C++ Views

Nicolai M. Josuttis

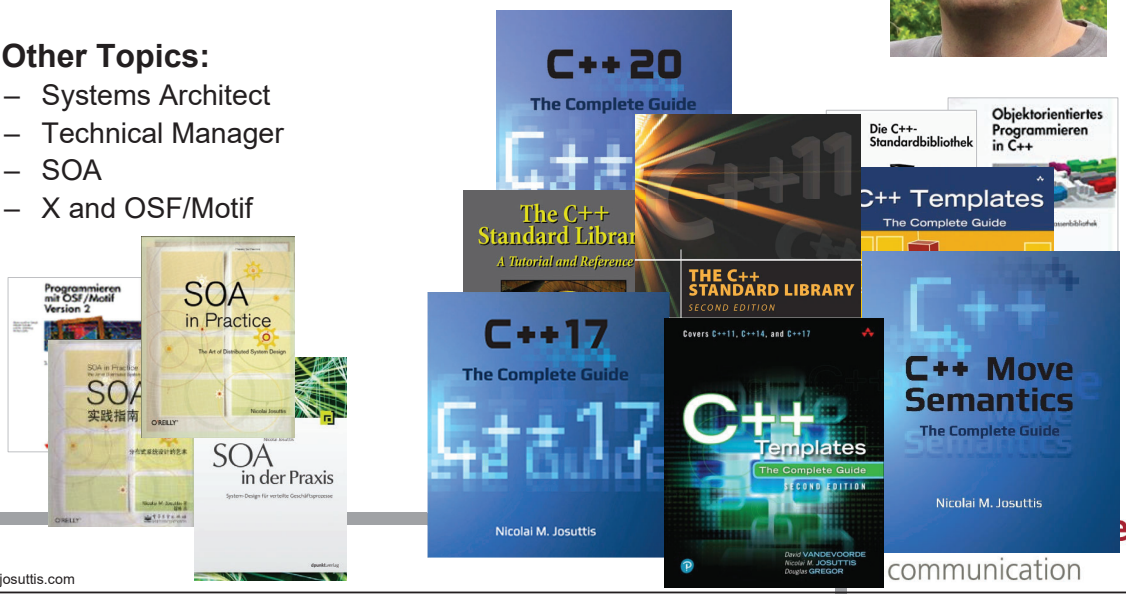
josuttis.com

@NicoJosuttis

04/23

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++20

Views

Generic Print Function

C++11

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

std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
```

Output:

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

Generic Print Function

C++17

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

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
```

Class Template Argument Deduction deduces int as element type

Output:

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


Using Abbreviated Function Template Syntax

C++20

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

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
```

Output:

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

Using Concepts

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> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
```

Output:

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


Using 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> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
print(std::views::take(vec, 3));       // print first three elements
print(std::views::take(lst, 3));       // print first three elements
print(vec | std::views::take(3));      // print first three elements
print(lst | std::views::take(3));      // print first three elements

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

Output:

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

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 Multiple Views

C++20

```
// view 4th to 11th value that are multiples of 3 with suffix "s":
auto v = std::views::iota(1, 100)                       // from 1 to 99
         | 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 Zip Views

C++23

```
#include <iostream>
#include <string>
#include <vector>
#include <ranges>

int main()
{
  std::vector<std::string> coll{"one", "two", "three"};

  for (auto elem : coll) {
    std::cout << elem << '\n';
  }

  for (auto [idx, elem] : std::views::zip(std::views::iota(1), coll)) {
    std::cout << idx << ": " << elem << '\n';
  }
}
```

Output:

```
one
two
three
1: one
2: two
3: three
```


C++20: How Views Operate

C++20

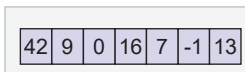
```
void print(const auto& rg)
{
  for (const auto& elem : rg) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}
```

```
std::vector coll{42, 9, 0, 16, 7, -1, 13};
print(coll);

print(coll | std::views::take(3)
           | std::views::transform(std::negate{}));
```

Output:

```
42 9 0 16 7 -1 13
-42 -9 0
```

coll:



C++20: How Views Operate

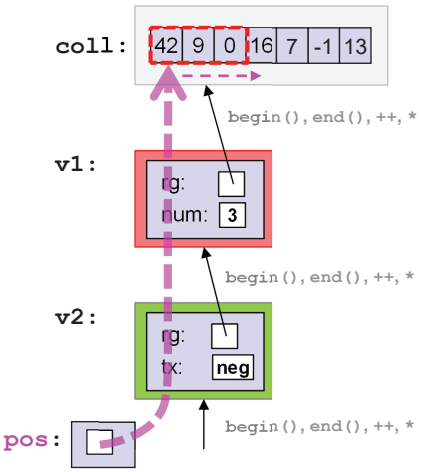
C++20

```
void print(const auto& rg)
{
  for (auto pos = std::ranges::begin(rg); pos != std::ranges::end(rg); ++pos) {
    std::cout << *pos << ' ';
  }
  std::cout << '\n';
}
```

Output:

```
42 9 0 16 7 -1 13
-42 -9 0
*pos: -42
*pos: -9
```

```
std::vector coll{42, 9, 0, 16, 7, -1, 13};
print(coll);

print(coll | std::views::take(3)
           | std::views::transform(std::negate{}));

auto v1 = std::views::take(coll, 3);
auto v2 = std::views::transform(v1, std::negate{});

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



C++20

API of Views

Processing Containers and Views

C++20

```
void print(const auto& coll)
{
  std::cout << coll.size() << " elems\n";
}
```

Output:

```
6 elems
6 elems
3 elems
3 elems
3 elems
3 elems
ERROR
```

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print all but first three elements
print(lst | std::views::drop(3));          // print all but first three elements
print(vec | std::views::filter([](auto elem){ // ERROR
                                 return elem % 3 == 0;
                               }));
```


Processing Containers and Views

C++20

```
void print(const auto& coll)
{
  std::cout << coll.size() << " elems: ";

  std::cout << coll.front() << " ...\n"
}
```

Output:

```
6 elems: 0 ...
6 elems: 0 ...
3 elems: 0 ...
3 elems: 0 ...
3 elems: 47 ...
ERROR
ERROR
```

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print all but first three elements
print(lst | std::views::drop(3));          // ERROR
print(vec | std::views::filter([](auto elem){ // ERROR
                                 return elem % 3 == 0;
                               }));
```

Processing Containers and Views

C++20

```
void print(const auto& coll)
{
  std::cout << coll.size() << " elems: ";

  std::cout << coll.front() << " ... "
            << coll.back() << '\n';
}
```

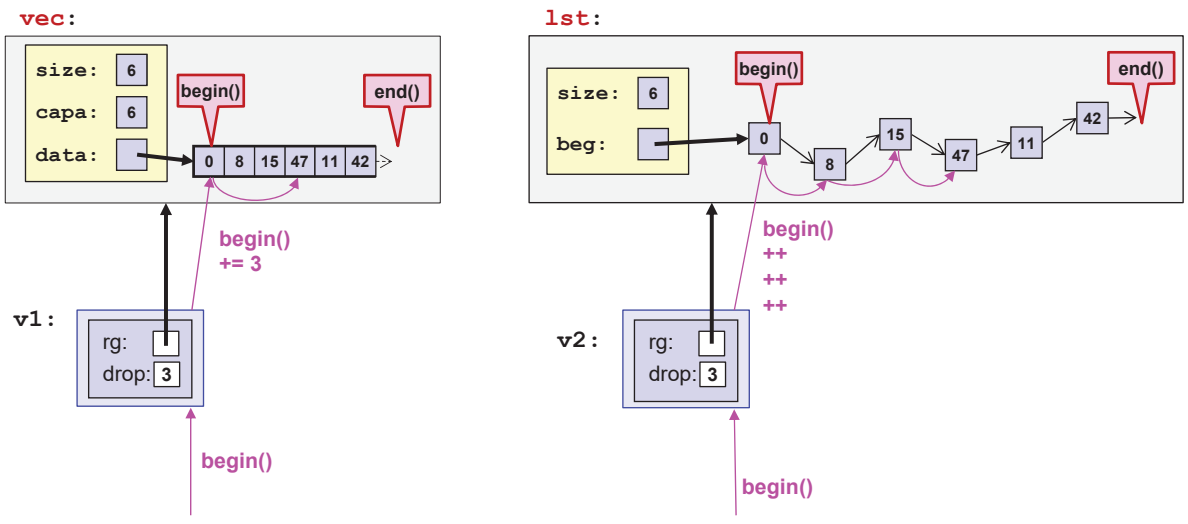
Output:

```
6 elems: 0 ... 42
6 elems: 0 ... 42
3 elems: 0 ... 15
ERROR
3 elems: 47 ... 42
ERROR
ERROR
```

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // ERROR
print(vec | std::views::drop(3));          // print all but first three elements
print(lst | std::views::drop(3));          // ERROR
print(vec | std::views::filter([](auto elem){ // ERROR
                                 return elem % 3 == 0;
                               }));
```


How Expensive is `begin()` ?

C++20



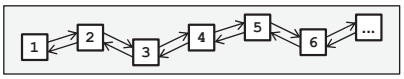

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
auto v1 = vec | std::views::drop(3);
auto pos = v1.begin();

std::list<int> lst, 8, 15, 47, 11, 42};
auto v2 = lst | std::views::drop(3);
auto pos = v2.begin();
```

Member Functions of Views

C++20

- **Views do not provide expensive member functions**
  - Unless they can't be used otherwise


```
std::vector vec{1, 2, 3, 4, 5, ...};
auto vVec = vec | std::views::drop(n);
vVec.begin()       // fast: vec.begin() + n
vVec.empty()       // fast: vec.size() <= n
vVec.size()        // fast: n >= vec.size() ? 0 : vec.size() - n
vVec[idx]          // fast: vec[idx + n]
```




```
std::list lst{1, 2, 3, 4, 5, ...};
auto vLst = lst | std::views::drop(n);
vLst.begin()       // slow: lst.begin() and n times ++
vLst.empty()       // fast:  lst.size() <= n
vLst.size()        // fast:  n >= lst.size() ? 0 : lst.size() - n
vLst[idx]          // slow: lst.begin() and n + idx times ++

auto vFlt = coll | std::views::filter(pred);
vFlt.begin()       // slow: pred for all elements until first true
vFlt.empty()       // slow: pred for all elements until first true
vFlt.size()        // slow: pred for all elements
vFlt[idx]          // slow: pred for all elements until idx times true
```


Basic API of Views

C++20

- **The basic API of views depends on constraints**

Member function	Meaning	Constraints
`begin()`	Begin iterator	
`end()`	Sentinel (end iterator)	
`empty()`	is empty?	forward range
conversion to `bool`	has elements?	`std::ranges::empty()` valid
`size()`	number of elements	forward range and cheap
`front()`	first element	forward range
`back()`	last element	bidirectional and common range
`[idx]`	*idx*-th element	random-access range
`data()`	raw pointer to elements	contiguous range

- **No `cbegin()`, `cend()`, `cdata()`** (yet)

C++20: Basic View Interface

C++20

```
namespace std::ranges {
template<typename D>                                  // for each derived view type D
requires is_class_v<D> && same_as<D, remove_cv_t<D>>
class view_interface {
  ...
 public:
  // (not) empty? :
  constexpr bool empty() requires forward_range<D>;
  constexpr explicit operator bool() requires requires {
                                                ranges::empty(static_cast<D&>(*this));
                                               };
  // number of elements:
  constexpr auto size() requires forward_range<D> &&
                                 sized_sentinel_for<sentinel_t<D>, iterator_t<D>>;
  // element access:
  constexpr decltype(auto) front() requires forward_range<D>;
  constexpr decltype(auto) back() requires bidirectional_range<D> && common_range<D>;
  template<random_access_range R = D>
    constexpr decltype(auto) operator[](range_difference_t<R> n);
  constexpr auto data() requires contiguous_iterator<iterator_t<D>> {
    return to_address(ranges::begin(ranges::empty(static_cast<D&>(*this))));
  }
  ...    // same as const member for const D
};
}
```

View types add `begin()`, `end()`, and specific member functions

P2278 will add for C++23: `cbegin()` and `cend()`

C++20

Issues of Views

Processing Containers and Views

C++20

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

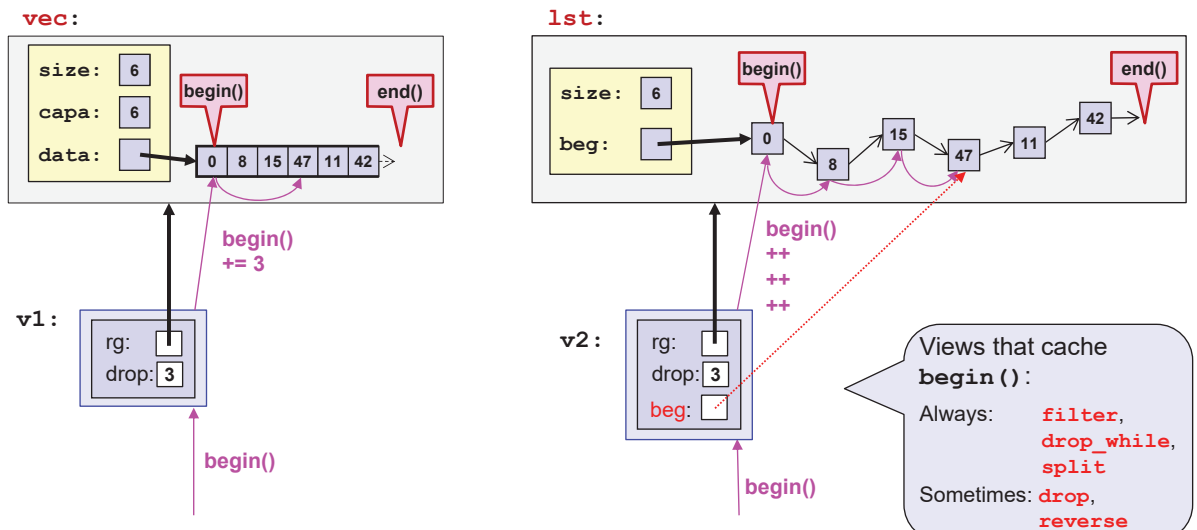
```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print fourth to last element
print(lst | std::views::drop(3));          // Compile-time ERROR
for (int v : lst | std::views::drop(3)) {  // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

```
0 8 15 47 11 42
0 8 15 47 11 42
0 8 15
0 8 15
47 11 42
ERROR
47 11 42
```


Views that Cache `begin()`

C++20




```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
auto v1 = vec | std::views::drop(3);
auto pos = v1.begin();

std::list<int> lst{0, 8, 15, 47, 11, 42};
auto v2 = lst | std::views::drop(3);
auto pos = v2.begin();
```

Member Functions of Views

C++20

	1st begin()	2nd begin()	size()	1st empty()	2nd empty()
std::vector vec	constant	constant	constant	constant	constant
std::list lst	constant	constant	constant	constant	constant
vec | drop(*n*)	constant	constant	constant	constant	constant
lst | drop(*n*)	linear	constant	constant	constant	constant
vec | filter(...)	linear	constant	--	linear	constant
lst | filter(...)	linear	constant	--	linear	constant
vec | filter(...) | drop(*n*)	linear	constant	--	linear	constant
lst | filter(...) | drop(*n*)	linear	constant	--	linear	constant

"First linear, then constant" is called "*amortized constant*" in the C++ standard

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

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, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print fourth to last element
print(lst | std::views::drop(3));          // Compile-time ERROR
for (int v : lst | std::views::drop(3)) {  // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

```
0 8 15 47 11 42
0 8 15 47 11 42
0 8 15
0 8 15
47 11 42
ERROR
47 11 42
```

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, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print fourth to last element
print(lst | std::views::drop(3));          // Compile-time ERROR
auto v = lst | std::views::drop(3);
print(std::ranges::subrange{v.begin(), v.end()});   // OK
```

Output:

```
0 8 15 47 11 42
0 8 15 47 11 42
0 8 15
0 8 15
47 11 42
ERROR
47 11 42
```


Passing Containers and Views by Universal Reference

C++20

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

std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);
print(lst);
print(vec | std::views::take(3));          // print first three elements
print(lst | std::views::take(3));          // print first three elements
print(vec | std::views::drop(3));          // print fourth to last element
print(lst | std::views::drop(3));          // OK: print fourth to last element
for (int v : lst | std::views::drop(3)) {  // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Universal (or *forwarding*) ***reference***

- Can universally refer to every expression (even temporaries/rvalues) without making it `const`

Output:

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

Concurrent Iterations over Views

C++20

```
auto printAndSum(auto&& rg) {
  // one thread prints the elements:
  std::jthread printThread{[&] {
    for (const auto& elem : rg) {
      std::cout << elem << ' ';
    }
    std::cout << '\n';
  }};

  // this thread computes the sum of the element values:
  return std::accumulate(rg.begin(), rg.end(),
                         0L);
}
```

Use `const auto&` in this case

Concurrent read iterations
cause undefined behavior
- Concurrent `begin()` and `end()` only safe for containers

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

auto sum1 = printAndSum(vec);                          // OK
auto sum2 = printAndSum(lst);                          // OK

auto sum3 = printAndSum(vec | std::views::drop(2));    // OK
auto sum4 = printAndSum(lst | std::views::drop(2));    // Runtime ERROR
```


Passing Containers and Views by Value

C++20

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

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);                                  // expensive
print(lst);                                  // expensive
print(vec | std::views::take(3));            // print first three elements
print(lst | std::views::take(3));            // print first three elements
print(vec | std::views::drop(3));            // print fourth to last element
print(lst | std::views::drop(3));            // OK: print fourth to last element
for (int v : lst | std::views::drop(3)) {    // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

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

Accepting Only Views by Value

C++20

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

std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(vec);                                  // ERROR
print(lst);                                  // ERROR
print(vec | std::views::take(3));            // print first three elements
print(lst | std::views::take(3));            // print first three elements
print(vec | std::views::drop(3));            // print fourth to last element
print(lst | std::views::drop(3));            // OK: print fourth to last element
for (int v : lst | std::views::drop(3)) {    // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

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


Accepting Only Views by Value

C++20

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

std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};

print(std::views::all(vec));                 // cheap
print(std::views::all(lst));                 // cheap
print(vec | std::views::take(3));            // print first three elements
print(lst | std::views::take(3));            // print first three elements
print(vec | std::views::drop(3));            // print fourth to last element
print(lst | std::views::drop(3));            // OK: print fourth to last element
for (int v : lst | std::views::drop(3)) {    // OK: print fourth to last element
   std::cout << v << ' ';
}
```

Output:

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

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::take(3));          // OK, calls 1st print()

print(getColl());                          // OK, calls 2nd print()
print(getColl() | std::views::take(3));    // OK, calls 1st print()
```

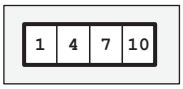

C++20

Modifying View Elements

Using the Filter View

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);
```



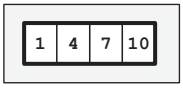
Output:

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


Using the Filter View

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
```

Using the Filter View

- **Main use case of a filter:**
  - Fix an attribute that some elements might have

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.

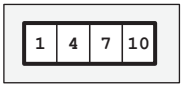
```
// as a shaman:
for (auto& m : monsters | std::views::filter(isDead)) {
  m.resurrect();   // undefined behavior: because no longer dead
  m.burn();        // OK (because it is still dead)
}
```

Thanks to Patrice Roy for this example

Using the Filter View

C++20

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

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




```
// increment even elements:
for (int& i : coll | std::views::filter(isEven)) {
  i += 1;          // UB: but works
}
print(coll);

// increment even elements:
for (int& i : coll | std::views::filter(isEven)) {
  i += 1;          // UB: but works
}
print(coll);
```

Output:

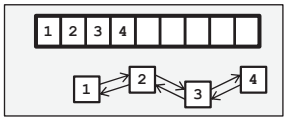
```
1 4 7 10
1 5 7 11
1 5 7 11
```

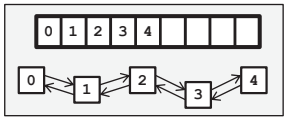
Use (and reuse) views ad hoc

Modifications Considered Harmful with Caching

C++20

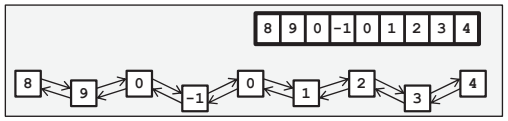
```
std::vector vec{1, 2, 3, 4};
vec.reserve(9);
std::list lst{1, 2, 3, 4};
auto vVec = vec | std::views::drop(2);
auto vLst = lst | std::views::drop(2);
```






```
2 3 4
2 3 4
```

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




```
vec.insert(vec.begin(), {8, 9, 0, -1});
lst.insert(lst.begin(), {8, 9, 0, -1});
print(vVec, vLst);
```

```
0 -1 0 1 2 3 4
2 3 4
```

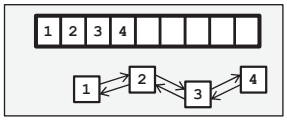
```
auto vVecB = vVec;
auto vLstB = vLst;
print(vVecB, vLstB);
```

```
0 -1 0 1 2 3 4
0 -1 0 1 2 3 4
```

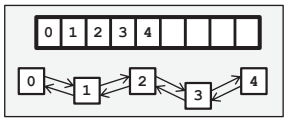

Modifications Considered Harmful with Caching

C++20

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

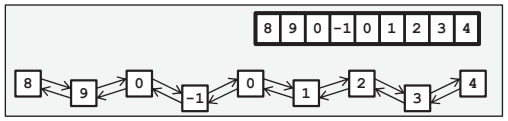



```
3 4
3 4
```




```
2 3 4
3 4
```

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




```
vec.insert(vec.begin(), {8, 9, 0, -1});
lst.insert(lst.begin(), {8, 9, 0, -1});
print(vVec, vLst);
```

```
0 -1 0 1 2 3 4
3 4
```

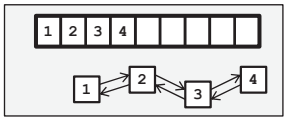
```
auto vVecB = vVec;
auto vLstB = vLst;
print(vVecB, vLstB);
```

```
0 -1 0 1 2 3 4
0 -1 0 1 2 3 4
```

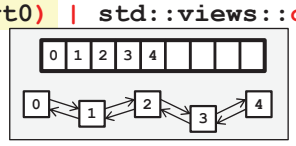
Modifications Considered Harmful with Caching

C++20

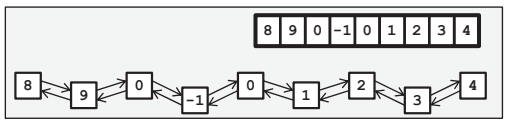
```
std::vector vec{1, 2, 3, 4};
vec.reserve(9);
std::list lst{1, 2, 3, 4};
auto gt0 = [](const auto& v){return v > 0;};
auto vVec = vec | std::views::filter(gt0) | std::views::drop(2);
auto vLst = lst | std::views::filter(gt0) | std::views::drop(2);
print(vVec, vLst);

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

vec.insert(vec.begin(), {8, 9, 0, -1});
lst.insert(lst.begin(), {8, 9, 0, -1});
print(vVec, vLst);

auto vVecB = vVec;
auto vLstB = vLst;
print(vVecB, vLstB);
```




```
3 4
3 4
```




```
2 3 4
3 4
```



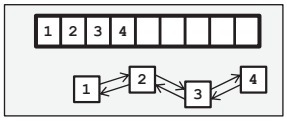

```
0 1 2 3 4
3 4
```

```
1 2 3 4
1 2 3 4
```

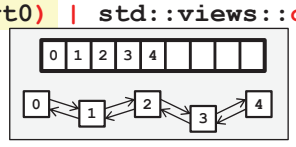

Modifications Considered Harmful with Caching

C++20

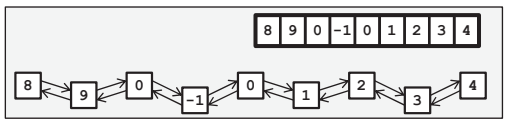
```
std::vector vec{1, 2, 3, 4};
vec.reserve(9);
std::list lst{1, 2, 3, 4};
auto gt0 = [](const auto& v){return v > 0;};
auto vVec = vec | std::views::filter(gt0) | std::views::drop(2);
auto vLst = lst | std::views::filter(gt0) | std::views::drop(2);
print(vVec, vLst);

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

vec.insert(vec.begin(), {8, 9, 0, -1});
lst.insert(lst.begin(), {8, 9, 0, -1});
print(vVec, vLst);

auto vVecB = vVec;
auto vLstB = vLst;
print(vVecB, vLstB);
```




```
3 4
3 4
```



undefined behavior
3 4



undefined behavior
3 4

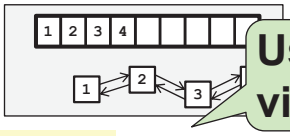
Usually: 1 2 3 4

undefined behavior
1 2 3 4

Modifications Considered Harmful with Caching

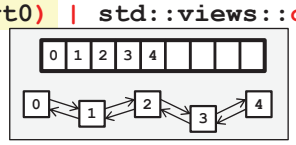
C++20

```
std::vector vec{1, 2, 3, 4};
vec.reserve(9);
std::list lst{1, 2, 3, 4};
auto gt0 = [](const auto& v){return v > 0;};
auto vVec = vec | std::views::filter(gt0) | std::views::drop(2);
auto vLst = lst | std::views::filter(gt0) | std::views::drop(2);
print(vVec, vLst);

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

vec.insert(vec.begin(), {8, 9, 0, -1});
lst.insert(lst.begin(), {8, 9, 0, -1});
print(vVec, vLst);

auto vVecB = vVec;
auto vLstB = vLst;
print(vVecB, vLstB);
```

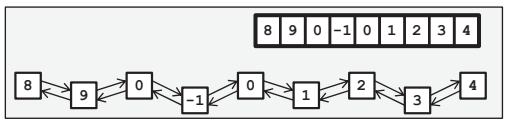


Use (and reuse) views ad hoc

3 4



undefined behavior
3 4



undefined behavior
3 4

Usually: 1 2 3 4

undefined behavior
1 2 3 4

C++20

Views and `const`

cbegin()

C++20

```
template<typename T>
void print(T&& coll) {
  auto pos = coll.cbegin();              // OOPS: not available for views
  std::cout << *pos;

}
```

C++20

```
std::vector vec{1, 2, 3, 4, 5};
print(vec);                              // OK
print(vec | std::views::drop(3));        // ERROR
print(vec | std::views::filter(...));    // ERROR

std::list lst{1, 2, 3, 4, 5};
print(lst | std::views::drop(3));        // ERROR
```


cbegin()

C++20

```
template<typename T>
void print(T&& coll) {
  auto pos = coll.cbegin();
  *std::cbegin(coll) = 0;                // OOPS: modifies first elem of views
  *std::ranges::cbegin(coll) = 0;        // OOPS: modifies first elem of views
}
```

C++20

```
std::vector vec{1, 2, 3, 4, 5};
print(vec);                              // Compile-time ERROR
print(vec | std::views::drop(3));        // OOPS: compiles and modifies coll
print(vec | std::views::filter(...));    // OOPS: compiles and modifies coll

std::list lst{1, 2, 3, 4, 5};
print(lst | std::views::drop(3));        // OOPS: compiles and modifies coll
```

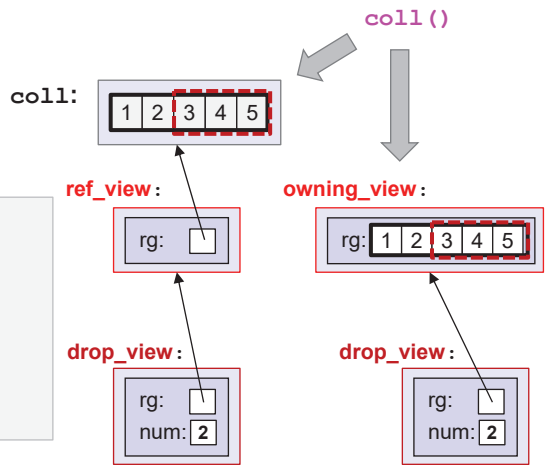
Exact Type of Views to Containers

C++20

- std::ranges::ref_view for named objects (lvalues) const
- std::ranges::owning_view for temporaries (rvalues)
  - Added later to C++20 as fix with http://wg21.link/P2415

```
std::vector<int> coll()
{
  return {1, 2, 3, 4, 5};
}
```

```
std::vector<int> c = coll();
auto v1 = c | std::views::drop(2);
        drop_view<ref_view<vector<int>>>

auto v2 = coll() | std::views::drop(2);
        drop_view<owning_view<vector<int>>>
```



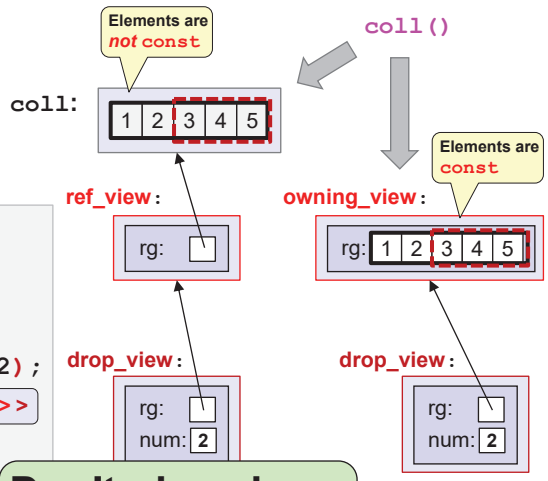
const Propagation of Views

C++20

- std::ranges::ref_view does *not* propagate const
- std::ranges::owning_view propagates const

```
std::vector<int> coll()
{
  return {1, 2, 3, 4, 5};
}
```

```
std::vector<int> c = coll();
const auto v1 = c | std::views::drop(2);
              drop_view<ref_view<vector<int>>>

const auto v2 = coll() | std::views::drop(2);
              drop_view<owning_view<vector<int>>>

v1[0] = 42;   // OK (bad)
v2[0] = 42;   // ERROR (good)
```



Don't give views a name

cbegin() C++23

Don't use `std::cbegin()` since C++20
- Prefer `std::ranges::` over `std::` C++23

```
template<typename T>
void print(T&& coll) {
  auto pos = coll.cbegin();          // OK: available since C++23
  *std::cbegin(coll) = 0;            // OOPS: still broken
  *std::ranges::cbegin(coll) = 0;    // ERROR
}
```

```
std::vector vec{1, 2, 3, 4, 5};
print(vec);                          // ERROR
print(vec | std::views::drop(3));    // ERROR(unless only std::begin() used)
print(vec | std::views::filter(...)); // ERROR (unless only std::begin() used)
std::list lst{1, 2, 3, 4, 5};
print(lst | std::views::drop(3));    // ERROR (unless only std::begin() used)
```


Declare Elements `const` C++20

```
void print(auto&& coll)
{
  for (const auto& elem : coll) {
    std::cout << elem << ' ';    // can't modify elem here
  }
  std::cout << '\n';
}
```

zip_view

C++23

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

```
std::vector v1{1, 2, 3};
std::vector v2{10, 20, 30};
printPairs(std::views::zip(v1, v2));
```

Output:

```
1:10
2:20
3:30
```


zip_view

C++23

```
void printPairs(const auto& coll)
{
  for (const auto& elem : coll) {
    if (elem.first == 2) {
      std::cout << "* ";
    }
    std::cout << elem.first << ':' << elem.second << '\n';
  }
}
```

```
std::vector v1{1, 2, 3};
std::vector v2{10, 20, 30};
printPairs(std::views::zip(v1, v2));
```

Output:

```
1:10
* 2:20
3:30
```

zip_view

C++23

```
void printPairs(const auto& coll)
{
  for (const auto& elem : coll) {
    if (elem.first = 2) {
      std::cout << "* ";
    }
    std::cout << elem.first << ':' << elem.second << '\n';
  }
}
```

Some views ignore this **const**

Some views ignore this **const**

```
std::vector v1{1, 2, 3};
std::vector v2{10, 20, 30};
printPairs(std::views::zip(v1, v2));
```

Output:

```
* 2:10
* 2:20
* 2:30
```


Basic Idioms Broken by Standards Views

C++20/C++23

- You can iterate if the range is `const` — Broken for views
- A read iteration does not change state — Broken for views
- Concurrent read iterations are safe — Broken for views
- `const` collections have `const` elements — Broken for views
- `cbegin()` makes elements immutable — Broken for views
- A copy of a range has the same state — Broken for views
- `const`-declared elements are `const` (C++23) — Broken for views

C++23: Some `const` Fixes for views

C++23

- **New type traits `std::const_iterator<>`, `std::const_sentinel<>`, and `std::iter_const_reference<>`**
- **`std::ranges::cbegin()` and `std::ranges::cend()` fixed**
  - return `const_iterator<>` and `const_sentinel<>`
  - `std::cbegin()` and `std::cend()` are still broken
- **All views have `cbegin()` and `cend()` member functions**
  - Returning `std::ranges::cbegin()` and `std::ranges::cend()`
- **New concept `std::ranges::constant_range<>`**
- **New helper function `std::ranges::as_const()`**
  - Makes elements `const`
  - Note: `std::as_const()` does ***not*** make elements `const`

Dealing with Containers and Views with C++23

C++23

Idea by Tomasz Kamiński

```
template<typename T>
void print(T&& coll) {
{
  if constexpr (std::ranges::const_range<T>) {
    // the passed range and its elements are constant
    for (const auto& elem : coll) {
      std::cout << elem << ' ';
    }
    std::cout << '\n';
  }
  else {
    // call this function again after making elements const:
    print(std::views::as_const(std::forward<T>(coll)));
  }
}
```

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

C++ Standard Views are Broken

for Ordinary Programmers

Belleviews

C++ Views that Just Work

Belleviews Goals

- **Usability**
  - Views should just work
  - Simplicity
  - Consistency
- **Safety**
  - No non-intuitive undefined behavior
- **Predictability**
  - No breakage of standard idioms
  - Common use cases should work as expected
- **Performance**
  - Caching can explicitly requested (e.g. by the **eager_begin** view)

Belleviews Principles

- **Iterating over a view is stateless**
  - You can always iterate when the view is const
  - You can iterate concurrently
  - Read iterations do not have any impact on later behavior
- **Respect the idioms of containers and arrays**
- **Honor constness**
  - Always propagate constness
  - Avoid using references that break const for elements
- **A copy of a view always behaves the same as its source**
- **All view types have a name ending with view**
  - Use sub_view instead of subrange
- **For all view types there is an adaptor/factory**
  - New factory bel::views::sub(beg,end)
- **Fix all knows flaws of specific views**

Iterate Over `const` Views

C++20 / Bel

```
void print(const auto& coll) {
  ...
}
```

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};
```

	Std Views
`print(vec | std::views::take(3));`	✓
`print(lst | std::views::take(3));`	✓
`print(vec | std::views::drop(3));`	✓
`print(lst | std::views::drop(3));`	CT Error
`for (int v : lst | std::views::drop(3)) { std::cout << v << ' '; }`	✓
`print(vec | std::views::filter(...));`	CT Error

Iterate Over `const` Views

C++20 / Bel

```
void print(const auto& coll) {
  ...
}
```

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};
```

	Std Views	Belleviews
`print(vec | bel::views::take(3));`	✓	✓
`print(lst | bel::views::take(3));`	✓	✓
`print(vec | bel::views::drop(3));`	✓	✓
`print(lst | bel::views::drop(3));`	CT Error	✓
`for (int v : lst | bel::views::drop(3)) { std::cout << v << ' '; }`	✓	✓
`print(vec | bel::views::filter(...));`	CT Error	✓

Iterate Over const Views

C++20 / Bel

```
void print(const auto& coll) {
  ...
}
```

```
std::vector<int> vec{0, 8, 15, 47, 11, 42};
std::list<int> lst{0, 8, 15, 47, 11, 42};
```

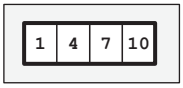
```
print(vec | std/bel::views::take(3));
print(lst | std/bel::views::take(3));
print(vec | std/bel::views::drop(3));
print(lst | std/bel::views::drop(3));
for (int v : lst | std/bel::views::drop(3)) {
   std::cout << v << ' ';
}
print(vec | std/bel::views::filter(...));
```

Std Views	Belleviews
✓	✓
✓	✓
✓	✓
CT Error	✓
✓	✓
CT Error	✓

Using the Filter View

C++20

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




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

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

Std Views	Belleviews
RT Error	✓

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

Std Views	Belleviews
RT Error	✓

Concurrent Iterations

C++20

```
std::list<int> coll{0, 8, 15, 47, 11, 42};

auto v = coll | std/bel::views::drop(2);

// while another thread prints the elements:
std::jthread printThread{[&] {
  for (const auto& elem : v) {
    std::cout << elem << ' ';
  }
  std::cout << '\n';
}};

// this thread computes the sum of the elements:
auto sum = std::accumulate(v.begin(), v.end(),
                           0L);
```

Std Views	Belleviews
RT Error	✓

const Propagation

C++20

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

const auto& v = vec | std/bel::views::drop(2);

v[0] += 42;

if (v.front() = 2) {
  ...
}

auto v2 = v;      // NOTE: removes constness
v2[0] += 42;
```

Std Views	Belleviews
OOPS: OK	ERROR
OOPS: OK	ERROR
OK	OK

Modifications

C++20

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

Std Views	Belleviews
3 4 3 4	3 4 3 4

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

Std Views	Belleviews
2 3 4 3 4	2 3 4 2 3 4

```
vec.insert(vec.begin(), {8, 9, 0, -1});
lst.insert(lst.begin(), {8, 9, 0, -1});
print(vVec, vLst);
```

Std Views	Belleviews
0 -1 0 1 2 3 4 3 4	0 -1 0 1 2 3 4 0 -1 0 1 2 3 4

```
auto vVecB = vVec;
auto vLstB = vLst;
print(vVecB, vLstB);
```

Std Views	Belleviews
0 -1 0 1 2 3 4 0 -1 0 1 2 3 4	0 -1 0 1 2 3 4 0 -1 0 1 2 3 4

Belleviews Status

- https://github.com/josuttis/belleviews

Please help to make C++ better

- **Not even beta**
  - I need your help (code, test, reviews)
- **But we already see how simply and safe views could have been**
- **I don't promise to solve all problems**
  - Ideally some time adopted by the C++ standard
- **Code based on C++20 ranges and const fixes of C++23**

Thank You!

Nicolai M. Josuttis

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