11 0 -15
-11
0
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
  std::cout << *pos << '\n';
```



C++20: How Views Operate

C++20

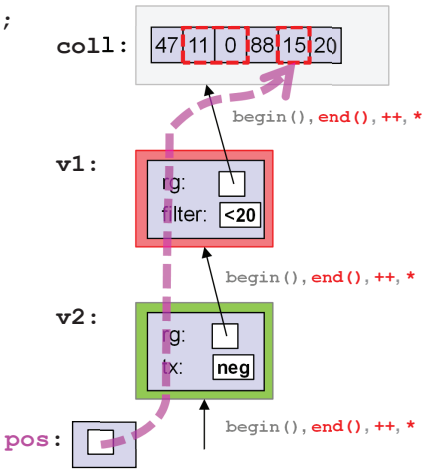
```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

Output:
```
47 11 0 88 15 20
-11 0 -15
-11
0
-15
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
  std::cout << *pos << '\n';
```



C++20: How Views Operate

C++20

```
std::vector<int> coll{47, 11, 0, 88, 15, 20};
print(coll);

auto less20 = [] (auto val) { return val < 20; };
print(coll | std::views::filter(less20)
           | std::views::transform(std::negate{}));
```

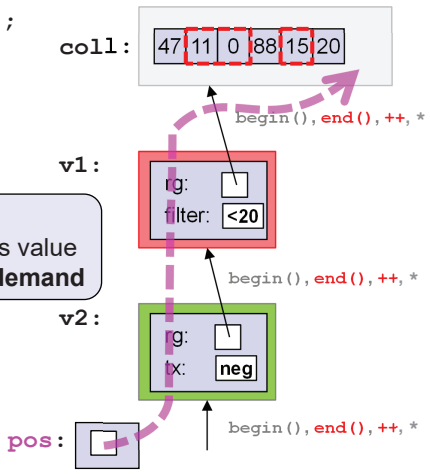
Output:
```
47 11 0 88 15 20
-11 0 -15
-11
0
-15
```

```
auto v1 = std::views::filter(coll, less20);
auto v2 = std::views::transform(v1, std::negate{});
```

print() step by step:

```
auto pos = v2.begin();
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
  std::cout << *pos << '\n';
  ++pos;
if (pos != v2.end())
```

Pull model:
next element and its value are processed **on demand**

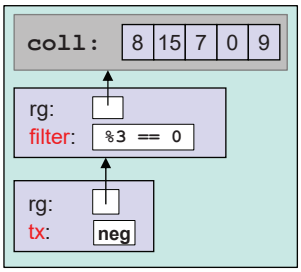


C++
©2024 by josuttis.com
20
josuttis | eckstein IT communication

Pipelines: Lazy Evaluation

C++20

```
std::vector<int> coll{ 8, 15, 7, 0, 9 };

// define a view on coll:
auto vColl = coll
             | std::views::filter([] (int i) {
                 std::cout << " filter "
                           << i << '\n';
                 return i % 3 == 0;
               })
             | std::views::transform([] (int i) {
                 std::cout << "  transform "
                           << i << '\n';
                 return -i;
               });

// and use it:
std::cout << "coll | filter | tx:\n";

for (int val : vColl) {
  std::cout << "val: " << val << '\n';
}
```



Output:

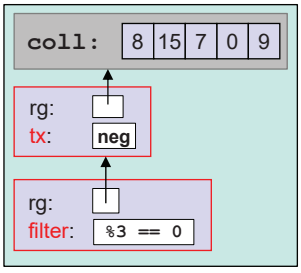
```
coll | filter | tx:
  filter 8
  filter 15
    transform 15
val: -15
  filter 7
  filter 0
    transform 0
val: 0
  filter 9
    transform 9
val: -9
```

Pull model **(lazy evaluation):** next element and its value are processed **on demand**

Pipelines: Price of Filter Views

C++20

```
std::vector<int> coll{ 8, 15, 7, 0, 9 };

// define a view on coll:
auto vColl = coll
             | std::views::transform([] (int i) {
                 std::cout << " transform "
                           << i << '\n';
                 return -i;
               })
             | std::views::filter([] (int i) {
                 std::cout << "  filter "
                           << i << '\n';
                 return i % 3 == 0;
               });

// and use it:
std::cout << "coll | tx | filter:\n";

for (int val : vColl) {
  std::cout << "val: " << val << '\n';
}
```



Output:

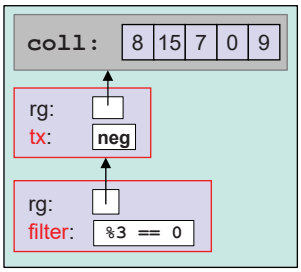
```
coll | tx | filter:
  transform 8
    filter -8
  transform 15
   filter -15
  transform 15
val: -15
  transform 7
   filter -7
  transform 0
   filter 0
  transform 0
val: 0
  transform 9
    filter -9
  transform 9
val: -9
```

Pipelines: Price of Filter Views

C++20

```
std::vector<int> coll{ 8, 15, 7, 0, 9 };

// define a view on coll:
auto vColl = coll
           | std::vie
               std::c
               return -i;
             })
           | std::views::filter([] (int i) {
               std::cout << "  filter "
                         << i << '\n';
               return i % 3 == 0;
             });

// and use it:
std::cout << "coll | tx | filter:\n";

for (int val : vColl) {
  std::cout << "val: " << val << '\n';
}
```

- **Filters** need the value to check, but pass its position (iterator)
- **Callers** need the value **again** to process



Output:

```
coll | tx | filter:
  transform 8
    filter -8
  transform 15
   filter -15
  transform 15
val: -15
  transform 7
    filter -7
  transform 0
    filter 0
  transform 0
val: 0
  transform 9
    filter -9
  transform 9
val: -9
```

- Use **filters early**
- Avoid expensive stuff before using them

Performance of Views

C++20

- **Good compilers optimize a lot**
  - but not everything (yet)

```
for (int v : rg | std::views::filter([] (int i) { return i % 3 == 0; })
               | std::views::transform([] (int i) { return -i; }) ) {
  process(v);
}
```

```
for (auto pos = rg.begin(); pos != rg.end(); ++pos) {
  if (*pos % 3 == 0) {
    process(-*pos);
  }
}
```

```
for (int v : rg | std::views::transform([] (int i) { return -i; })
               | std::views::filter([] (int i) { return i % 3 == 0; }) ) {
  process(v);
}
```

```
for (auto pos = rg.begin(); pos != rg.end(); ++pos) {
  if (-*pos % 3 == 0) {
    process(-*pos);
  }
}
```

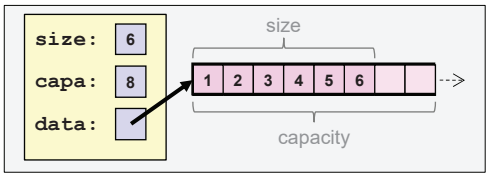
C++20

Performance of Filtering

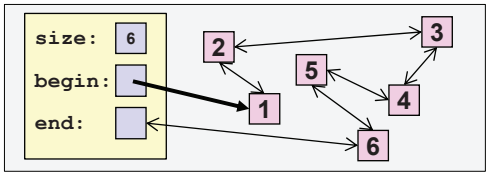
Performance of Containers

C++98

vector<>:



list<>:

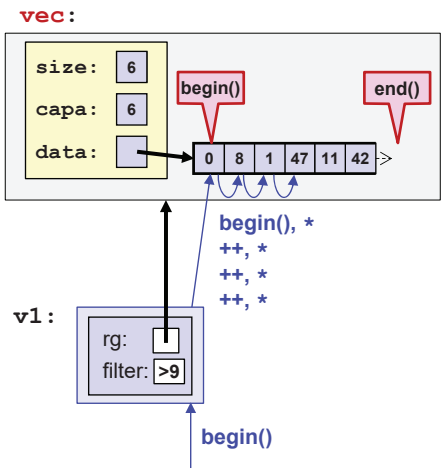


no operator `[]`

- **Declaration / default initialization is cheap**
- `begin()` (go to the first element) **is cheap**
- `end()` (go to the position behind the last element) **is cheap**
- `empty()` (check for no elements) **is cheap**
- `size()` (ask for the number of elements) **is cheap**
  or not provided: `forward_list<>`
- `operator[]` (jump to a specific elements) **is cheap**
  or not provided: only for random-access containers (vector, arrays, deque)

How Expensive is `begin()` for a Filter View ?

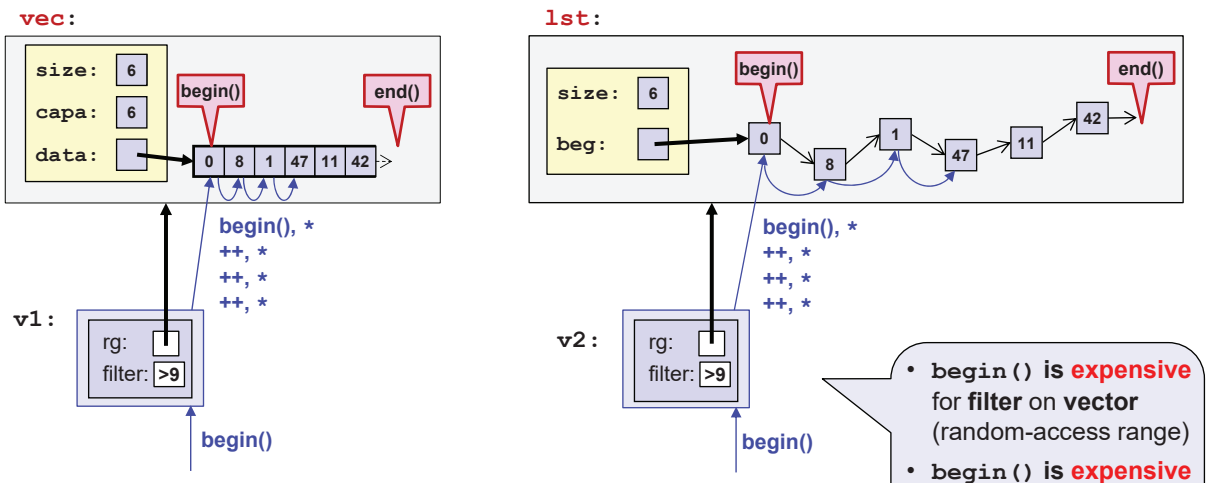
C++20




```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
```


How Expensive is `begin()` for a Filter View ?

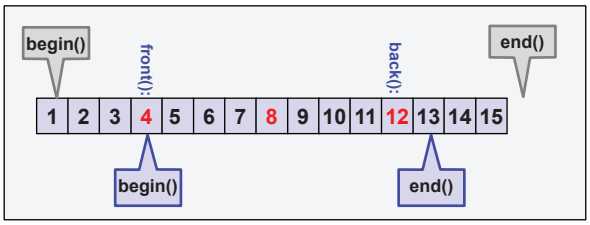
C++20



- `begin()` is **expensive** for **filter** on **vector** (random-access range)
- `begin()` is **expensive** for **filter** on **list** (forward range)

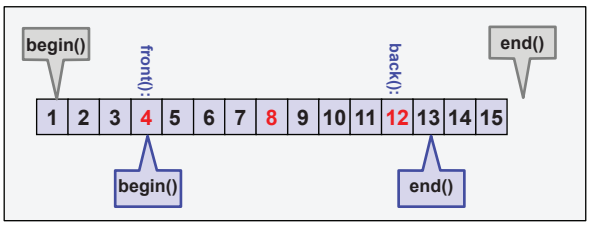
```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();

std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```

Performance of Filtering



- **Declaration / default initialization is cheap**
- `begin()` **can be expensive** (iterate to the first match)
- `end()` **can be expensive** (iterate behind the last match)
- `empty()` **can be expensive** (iterate to the first match)
- `size()` **is expensive** (iterate over all elements)
- `operator[]` **is expensive** (iterate over all elements)

Performance of Filtering



- **Declaration / default initialization is cheap**
- `begin()` **can be expensive** (iterate to the first match)
- `end()` **can be expensive** (iterate behind the last match)
- `empty()` **can be expensive** (iterate to the first match)
- ~~`size()` **is expensive** (iterate over all elements)~~
- ~~`operator[]` **is expensive** (iterate over all elements)~~

⇨ **Filter iterators do not support random access**

Using Filter Views in Generic Code

C++20

```
std::vector<int> coll{1, 2, 3, 4, 5, 6, 7, 8, 9};

auto isEven = [] (const auto& val) { return val % 2 == 0; };
auto collEven = coll | std::views::filter(isEven);

if (collEven.size() == 0) return;   // ERROR
if (collEven.empty()) return;       // OK

std::ranges::sort(coll);            // OK
std::ranges::sort(collEven);        // ERROR: no random_access_range
```

- Prefer `empty()` over `size()`

concept in `std::ranges`

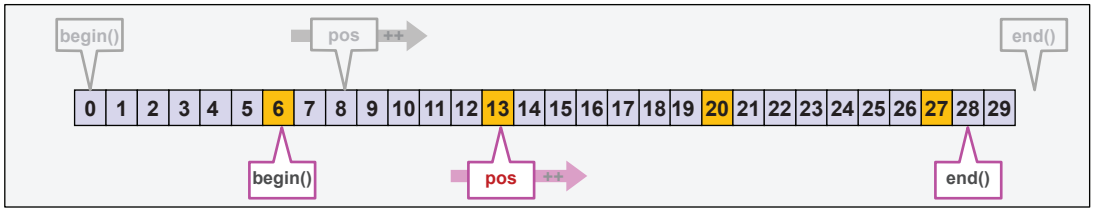
C++20

Iterating Over a Filter Views

Iterating Over a Filter View

C++11/C++20

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



filter

- **Range-based iteration:**

```
for (const auto& elem : rg) {
  process(elem);
}
```

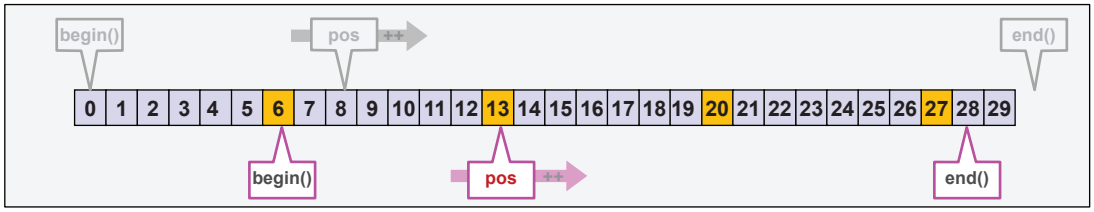
One call of **begin()**
One call of **end()**

```
auto beg = rg.begin();
auto end = rg.end();
for (auto pos = beg; pos != end; ++pos) {
  process(*pos);
}
```


Iterating Over a Filter View

C++98/C++20

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



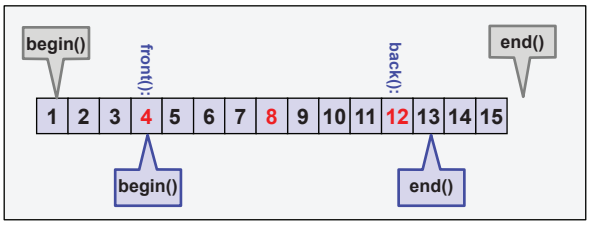
filter

- **Manual iteration:**

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

One call of **begin()**
Multiple calls of **end()**

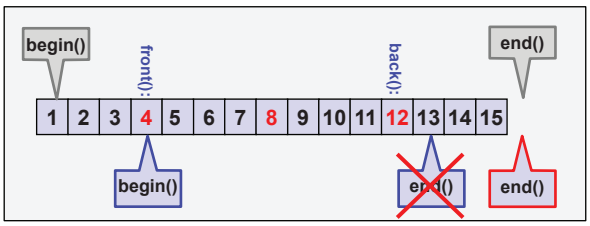
Performance of Filtering



- Declaration / default initialization is cheap
- `begin()` can be expensive (iterate to the first match)
- `end()` can be expensive (iterate behind the last match)
- `empty()` can be expensive (iterate to the first match)
- ~~`size()` is expensive (iterate over all elements)~~
- ~~`operator[]` is expensive (iterate over all elements)~~

⇨ Filter iterators do not support random access

Performance of Filtering



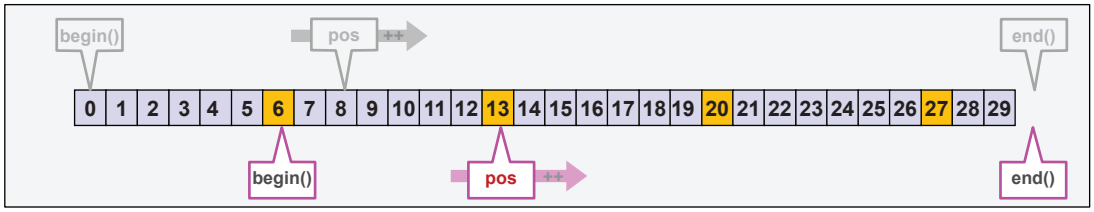
- Declaration / default initialization is cheap
- `begin()` can be expensive (iterate to the first match)
- `end()` is cheap (just the underlying end)
- `empty()` can be expensive (iterate to the first match)
- ~~`size()` is expensive (iterate over all elements)~~
- ~~`operator[]` is expensive (iterate over all elements)~~

⇨ Filter iterators do not support random access

Iterating Over a Filter View

C++11/C++20

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



filter

- ***Smart*** **iteration:**

```
if (rg.empty()) return;
...
for (const auto& elem : rg) {
  process(elem);
}
```

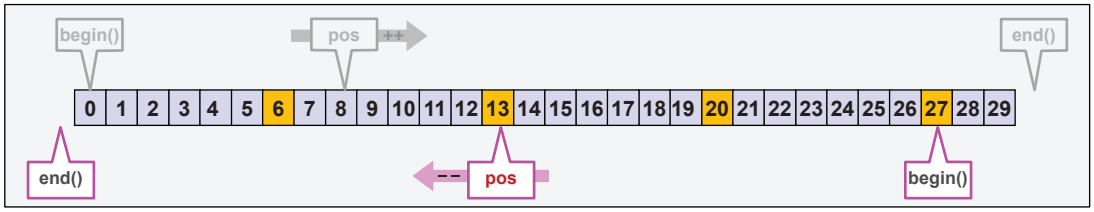
Two calls of **begin()**
One call of **end()**

```
if (rg.empty()) return;   // aka begin()==end()
...
auto beg = rg.begin();
auto end = rg.end();
for (auto pos = beg; pos != end; ++pos) {
  process(*pos);
}
```


Reverse Iterating Over a Filter View

C++11/C++20

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



reverse on filter

- **Reverse range iteration:**

```
for (const auto& elem : rg) {
  process(elem);
}
```

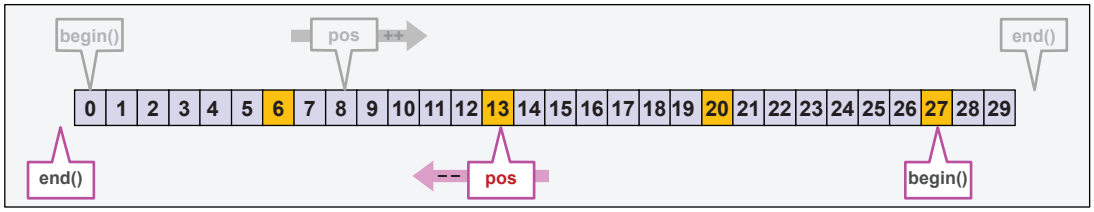
One call of **begin()**
One call of **end()**

```
auto beg = rg.begin();   // calls end()
auto end = rg.end();     // calls begin()
for (auto pos = beg; pos != end; ++pos) {
  process(*pos);
}
```

Reverse Iterating Over a Filter View

C++11/C++20

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



reverse on filter

- **Reverse manual iteration:**

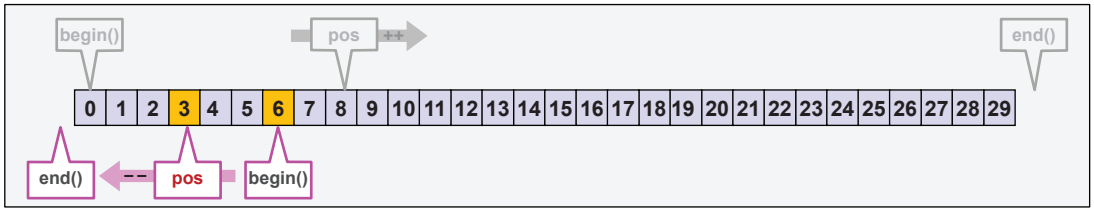
```
for (auto pos = rg.begin(); pos != rg.end(); ++pos) {
  process(*pos);
}
```

Multiple calls of **begin()**
One call of **end()**

Reverse Iterating Over a Filter View

C++11/C++20

```
v = coll | std::views::filter(3and6) | std::views::reverse
```



reverse on filter

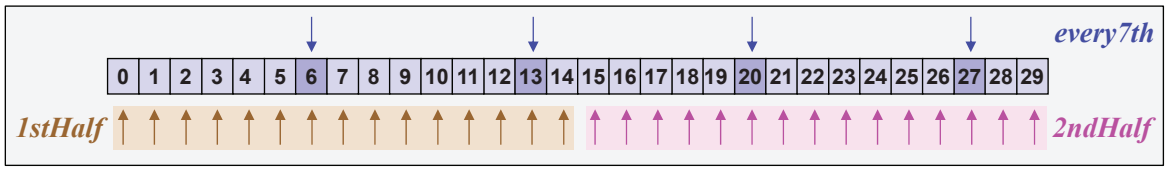
- **Reverse manual iteration:**

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

Multiple calls of **begin()**
One call of **end()**

In this scenario
begin() grows **quadratically**

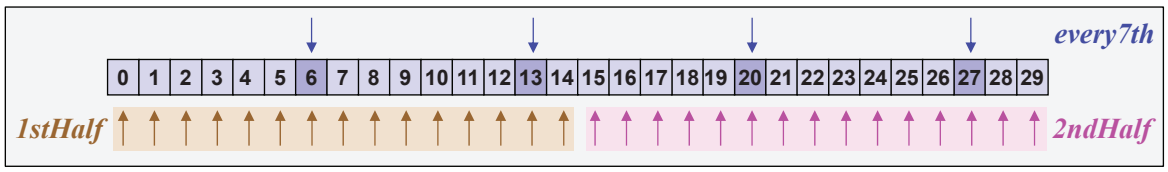
Performance of Naive Filtering



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

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

Performance of Filter View



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

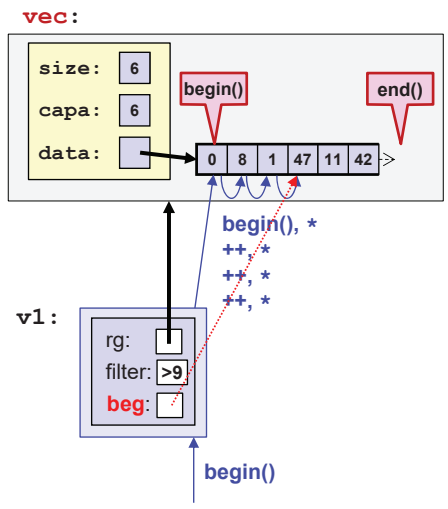
while(pos!=end())	2000	4000	8000	16000	32000	64000	128000	256000
filter(every7th)	0.002	0.003	0.006	0.012	0.026	0.051	0.101	0.220
filter(1stHalf)	0.003	0.005	0.011	0.019	0.041	0.080	0.160	0.322
filter(2ndHalf)	0.002	0.004	0.009	0.018	0.035	0.075	0.151	0.316
filter(2ndHalf) | reverse	0.003	0.005	0.010	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

C++20

Filter View Caching

Caching `begin()`

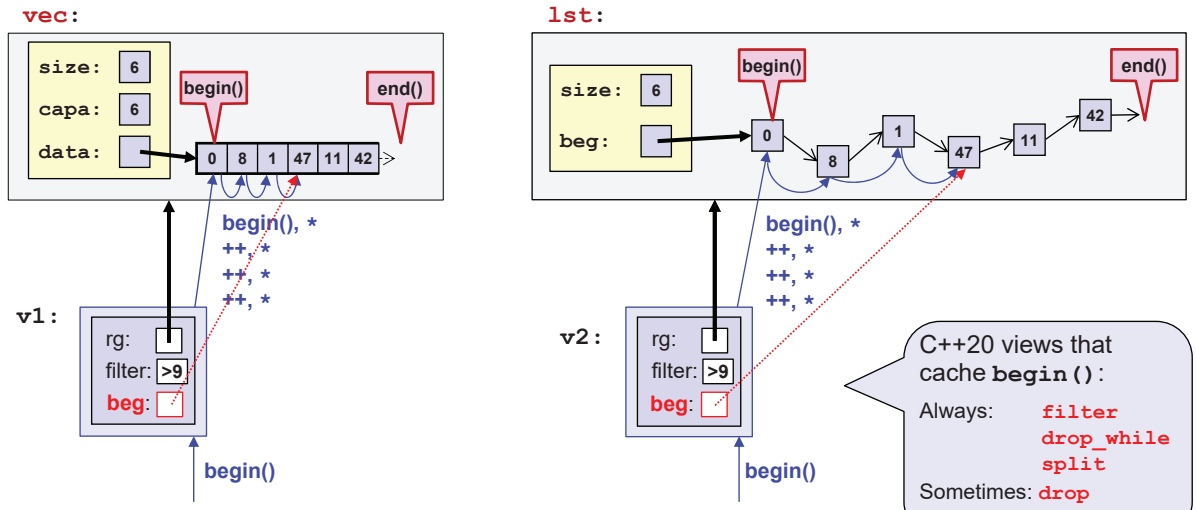
C++20




```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
```

Caching `begin()`

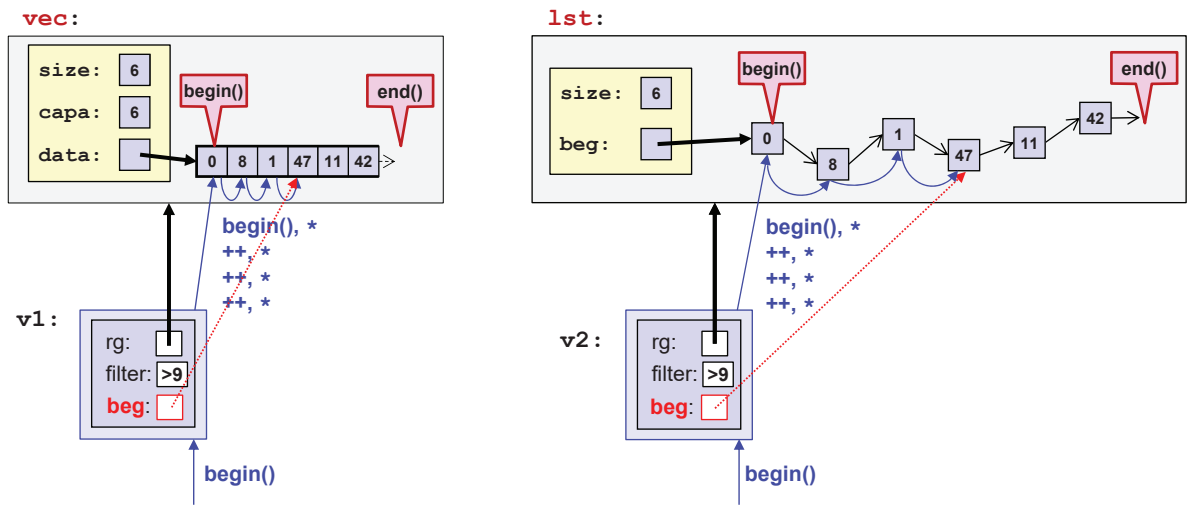
C++20




```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();

std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```


Ways of Caching `begin()`

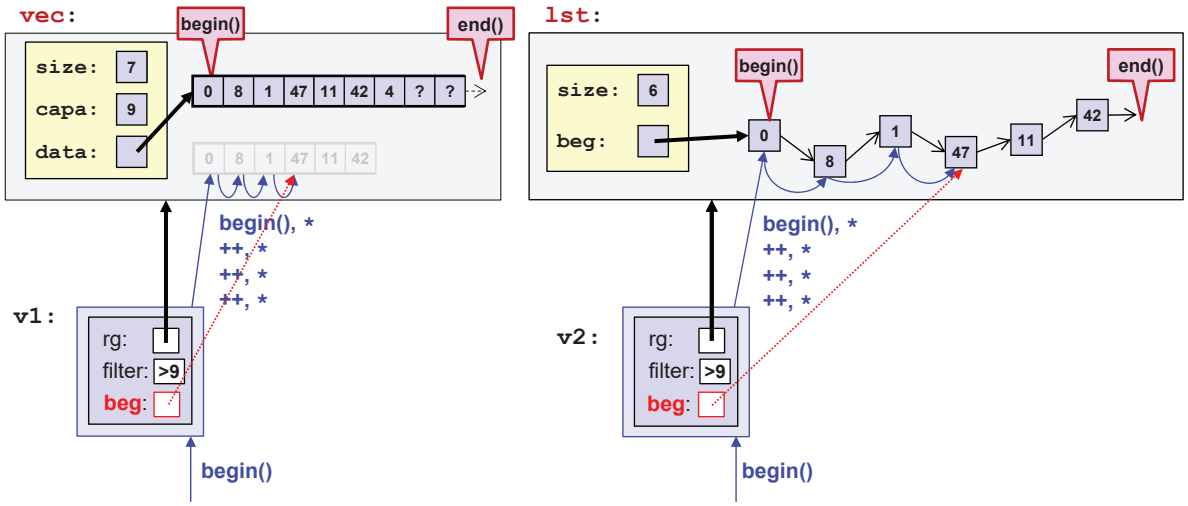
C++20




```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
vec.push_back(4);
std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```

Ways of Caching `begin()`

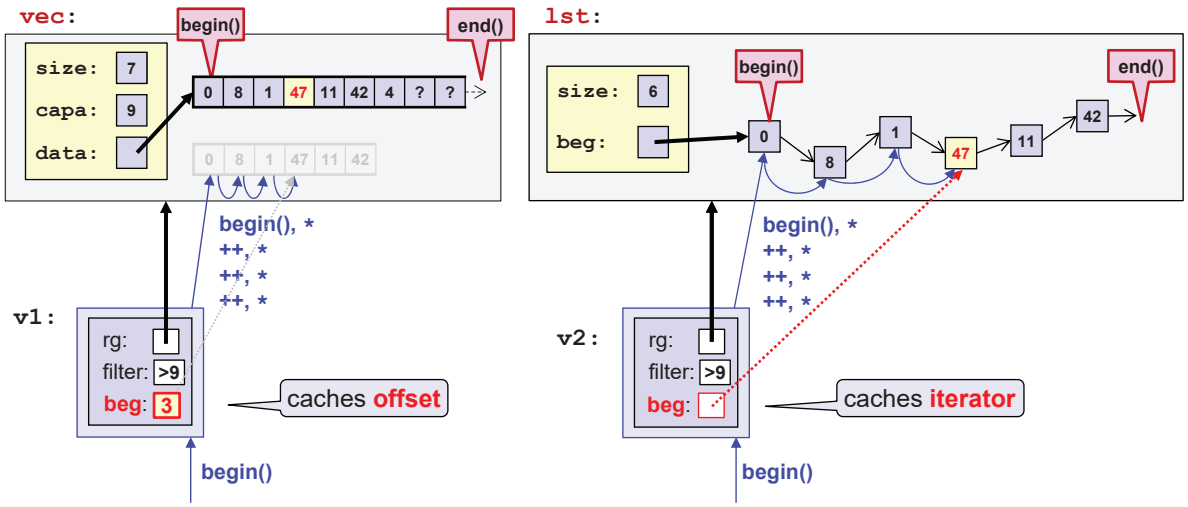
C++20




```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
vec.push_back(4);
std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```


Ways of Caching `begin()`

C++20

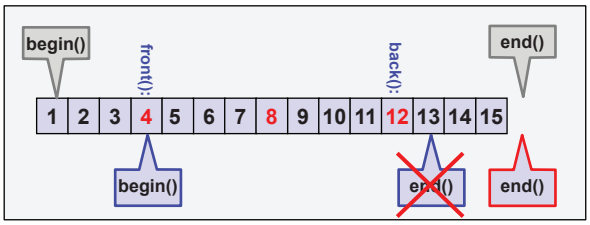



```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
vec.push_back(4);
std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```

There is support for modifying the underlying range after initializing the view

Performance of Filtering

C++20



- **Declaration / default initialization is cheap**
- **`begin()` can be expensive on the first call** (with **`empty()`**)
- **`end()` is cheap** (just the underlying end)
- **`empty()` can be expensive on the first call** (with **`begin()`**)
- ~~**`size()` is expensive** (iterate over all elements)~~
- ~~**`operator[]` is expensive** (iterate over all elements)~~

⇨ **Filter iterators do not support random access**

Guarantees for Ranges

C++98/C++20

- **Guarantees for Containers**
  - Declaration / default initialization has constant complexity
  - **`begin()`** has constant complexity
  - **`end()`** has constant complexity
- **Guarantees for Views**
  - Initialization has constant complexity
  - **`begin()`** has amortized constant complexity
  - **`end()`** has amortized constant complexity

C++20:

[range.range]:
Given an expression **`t`** such that **`decltype((t))`** is **`T&`**,
`T` **models concept std::range only if**
...
(3.2) — both **`ranges::begin(t)`** and **`ranges::end(t)`**
are **amortized constant time** and non-modifying,
...

C++20

Consequences for Read Iterations

Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
0 -8 -1 -47 -11 -42 -2
ERROR
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);

print(vec | std::views::take(3));                    // OK

print(vec | std::views::transform(std::negate{}));   // OK

auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                // Compile-time ERROR

for (int v : vec | std::views::filter(gt9)) {        // OK
  std::cout << v << ' ';
}
```

Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\
}

std::vector<int>
print(vec);

print(vec | std::

print(vec | std::

auto gt9 = [] (au
print(vec | std::

for (int v : vec
  std::cout << v
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
```

```
<source>: In instantiation of 'void print(const auto:10&)
[with auto:10 = std::ranges::filter_view<std::ranges::ref_view<std::vector<int> >,
      main()::<lambda(auto:11)> >]':

<source>:21:8:   required from here
   21 |   print(vec | std::views::filter(gt9));
      |   ~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

<source>:8:3: error:
passing 'const std::ranges::filter_view<std::ranges::ref_view<std::vector<int> >,
main()::<lambda(auto:11)> >' as 'this' argument discards qualifiers [-fpermissive]
    8 |   for (const auto& elem : coll) {
      |   ^~~

In file included from <source>:4:
/opt/compiler-explorer/gcc-trunk-20240916/include/c++/15.0.0/ranges:1794:7:
note:   in call to 'constexpr std::ranges::filter_view<_Vp, _Pred>::_Iterator
std::ranges::filter_view<_Vp, _Pred>::begin()
[with _Vp = std::ranges::ref_view<std::vector<int> >;
      _Pred = main()::<lambda(auto:11)>]'
 1794 |       begin()
      |       ^~~~~
```


Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);

print(vec | std::views::take(3));                          // OK

print(vec | std::views::transform(std::negate{}));         // OK

auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                      // Compile-time ERROR

for (int v : vec | std::views::filter(gt9)) {              // OK
  std::cout << v << ' ';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
0 -8 -1 -47 -11 -42 -2
ERROR
47 11 42
```

Processing Containers and Views by Universal Reference C++20

```
void print(auto&& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Universal (or *forwarding*) ***reference***
- Can universally refer to every expression (even temporaries/rvalues) without making it `const`

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);
print(vec | std::views::take(3));                          // OK
print(vec | std::views::transform(std::negate{}));         // OK
auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                      // OK
```

```
0 8 1
0 -8 -1 -47 -11 -42 -2
47 11 42
```


Concurrent Read Access to Views C++20

Use `const auto&` in this case

```
void process(auto&& coll) {
  // separate thread to read access the elements:
  std::jthread t1{[&] {
                    for (const auto& elem : coll) {   // DANGER: calls begin()
                      ...
                    }
                  }};

  ...       // so far the code is safe, but...

  if (!coll.empty()) {   // OOPS: concurrent call of begin()
    ...
  }
}
```

Concurrent read access is undefined behavior
- Concurrent iteration with `begin()` or `empty()` or `front()` ...

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
process(vec);                                              // OK
process(vec | std::views::take(3));                        // OK
process(vec | std::views::transform(std::negate{}));       // OK
auto gt9 = [] (auto val) { return val > 9; };
process(vec | std::views::filter(gt9));                    // Runtime ERROR
```

Forcing Caching

C++20

```
void process(auto&& coll) {
  (void)coll.empty();     // force caching (DO NOT REMOVE)
  // separate thread to read access the elements:
  std::jthread t1{[&] {
    for (const auto& elem : coll) {   // OK: begin() was called
      ...
    }
  }};

  ...       // so far the code is safe, but...

  if (!coll.empty()) {    // OK: begin() was called
    ...
  }
}
```

(void) to avoid that compilers warn about a useless call

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
process(vec);                                          // OK
process(vec | std::views::take(3));                    // OK
process(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
process(vec | std::views::filter(gt9));                // works but UB
```


Processing Containers and Views

C++20

```
void print(const auto& coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
0 -8 -1 -47 -11 -42 -2
ERROR
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);
print(vec | std::views::take(3));                    // OK
print(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                // Compile-time ERROR
auto fv{vec | std::views::filter(gt9)};
print(std::ranges::subrange{fv});                    // OK
```

Does ***not*** work in one statement

subrange is a view that stores/caches begin() on initialization

Processing Containers and Views by Value

C++20

```
void print(auto coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
0 -8 -1 -47 -11 -42 -2
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);                                          // expensive copy
print(vec | std::views::take(3));                    // OK
print(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                // OK
```


Processing Containers and Views by Value

C++20

```
void print(std::ranges::view auto coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

ERROR

```
0 8 1
0 -8 -1 -47 -11 -42 -2
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(vec);                                          // ERROR
print(vec | std::views::take(3));                    // OK
print(vec | std::views::transform(std::negate{}));   // OK
auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                // OK
```

Processing Containers and Views by Value

C++20

```
void print(std::ranges::view auto coll) {
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
0 8 1 47 11 42 2
0 8 1
0 -8 -1 -47 -11 -42 -2
47 11 42
```

```
std::vector<int> vec{0, 8, 1, 47, 11, 42, 2};
print(std::views::all(vec));                          // cheap
print(vec | std::views::take(3));                     // OK
print(vec | std::views::transform(std::negate{}));    // OK

auto gt9 = [] (auto val) { return val > 9; };
print(vec | std::views::filter(gt9));                 // OK
```


Overloading for Containers and Views

C++20

```
void print(std::ranges::view auto coll) { // for views only
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}

template<typename T>
void print(const T& coll)                 // not for views
requires (!std::ranges::view<T>)
{
  print(std::views::all(coll));
}
```

Necessary to avoid ambiguities when views are passed

```
std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
print(vec);                               // OK, calls 2nd print()
print(vec | std::views::filter(gt9));     // OK, calls 1st print()

print(getColl());                         // OK, calls 2nd print()
print(getColl() | std::views::filter(gt9)); // OK, calls 1st print()
```

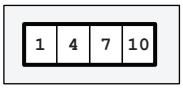
C++20

Consequences for Write Iterations

Modifying Filtered Elements

C++20

```
std::vector<int> coll{1, 4, 7, 10};
print(coll);

auto isEven = [] (auto&& i) { return i % 2 == 0; };
auto collEven = coll | std::views::filter(isEven);

// add 2 to even elements:
for (int& i : collEven) {
  i += 2;
}
print(coll);

// add 2 to even elements:
for (int& i : collEven) {
  i += 2;
}
print(coll);
```



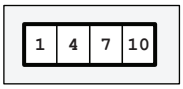
Output:

```
1 4 7 10
1 6 7 12
1 8 7 14
```

Modifying Filtered Elements

C++20

```
std::vector<int> coll{1, 4, 7, 10};
print(coll);

auto isEven = [] (auto&& i) { return i % 2 == 0; };
auto collEven = coll | std::views::filter(isEven);

// increment even elements:
for (int& i : collEven) {
  i += 1;          // Runtime Error: UB: predicate broken
}
print(coll);

// increment even elements:
for (int& i : collEven) {
  i += 1;          // Runtime Error: UB: predicate broken
}
print(coll);
```



Output:

```
1 4 7 10
1 5 7 11
1 6 7 11
```


Modifying Filtered Elements

C++20

- **Key use case of a filter:**
  - Fix broken elements

has undefined behavior:

[range.filter.iterator]:
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.

Modifying Filtered Elements

C++20

- **Key use case of a filter:**
  - Fix broken elements

has undefined behavior:

[range.filter.iterator]:
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.

Modifying Filtered Elements

C++20

- **Key use case of a filter:**
  - Fix broken elements

has undefined behavior:

[range.filter.iterator]:
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.

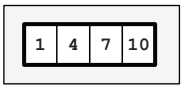
```
// as a shaman:
for (auto& m : monsters | std::views::filter(isDead)) {
  m.resurrect();   // undefined behavior: because no longer dead
  m.burn();        // OK (because it is still dead)
}
```

Thanks to Patrice Roy for this example

Modifying Filtered Elements

C++20

```
std::vector<int> coll{1, 4, 7, 10};
print(coll);

auto isEven = [] (auto&& i) { return i % 2 == 0; };
```




```
// add 2 to even elements:
for (int& i : coll | std::views::filter(isEven)) {
  i += 2;          // UB: but works
}
print(coll);

// add 2 to even elements:
for (int& i : coll | std::views::filter(isEven)) {
  i += 2;          // UB: but works
}
print(coll);
```

Output:

```
1 4 7 10
1 5 7 11
1 5 7 11
```

Use (and reuse) views ad hoc

Defining Views Ahead

C++20

```
std::vector<int> coll{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
print(coll);

auto isEven = [] (auto&& i) { return i % 2 == 0; };
auto first3Even = std::views::filter(isEven) | std::views::take(3);
```

Define a **view/pipeline without** the **underlying range**

```
// add 2 to even elements:
for (int& i : coll | first3Even) {
  i += 2;          // UB: but works
}
print(coll);

// add 2 to even elements:
for (int& i : coll | first3Even) {
  i += 2;          // UB: but works
}
print(coll);
```

Output:

```
1 2 3 4 5 6 7 8 9 10
1 4 3 6 5 8 7 8 9 10
1 6 3 8 5 10 7 8 9 10
```

Apply views and pipelines ad hoc

Passing Views

C++20

```
void add(auto& rg,
         const auto& val)
{
  for (auto& i : rg) {
    i += val;
  }
}
```

Output:

```
values: 10 8 5 15 0
values: 10ms 8ms 5ms 15ms 0ms
values: 10msms 8ms 5ms 15ms 0ms
```

```
std::vector<std::string> coll{"values:", "10", "8", "5", "15", "0"};
print(coll);

auto noSuffix = [] (auto& s) { return !s.ends_with("ms"); };
auto vEven = coll | std::views::drop(1)| std::views::filter(noSuffix);

add(vEven, "ms");
print(coll);

add(vEven, "ms");
print(coll);
```

Undefined behavior and fails

Passing Views

C++20

```
void add(auto& rg, const auto& pipe,
         const auto& val)
{
  for (auto& i : rg | pipe) {
    i += val;
  }
}
```

Output:

```
values: 10 8 5 15 0
values: 10ms 8ms 5ms 15ms 0ms
values: 10ms 8ms 5ms 15ms 0ms
```

```
std::vector<std::string> coll{"values:", "10", "8", "5", "15", "0"};
print(coll);

auto noSuffix = [] (auto& s) { return !s.ends_with("ms"); };
auto pEven = std::views::drop(1) | std::views::filter(noSuffix);

add(coll, pEven, "ms");
print(coll);

add(coll, pEven, "ms");
print(coll);
```

Undefined behavior but works

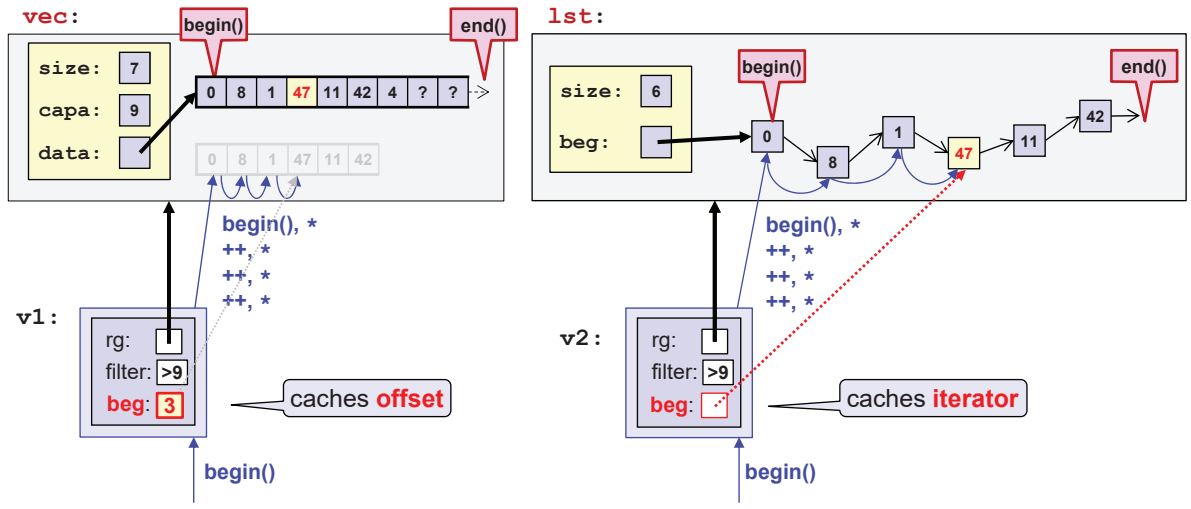
Pass views/pipelines without the underlying range

C++20

Consequences for
Modifications between Iterations

Ways of Caching `begin()`

C++20



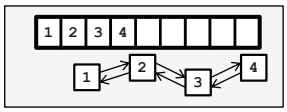

```
std::vector<int> vec{0, 8, 1, 47, 11, 42};
auto v1 = vec | std::views::filter(gt9);
auto pos = v1.begin();
vec.push_back(4);
std::list<int> lst{0, 8, 1, 47, 11, 42};
auto v2 = lst | std::views::filter(gt9);
auto pos = v2.begin();
```

There is **support** for **modifying** the **underlying range** after initializing the view

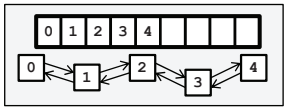
Filter Views on Modified Ranges

C++20

```
std::vector vec{1, 2, 3, 4};
vec.reserve(9);
std::list lst{1, 2, 3, 4};
auto vVec = vec | std::views::filter(even);
auto vLst = lst | std::views::filter(even);
```

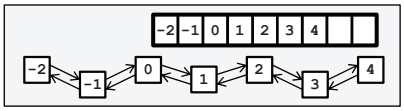



```
// insert new elements at the front:
vec.insert(vec.begin(), 0);
lst.insert(lst.begin(), 0);
print(vVec, vLst);
```




```
0 2 4
0 2 4
```

```
vec.insert(vec.begin(), {-2, -1});
lst.insert(lst.begin(), {-2, -1});
print(vVec, vLst);
```

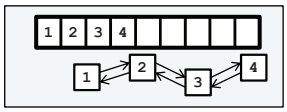



```
-2 0 2 4
0 2 4
```


Filter Views on Modified Ranges

C++20

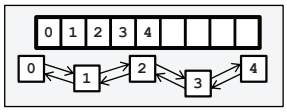
```
std::vector vec{1, 2, 3, 4};
vec.reserve(9);
std::list lst{1, 2, 3, 4};
auto vVec = vec | std::views::filter(even);
auto vLst = lst | std::views::filter(even);
print(vVec, vLst);
```




```
2 4
2 4
```

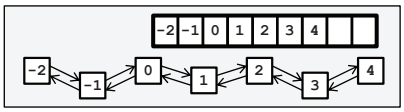
`print()` or `empty()` or other reads **change state**

```
// insert new elements at the front:
vec.insert(vec.begin(), 0);
lst.insert(lst.begin(), 0);
print(vVec, vLst);
```




```
1 2 4
2 4
```

```
vec.insert(vec.begin(), {-2, -1});
lst.insert(lst.begin(), {-2, -1});
print(vVec, vLst);
```

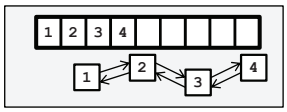



```
-1 0 2 4
2 4
```

Copying Filter Views

C++20

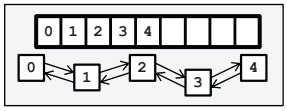
```
std::vector vec{1, 2, 3, 4};
vec.reserve(9);
std::list lst{1, 2, 3, 4};
auto vVec = vec | std::views::filter(even);
auto vLst = lst | std::views::filter(even);
print(vVec, vLst);
```




```
2 4
2 4
```

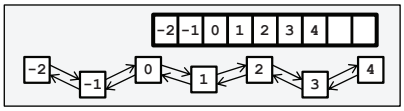
print() or empty() or other reads **change state**

```
// insert new elements at the front:
vec.insert(vec.begin(), 0);
lst.insert(lst.begin(), 0);
print(vVec, vLst);
```




```
1 2 4
2 4
```

```
vec.insert(vec.begin(), {-2, -1});
lst.insert(lst.begin(), {-2, -1});
print(vVec, vLst);
```




```
-1 0 2 4
2 4
```

```
// copy the views:
auto vVecB = vVec;
auto vLstB = vLst;
print(vVecB, vLstB);
```

Copying the view ***might*** uncache

```
-1 0 2 4
-2 0 2 4
```


Filter Views and Call-by-Value

C++20

```
void printByVal(auto coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
void printByRef(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::set coll{1, 2, 3, 4};
auto collEven = coll | std::views::filter(even);

  coll.insert(0);

printByVal(collEven);
printByRef(collEven);
```

Output:

```
0 2 4
0 2 4
```

Filter Views and Call-by-Value

C++20

```
void printByVal(auto coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
void printByRef(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::set coll{1, 2, 3, 4};
auto collEven = coll | std::views::filter(even);
if (!collEven.empty()) {
  coll.insert(0);
}
printByVal(collEven);
printByRef(collEven);
```

Output:

```
0 2 4
2 4
```

Different behavior
on **call-by-value**
and **call-by-reference**

C++20

Summary

Basic Container Idioms Broken by Filter Views

C++98/C++20

- You can iterate if the range is `const` — Broken for filters
- A read iteration does not change state — Broken for filters
- `empty()` doesn't have side effects — Broken for filters
- Concurrent read iterations are safe — Broken for filters
- A copy of a range has the same state — Broken for filters
- Modifications between iterations are safe — Broken for filters
- Modifications via iterations are safe — Broken for filters

How to Use the Filter View

C++20

- Put filters early in a pipeline
- Apply filter views ad-hoc
  - Pass views/pipelines without the underlying range
- Do not modify elements via a filter
  - or if you modify
    - do not break the predicate
    - or iterate only once from begin to end (like an input iterator)

You cannot use a filter view to "heal" broken elements

works but formally UB

- Do not modify underlying ranges after applying a filter
  - or if you modify
    - No `empty()`, `front()`, or read iteration before the modification
- Do not use filters in concurrent code
  - or no concurrent iteration, `empty()`, `begin()`, `front()`, `if`
- Prefer `empty()` over `size()==0`

Design Alternatives for Filter Views

- **`begin()` is cached** C++20
  - Compile-time errors: not usable if `const`
  - Runtime errors: reading is not stateless, healing broken elements is UB, trivial modifications cause UB, concurrent reads cause UB
- **`begin()` is initialized during construction**
  - Performance issue: Initialization should have constant complexity
- **`begin()` is thread safe** (using `mutable`)
  - Performance issue: Makes `begin()` very expensive
- **No caching at all**
  - Performance issue: Some use cases have quadratic complexity
    - Reverse view should cache instead
    - Programmers can and have to use workaround (subrange or cacheBeg | filter)
- **No caching and filter iterators become input iterators**
  - Disables algorithms with multiple or reverse iterations
  - Some non-trivial use cases like with reverse no longer compile

Design Alternatives for Filter Views

- **`begin()` is cached** C++20
  - Severe compile-time errors: unusable with `const`
  - Runtime errors: reading is
    trivial mod
- **`begin()` is initialized du**
  - Performance issue: Initializ
- **`begin()` is thread safe** (
  - Performance issue: Makes
- **No caching at all**
  - Performance issue: Some
    - Reverse view should cache ins
    - Programmers can and have to use round (subrange or cacheBeg | filter)
- **No caching and filter iterators become input iterators** C++??
  - Disables algorithms with multiple or reverse iterations
  - Some non-trivial use cases like with reverse no longer compile

Possible path to go ?

- **All described issues are gone**
  - Can read iterate
  - Read is thread safe
  - `empty()` is stateless
  - Can write via iteration
  - Can write between iterations
- **What compiles is backward compatible**
  - Bad performance does not compile

Thank You!

Nicolai M. Josuttis

www.josuttis.com
nico@josuttis.com
@NicoJosuttis