Range adaptors – 5 years after C++20

Hannes Hauswedell

Introduction

Core concepts

Indirections and lifetime

Range adaptors

Summary

About me

Bioinformatics

- PhD from FU Berlin
- Book on Sequence Analysis and C++

ISO Committee

- Irregular since 2018.
- Mostly LEWG & SG9.
- Represent Iceland.

DeepL

- One of the biggest AI companies in Germany/Europe.
- C++, Software Engineering, Research data, some ML...
- Disclaimer: all views are my own!

Why this talk?

Ask Reddit:

Reddit · r/cpp
Ca. 20 Kommentare · vor 2 Jahren

C++ views are broken : r/cpp

Some, but not all. **Views having non-const begin/end is still a huge flaw**, and makes it incompatible with existing libraries, as well as unsafe ...

Reddit · r/cpp
Ca. 20 Kommentare · vor 2 Jahren

Are standard views "completely broken"? : r/cpp

It's not broken. It's by design. Fixing some aspects means loss of some functionalities or performance.

Reddit · r/cpp
Ca. 30 Kommentare · vor 2 Jahren

Are there likely to be any changes to views to fix the issues ...

Or are we now stuck since any change will either **break** backwards compatibility or require adding a second official **view** library (yuck)? I use ...

39 Antworten · Top-Antwort: so I can easily imagine that there is another side to the story Indeed. The t…

Artistic_Yoghurt4754 • vor 3 Jahren

Omg, this issue is a real war zone and I am not even sure I want to pick a side. I'm not sure I want to spend the time learning ranges anymore :'(

5 Antworten Auszeichnen Teilen ···

Ask my mailbox (in 2025):

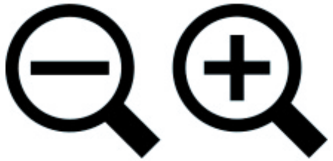
`[isocpp-sg9] ... the broken filter view ...                178 messages`

Why this talk?

- C++ Ranges are the largest addition to the standard library (ever).
- There are different expectations of what they can/should do.
- In hindsight, we would likely do some things differently.

Why this talk?

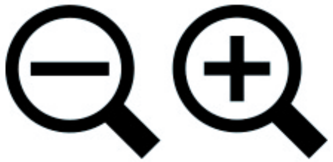
- C++ Ranges are the largest addition to the standard library (ever).
- There are different expectations of what they can/should do.
- In hindsight, we would likely do some things differently.



- Many controversies arise around very specific examples.
- Many suggested "fixes" only change that specific example, do not consider "the big picture".

Why this talk?

- C++ Ranges are the largest addition to the standard library (ever).
- There are different expectations of what they can/should do.
- In hindsight, we would likely do some things differently.



- Many controversies arise around very specific examples.
- Many suggested "fixes" only change that specific example, do not consider "the big picture".

- This talk will cover some aspects of what I consider "the big picture".
- I will sometimes make general statements and not mention the tiny exceptions (because everything in C++ has exceptions).

Cool stuff

```
void printLongWordsUppercase(std::string_view const str)
{
  auto notPunct = [] (unsigned char c) -> bool { return !std::ispunct(c); };
  auto toUpper  = [] (unsigned char c) -> char { return std::toupper(c); };
  auto isLong   = [] (auto && w) -> bool { return std::ranges::distance(w) > 4; };

  auto view = str                               // "This is an average, boring sen…"
            | std::views::filter(notPunct)      // "This is an average boring sent…"
            | std::views::transform(toUpper)    // "THIS IS AN AVERAGE BORING SENT…"
            | std::views::split(' ')            // ["THIS", "IS", "AN", "AVERAGE", …]
            | std::views::filter(isLong);       // ["AVERAGE", "BORING", "SENTENCE"]


  std::print("{::s}", view);
}
```

```
printLongWordsUppercase("This is a an average, boring sentence.");
// [AVERAGE, BORING, SENTENCE]
```

Iterators

```
auto find(auto it, auto sen, auto const & val)
{
    while ((it != sen) && (*it != val))
        ++it;

    return it;
}
```

- Iterators are the abstraction of choice since C++98.
- Generalisation of "pointers to *begin* and *end* of collection".
- Since `sen` may be of a different type from `it`, we say "iterator-sentinel-pair".

Ranges

```
auto find(auto it, auto sen, auto const & val)
{
    while ((it != sen) && (*it != val))
        ++it;

    return it;
}

auto find(auto && rng, auto const & val)
{
    return find(begin(rng), end(rng), val);
}
```

- The "new" abstraction for collections since C++20.
- A range is a type where `begin()` returns an iterator and `end()` returns a sentinel for that iterator.

👾 Build ranges on iterators – first design decision

Status quo:

- The ranges design is based on iterators.

👾 Build ranges on iterators – first design decision

Status quo:

- The ranges design is based on iterators.

Pro:

- Compatible with interfaces that expect an iterator-sentinel-pair.
  - If you have a range and need iterators, just call `begin()` / `end()`.
  - If you have iterators and need a range, just create a `std::ranges::subrange`.
- Well-understood and established design.

Con:

- Inherits some of the limitations of the previous design.
- In particular: "go-next" (`operator++`) and "read-value" (`operator*`) are always separate operations.

The range concept

`std::ranges::range` is the "core" concept.

- requires that `sr::begin()` on the type return an iterator
- requires that `sr::end()` on the type return a sentinel
- *Nothing is promised beyond this!*

ℹ️ On a lot of slides:

```
namespace sr =
std::ranges;
```

The range concept

`std::ranges::range` is the "core" concept.

- requires that `sr::begin()` on the type return an iterator
- requires that `sr::end()` on the type return a sentinel
- *Nothing is promised beyond this!*

ℹ️ On a lot of slides:

```
namespace sr = std::ranges;
```

Containers (e.g. `std::vector<int>`) are ranges:

- `sr::begin(cnt)` → `cnt.begin()`
- `sr::end(cnt)` → `cnt.end()`

The range concept

`std::ranges::range` is the "core" concept.

- requires that `sr::begin()` on the type return an iterator
- requires that `sr::end()` on the type return a sentinel
- *Nothing is promised beyond this!*

ℹ️ On a lot of slides:

```
namespace sr =
std::ranges;
```

Containers (e.g. `std::vector<int>`) are ranges:

- `sr::begin(cnt)` → `cnt.begin()`
- `sr::end(cnt)` → `cnt.end()`

An iterator-sentinel-pair on it's own is already a range:

- `std::ranges::subrange<It, Sen>`, bascically a `std::pair<It, Sen>`

Range categories

Iterator	Properties	Range
std::input_iterator	++ (one iteration)	sr::input_range
std::forward_iterator	++ (multi-pass)	sr::forward_range
std::bidirectional_iterator	--	sr::bidirectional_range
std::random_access_iterator	[]	sr::random_access_range
std::contiguous_iterator	adjacent storage	sr::contiguous_range

- Iterator categories refine the previous one.
- Range categories refine the previous one and require the respective iterator concept.

Range categories

The range categories add no requirements beyond the iterator-sentinel requirements.

```
template <sr::random_access_range Rng>
    requires std::integral<sr::range_value_t<Rng>> && sr::sized_range<Rng>
auto sum(Rng && rng)
{
    sr::range_value_t<Rng> r{};
    for (size_t i = 0; i < sr::size(rng); ++i)
        r += rng[i];
    return r;
}
```

Range categories

The range categories add no requirements beyond the iterator-sentinel requirements.

```
template <sr::random_access_range Rng>
    requires std::integral<sr::range_value_t<Rng>> && sr::sized_range<Rng>
auto sum(Rng && rng)
{
    sr::range_value_t<Rng> r{};
    for (size_t i = 0; i < sr::size(rng); ++i)
        r += rng[i];
    return r;
}
```

- Example is under-constrained!

Range categories

The range categories add no requirements beyond the iterator-sentinel requirements.

```
template <sr::random_access_range Rng>
    requires std::integral<sr::range_value_t<Rng>> && sr::sized_range<Rng>
auto sum(Rng && rng)
{
    sr::range_value_t<Rng> r{};
    for (size_t i = 0; i < sr::size(rng); ++i)
        r += sr::begin(rng)[i]; // NEED TO OPERATE ON ITERATOR!
    return r;
}
```

- Example is under-constrained!
- `sr::random_access_range` does not imply that the range offers `operator[]`, only the iterator is required to.

Range categories

The range categories add no requirements beyond the iterator-sentinel requirements.

```
template <sr::random_access_range Rng>
    requires std::integral<sr::range_value_t<Rng>> && sr::sized_range<Rng>
auto sum(Rng && rng)
{
    sr::range_value_t<Rng> r{};
    for (size_t i = 0; i < sr::size(rng); ++i)
        r += sr::begin(rng)[i]; // NEED TO OPERATE ON ITERATOR!
    return r;
}
```

- Example is under-constrained!
- `sr::random_access_range` does not imply that the range offers `operator[]`, only the iterator is required to.
- C++26 will ship the four random access ranges `std::flat_*`; two without `operator[]` and two with different `operator[]`.

The multi-pass guarantee

Forward (multi-pass) iterators:

1. can be default-constructed, copied, compared-against-self (`std::regular`)
2. two iterators are a range (`std::sentinel_for<FIt, FIt>`)
3. incrementing one iterator does not invalidate another
4. copies that are independently incremented yield the same values

The multi-pass guarantee

Forward (multi-pass) iterators:

1. can be default-constructed, copied, compared-against-self (`std::regular`)
2. two iterators are a range (`std::sentinel_for<FIt, FIt>`)
3. incrementing one iterator does not invalidate another
4. copies that are independently incremented yield the same values

---

Forward (multi-pass) ranges:

1. `sr::begin(rng)` returns a forward iterator.
2. `sr::begin(rng)` can be called multiple times, and does not modify `rng`.
3. Complexity of calling `sr::begin()` repeatedly shall ammortize to O(1).

You can iterate over the range multiple times and will observe the same elements.

The multi-pass guarantee

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

auto it  = vec.begin();
auto it2 = vec.begin();
auto cpy = it;

assert(it == cpy);
assert(*it == *cpy);

assert(it++ == cpy);
assert(++cpy == it);
assert(*it == *cpy);
```

The multi-pass guarantee

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

auto it  = vec.begin();
auto it2 = vec.begin();
auto cpy = it;

assert(it == cpy);
assert(*it == *cpy);

assert(it++ == cpy);
assert(++cpy == it);
assert(*it == *cpy);
```

- All containers are forward (multi-pass) ranges!
- **The iterator is an *observer* to the range;** creating the iterator (`begin()`), incrementing or deref-ing do not change the range *in an observable way.*
- The properties are likely what most people expect.

The multi-pass guarantee

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

auto it  = vec.begin();
auto it2 = vec.begin();
auto cpy = it;

assert(it == cpy);
assert(*it == *cpy);

assert(it++ == cpy);
assert(++cpy == it);
assert(*it == *cpy);
```

- All containers are forward (multi-pass) ranges!
- **The iterator is an *observer* to the range;** creating the iterator (`begin()`), incrementing or deref-ing do not change the range *in an observable way.*
- The properties are likely what most people expect.

⚠️ The following is **not** implied by `sr::forward_range<Rng>` ⚠️

- Default-construct, copy, or compare (`std::regular<Rng>`).
- Const-iterate (`sr::forward_range<Rng const>`).

Single-pass ranges

Single-pass iterators ("input" but not "forward"):

- Do not provide the multi-pass guarantee 🤓
- Not required to be default-constructible, copyable or comparable-with-self.
- May require separate sentinel type to form a range.
- If copyable, incrementing one iterator may invalidate others.

Single-pass ranges

Single-pass iterators ("input" but not "forward"):

- Do not provide the multi-pass guarantee 🤓
- Not required to be default-constructible, copyable or comparable-with-self.
- May require separate sentinel type to form a range.
- If copyable, incrementing one iterator may invalidate others.

Single-pass ranges:

1. `sr::begin(rng)` returns a single-pass iterator.
2. *Undefined behaviour* to call `sr::begin(rng)` more than once (generally).
3. If safe to call `sr::begin()` multiple times, no promises about complexity.

In general, assume that you can call `begin()` once and that the range is "exhausted" (empty) when/if the end is reached.

Single-pass ranges

```
std::istringstream strstr("1 3 5 7 8 9 10");

auto it  = std::istream_iterator<int>(strstr);
auto it2 = std::istream_iterator<int>(strstr);
auto cpy = it;

++it; // modifies strstr AND it2 AND cpy 👻
```

- No single-pass ranges in pre-C++20-stdlib.
- Only two single-pass iterators: `std::istream_iterator` and `std::istreambuf_iterator`

Single-pass ranges

```
std::istringstream strstr("1 3 5 7 8 9 10");

auto it  = std::istream_iterator<int>(strstr);
auto it2 = std::istream_iterator<int>(strstr);
auto cpy = it;

++it; // modifies strstr AND it2 AND cpy 👻
```

- No single-pass ranges in pre-C++20-stdlib.
- Only two single-pass iterators: `std::istream_iterator` and `std::istreambuf_iterator`

- C++20 introduces `std::generator` which will make them more common.
- Nico Jusuttis has a talk on generators tomorrow, go check it out!

```
std::generator<size_t> my_iota()
{
    for (size_t i = 0; true; ++i)
        co_yield i;
}
```

Single-pass ranges

- Single-pass ranges **change state** during iteration.
- The iterators **are not observers**, they modify the range!
- Natural to not always be default-constructible, comparable and/or copyable.

Single-pass ranges

- Single-pass ranges **change state** during iteration.
- The iterators **are not observers**, they modify the range!
- Natural to not always be default-constructible, comparable and/or copyable.

- Conceptionally, they only have one operation: `make_item()`.
- Separate `operator++` and `operator*` are not ideal; you can deref multiple times.

```
std::generator<std::string> parse_lines(std::istream & i) { /**/ }
```

- Should `operator*` return `std::string`, `std::string &` or `std::string &&`?

Single-pass ranges

- Single-pass ranges **change state** during iteration.
- The iterators **are not observers**, they modify the range!
- Natural to not always be default-constructible, comparable and/or copyable.

- Conceptionally, they only have one operation: `make_item()`.
- Separate `operator++` and `operator*` are not ideal; you can deref multiple times.

```
std::generator<std::string> parse_lines(std::istream & i) { /**/ }
```

- Should `operator*` return `std::string`, `std::string &` or `std::string &&`?

Post scriptum: not every range that generates elements is single-pass!

- e.g. `sr::repeat_view` generates one value for infinity.
- e.g. `sr::iota_view` generates a series of values.
- No state change, or only iterator, so both are multi-pass ranges.

👾 Minimal vs meaningful category concepts

Status quo:

- "Minimal" (range category concepts only comprise iterator category concepts).
- No `[]` on random access ranges.
- No (semi-)regularity or const-iterability on forward ranges.

👾 Minimal vs meaningful category concepts

Status quo:

- "Minimal" (range category concepts only comprise iterator category concepts).
- No `[]` on random access ranges.
- No (semi-)regularity or const-iterability on forward ranges.

Pro:

- Concepts can be composed, so they should be concise.
- You can always make your own, more refined concepts!
- If e.g. `random_access_range` requires more than the iterator concept, what would a range without the extras be?

Con:

- Concepts should be meaningful!
- Minimal concepts results in lack of features and consistency, e.g. `std::flat_set` lacking `[]`.

👾 One abstraction for single-pass and multi-pass ranges

Status quo:

- Single-pass (input-only) ranges are also "ranges", although fundamentally different.
- But, clearly separated by `sr::input_range` ↔ `sr::forward_range`.

👾 One abstraction for single-pass and multi-pass ranges

Status quo:

- Single-pass (input-only) ranges are also "ranges", although fundamentally different.
- But, clearly separated by `sr::input_range` ↔ `sr::forward_range`.

Pro:

- Consistent with established iterator concepts.
- Several algorithms, e.g. `find()`, work on either (although they mean slightly different things).

Con:

- Other designs more suitable for single-pass "ranges".
- Some unexpected and some undefined behaviour.
- Increased complexity for handling both in algorithms.

Indirections

	`std::vector<int>`	`std::vector<int> *`
as const	protects elements	elements mutable
copy	copies elements	pointer copied
==	compares elements	compares address
destruction	frees elements	elements untouched
complexities	O(n)	O(1)
`sr::range`	yes	no

Indirections

	`std::vector<int>`	`std::vector<int> *`	`sr::subrange<int*,int*>`
as const	protects elements	elements mutable	elements mutable
copy	copies elements	pointer copied	pointers copied
==	compares elements	compares address	n/a
destruction	frees elements	elements untouched	elements untouched
complexities	O(n)	O(1)	O(1)
`sr::range`	yes	no	yes

Indirections

	`std::vector<int>`	`std::vector<int> *`	`sr::subrange<int*,int*>`
as const	protects elements	elements mutable	elements mutable
copy	copies elements	pointer copied	pointers copied
==	compares elements	compares address	n/a
destruction	frees elements	elements untouched	elements untouched
complexities	O(n)	O(1)	O(1)
`sr::range`	yes	no	yes

The range concepts cover containers—but also types that behave like pointers to containers.

Indirections

```
void foobar(std::ranges::forward_range auto && rng)
{
    auto const & cns = rng;  // Protects the elements or not? Is it even a range?




}
```

Indirections

```
void foobar(std::ranges::forward_range auto && rng)
{
    auto const & cns = rng;  // Protects the elements or not? Is it even a range?

    auto         cpy = rng;  // Copies elements? Allocates? Is it even well-formed?


}
```

Indirections

```
void foobar(std::ranges::forward_range auto && rng)
{
    auto const & cns = rng;  // Protects the elements or not? Is it even a range?

    auto         cpy = rng;  // Copies elements? Allocates? Is it even well-formed?

    bool ret = (cpy == rng); // What is being compared? Is it even well-formed?
}
```

Standard library range concepts make no promises!

👾 One abstraction for containers and indirect ranges (1)

Status quo:

- A Range can have the semantics of a container, or of a "pointer-to-container".
- The usage patterns are identical (no dereference required like for a pointer).

👾 One abstraction for containers and indirect ranges (1)

Status quo:

- A Range can have the semantics of a container, or of a "pointer-to-container".
- The usage patterns are identical (no dereference required like for a pointer).

Pro:

- Easy to use.
- Allows common interface for "vector" and "subrange of vector".
- Allows replacing `std::string const &` with `std::string_view`.

Con:

- Easy to mis-use.
- `const`-correctness becomes more difficult to achieve.
- Performance implications difficult foresee.

Lifetime

```
auto find(auto it, auto sen, auto const & val)
{
    while ((it != sen) && (*it != val))
        ++it;

    return it;
}

auto find(auto && rng, auto const & val)
{
    return find(sr::begin(rng), sr::end(rng), val);
}
```

Lifetime

```
auto find(auto it, auto sen, auto const & val)
{
    while ((it != sen) && (*it != val))
        ++it;

    return it;
}

auto find(auto && rng, auto const & val)
{
    return find(sr::begin(rng), sr::end(rng), val);
}
```

```
std::string s   = "foobar";
auto        it1 = find(s.begin(), s.end(), 'a');
auto        it2 = find(s, 'a');
auto        it3 = find(std::string{"foobar"}, 'a');
```

Lifetime

```
auto find(auto it, auto sen, auto const & val)
{
    while ((it != sen) && (*it != val))
        ++it;

    return it;
}

auto find(auto && rng, auto const & val)
{
    return find(sr::begin(rng), sr::end(rng), val);
}
```

```
std::string s    = "foobar";
auto        it1 = find(s.begin(), s.end(), 'a');        // safe
auto        it2 = find(s, 'a');                         // safe
auto        it3 = find(std::string{"foobar"}, 'a');     // rvalue, dangling
```

Lifetime

```
auto find(auto it, auto sen, auto const & val)
{
    while ((it != sen) && (*it != val))
        ++it;

    return it;
}

auto find(auto && rng, auto const & val)
{
    return find(sr::begin(rng), sr::end(rng), val);
}
```

```
std::string s   = "foobar";
auto        it1 = find(s.begin(), s.end(), 'a');        // safe
auto        it2 = find(s, 'a');                         // safe
auto        it3 = find(std::string{"foobar"}, 'a');     // rvalue, dangling
auto        it4 = find(sr::subrange{s}, 'a');           // rvalue, not dangling
```

Lifetime

```
auto find(auto it, auto sen, auto const & val)
{
    while ((it != sen) && (*it != val))
        ++it;

    return it;
}

auto find(auto & rng, auto const & val)
{
    return find(sr::begin(rng), sr::end(rng), val);
}
```

```
std::string s   = "foobar";
auto        it1 = find(s.begin(), s.end(), 'a');        // safe
auto        it2 = find(s, 'a');                         // safe
// auto        it3 = find(std::string{"foobar"}, 'a');  // dangling prevented ✔️
// auto        it4 = find(sr::subrange{s}, 'a');        // also prevented ❌
```

Borrowed ranges

```
auto find(auto it, auto sen, auto const & val)
{
    while ((it != sen) && (*it != val))
        ++it;

    return it;
}

auto find(sr::borrowed_range auto && rng, auto const & val)
{
    return find(begin(rng), end(rng), val);
}
```

```
std::string s   = "foobar";
auto        it1 = find(s.begin(), s.end(), 'a');        // safe
auto        it2 = find(s, 'a');                          // safe
// auto        it3 = find(std::string{"foobar"}, 'a');  // dangling prevented ✔
auto        it4 = find(sr::subrange{s}, 'a');            // "good" rvalue allowed ✔
```

Borrowed ranges

`std::ranges::borrowed_range`, "a range with reference semantics":

- Lvalues and rvalues of ranges whose iterators can outlive the range.
  - ✔️ `sr::subrange<It, Sen> &`,
  - ✔️ `sr::subrange<It, Sen>`, `sr::subrange<It, Sen> &&`
- Lvalue references to any range.
  - ✔️ `std::vector<int> &`
  - ❌ `std::vector<int>`, `std::vector<int> &&`

Borrowed ranges

`std::ranges::borrowed_range`, "a range with reference semantics":

- Lvalues and rvalues of ranges whose iterators can outlive the range.
  - ✔️ `sr::subrange<It, Sen> &`,
  - ✔️ `sr::subrange<It, Sen>`, `sr::subrange<It, Sen> &&`
- Lvalue references to any range.
  - ✔️ `std::vector<int> &`
  - ❌ `std::vector<int>`, `std::vector<int> &&`

```
                                           // Lifetime dependencies
std::vector<int> vec{1,2,3};               //  ]
auto it = vec.begin();                     //
                                           //
                                           //
                                           //
                                           //
```

Borrowed ranges

`std::ranges::borrowed_range`, "a range with reference semantics":

- Lvalues and rvalues of ranges whose iterators can outlive the range.
  - ✔️ `sr::subrange<It, Sen> &`,
  - ✔️ `sr::subrange<It, Sen>`, `sr::subrange<It, Sen> &&`
- Lvalue references to any range.
  - ✔️ `std::vector<int> &`
  - ❌ `std::vector<int>`, `std::vector<int> &&`

```
                                    // Lifetime dependencies
std::vector<int> vec{1,2,3};        //  ┐ ┐ ┐
auto it = vec.begin();              //  ┘ │ │
                                    //    │ │
sr::subrange indi{vec};             //    ┘ │
                                    //      │
auto it2 = indi.begin();            //      ┘
```

Borrowed ranges

`std::ranges::borrowed_range`, "a range with reference semantics":

- Lvalues and rvalues of ranges whose iterators can outlive the range.
  - ✔️ `sr::subrange<It, Sen> &`,
  - ✔️ `sr::subrange<It, Sen>`, `sr::subrange<It, Sen> &&`
- Lvalue references to any range.
  - ✔️ `std::vector<int> &`
  - ❌ `std::vector<int>`, `std::vector<int> &&`

```
                                        // Lifetime dependencies
std::vector<int> vec{1,2,3};            //   ┐ ┐ ┐
auto it = vec.begin();                  //   ┘ │ │
                                        //     ┘ │
sr::subrange indi{vec};                 //       │  ┐
                                        //       │  ─   not dependent on indi!
auto it2 = indi.begin();                //       ┘  ┘
```

Borrowed ranges

`std::ranges::borrowed_range`, "a range with reference semantics":

- Lvalues and rvalues of ranges whose iterators can outlive the range.
  - ✔️ `sr::subrange<It, Sen> &`,
  - ✔️ `sr::subrange<It, Sen>`, `sr::subrange<It, Sen> &&`
- Lvalue references to any range.
  - ✔️ `std::vector<int> &`
  - ❌ `std::vector<int>`, `std::vector<int> &&`

```
                                   // Lifetime dependencies
std::vector<int> vec{1,2,3};       //  ┐ ┐ ┐
auto it = vec.begin();             //  ┘ │ │
                                   //    │ │
std::vector<int> & indi{vec};      //    ┘ │  ┐
                                   //      │  -   not dependent on indi!
auto it2 = indi.begin();           //      ┘  ┘
```

👾 One abstraction for containers and indirect ranges (2)

Status quo:

- A Range can have the semantics of a container, or of a "pointer-to-container".
- In general, iterators depend on the lifetime of the range they were created from (e.g. containers), but for so called *borrowed ranges* they don't.

👾 One abstraction for containers and indirect ranges (2)

Status quo:

- A Range can have the semantics of a container, or of a "pointer-to-container".
- In general, iterators depend on the lifetime of the range they were created from (e.g. containers), but for so called *borrowed ranges* they don't.

Pro:

- Protect interfaces from lifetime issues.
- Clear definition of what "indirect" / "non-owning" means.
- Reference analogy helpful?

Con:

- Only workaround for flawed previous decision (mixing containers and indirect ranges)?
- `concept<T> == false` but `concept<T&> == true` may be surprising for some.
- Requires explicit opt-in through type trait.

Range adaptors - terminology

Range adaptor, a range that depends on, or wraps, another range.

- simple ones: `std::span<int>`, `std::string_view`, `sr::subrange<It,Sen>`

Range adaptors - terminology

Range adaptor, a range that depends on, or wraps, another range.

- simple ones: `std::span<int>`, `std::string_view`, `sr::subrange<It,Sen>`
- composable: `sr::transform_view<V,Fn>`, `sr::filter_view<V, Fn>` …

Range adaptors - terminology

Range adaptor, a range that depends on, or wraps, another range.

- simple ones: `std::span<int>`, `std::string_view`, `sr::subrange<It,Sen>`
- composable: `sr::transform_view<V,Fn>`, `sr::filter_view<V, Fn>` …
- container adaptors: `std::queue<T, Cont>`, `std::flat_set<T, Comp, Cont>` (C++26) …

Range adaptors - terminology

Range adaptor, a range that depends on, or wraps, another range.

- simple ones: `std::span<int>`, `std::string_view`, `sr::subrange<It,Sen>`
- composable: `sr::transform_view<V,Fn>`, `sr::filter_view<V, Fn>` …
- container adaptors: `std::queue<T, Cont>`, `std::flat_set<T, Comp, Cont>` (C++26) …

In this broad sense, no implications arise for programmers; and no `concept` covers all range adaptors.

- The "simple adaptors" have widespread usage in APIs. Container adaptors are used standalone.
- The composable adaptors are most powerful, but also raise the most questions.

Range adaptors - the big three topics

1. Range adaptors that cache begin.
2. Different forms of indirection.
3. The dilution of the `view` concept.

Range adaptors - the big three topics

1. Range adaptors that cache begin.
2. Different forms of indirection.
3. The dilution of the `view` concept.

®

Range adaptors - caching begin()

Forward (multi-pass) ranges:

1. `sr::begin(rng)` returns a forward iterator.
2. `sr::begin(rng)` can be called multiple times, and does not modify `rng`.
3. Complexity of calling `sr::begin()` repeatedly shall ammortize to O(1).

Range adaptors - caching begin()

Forward (multi-pass) ranges:

1. `sr::begin(rng)` returns a forward iterator.
2. `sr::begin(rng)` can be called multiple times, and does not modify `rng`.
3. Complexity of calling `sr::begin()` repeatedly shall ammortize to O(1).

```
std::vector<int> vec{1, 1, 1, 2, 2, 1, 2};

auto v = vec | std::views::filter(is_even) | std::views::reverse;

for (auto it = v.begin(); it != v.end(); ++it)
    std::println("{}", *it);
```

Range adaptors - caching begin()

Forward (multi-pass) ranges:

1. `sr::begin(rng)` returns a forward iterator.
2. `sr::begin(rng)` can be called multiple times, and does not modify `rng`.
3. Complexity of calling `sr::begin()` repeatedly shall ammortize to O(1).

```
std::vector<int> vec{1, 1, 1, 2, 2, 1, 2};

auto v = vec | std::views::filter(is_even) | std::views::reverse;

for (auto it = v.begin(); it != v.end(); ++it)
    std::println("{}", *it);
```

- Calling `begin()` on the filter need to `find()` the first even number (O(n)).
- Calling `end()` on the reverse invokes `begin()` on the filter → quadratic complexity unless `begin()` is cached.

Range adaptors - caching begin()

Strategies for caching begin():

1. Only advertise `input_range` and do not cache.
   - Okay if you only need one pass.
   - Prevents chaining adaptors that require multi-pass, like `std::views::reverse` (even you only need one pass over them!).

Range adaptors - caching begin()

Strategies for caching begin():

1. Only advertise `input_range` and do not cache.
    - Okay if you only need one pass.
    - Prevents chaining adaptors that require multi-pass, like `std::views::reverse` (even you only need one pass over them!).
2. Cache when `begin()` is called the first time.
    - Calling `begin()` now has side-effects (although not observable).
    - The range is no longer const-iterable, e.g. you cannot pass it to `void print(auto const & rng);`

Range adaptors - caching begin()

Strategies for caching begin():

1. Only advertise `input_range` and do not cache.
    - Okay if you only need one pass.
    - Prevents chaining adaptors that require multi-pass, like `std::views::reverse` (even you only need one pass over them!).
2. Cache when `begin()` is called the first time.
    - Calling `begin()` now has side-effects (although not observable).
    - The range is no longer const-iterable, e.g. you cannot pass it to `void print(auto const & rng);`
3. Cache on construction.
    - Cost of finding begin always paid, even if there is no iteration.
    - Not fully lazy-evaluated.

Range adaptors - caching begin()

Strategies for caching begin():

1. Only advertise `input_range` and do not cache.
   - Okay if you only need one pass.
   - Prevents chaining adaptors that require multi-pass, like `std::views::reverse` (even you only need one pass over them!).
2. Cache when `begin()` is called the first time.
   - Calling `begin()` now has side-effects (although not observable).
   - The range is no longer const-iterable, e.g. you cannot pass it to `void print(auto const & rng);`
3. Cache on construction.
   - Cost of finding begin always paid, even if there is no iteration.
   - Not fully lazy-evaluated.

Range adaptors - caching begin()

Forward (multi-pass) ranges:

1. `sr::begin(rng)` returns a forward iterator.
2. `sr::begin(rng)` can be called multiple times, and does not modify `rng`.
3. Complexity of calling `sr::begin()` repeatedly shall ammortize to O(1).

Range adaptors - caching begin()

Forward (multi-pass) ranges:

1. `sr::begin(rng)` returns a forward iterator.
2. `sr::begin(rng)` can be called multiple times, and does not modify `rng` **observably**.
3. Complexity of calling `sr::begin()` repeatedly shall ammortize to O(1).

Range adaptors - caching begin()

Forward (multi-pass) ranges:

1. `sr::begin(rng)` returns a forward iterator.
2. `sr::begin(rng)` can be called multiple times, and does not modify `rng` **observably**.
3. Complexity of calling `sr::begin()` repeatedly shall ammortize to O(1).

Cache when `begin()` is called the first time:

```
void print(auto const & rng);

std::vector<int> vec{1, 1, 1, 2, 2, 1, 2};
auto v = vec | std::views::filter(is_even);
// print(v);                                    // ill-formed
```

The programmer observes no change-of-state in `v`, but they cannot pass it by `const &`.

Range adaptors - caching begin()

1. Only advertise `input_range` and do not cache.
    - Okay if you only need one pass.
    - Prevents chaining adaptors that require multi-pass like `std::views::reverse` (even you only need one pass over them!).

Range adaptors - caching begin()

1. Only advertise `input_range` and do not cache.
    - Okay if you only need one pass.
    - Prevents chaining adaptors that require multi-pass like `std::views::reverse` (even you only need one pass over them!).

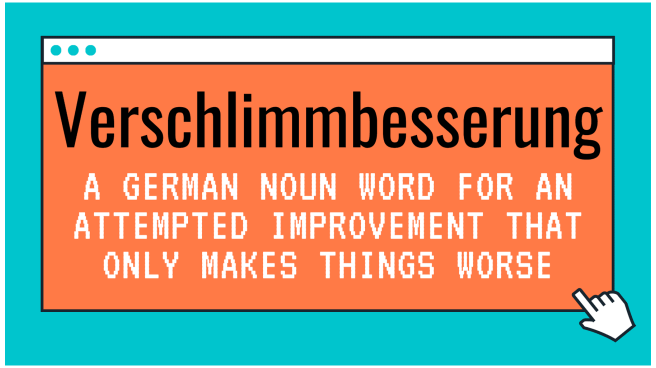
- C++26 introduces `std::views::input_filter` with the above semantics.
- Surprisingly, this will actually be const-iterable… because it doesn't need to cache.

Range adaptors - caching begin()

1. Only advertise `input_range` and do not cache.
   - Okay if you only need one pass.
   - Prevents chaining adaptors that require multi-pass like `std::views::reverse` (even you only need one pass over them!).

- C++26 introduces `std::views::input_filter` with the above semantics.
- Surprisingly, this will actually be const-iterable… because it doesn't need to cache.

- `input_range`: can change observably on `begin()` and/or iteration; but `views::input_filter` is const-iterable.
- `forward_range`: cannot change observably on `begin()` and/or iteration; but `views::filter` is not const-iterable.



👾 Caching begin()

Status quo:

- Some multi-pass range adaptors initialise a cache the first time `begin()` is called.
- Some range adaptors now intentionally demote category to `input_range` to avoid this.

👾 Caching begin()

Status quo:

- Some multi-pass range adaptors initialise a cache the first time `begin()` is called.
- Some range adaptors now intentionally demote category to `input_range` to avoid this.

Pro:

- Caching `begin()` is necessary for multi-pass ranges.
- "Lazier" than the alternative.

Con:

- Weakens the multi-pass guarantee (that would otherwise imply const-iterability).
- Const-iterable input ranges further confuse the mental model.

Range adaptors - the big three topics

1. Range adaptors that cache begin.
2. Different forms of indirection.
3. The dilution of the `view` concept.

Die drei ???

®

Range adaptors - indirection

Some adaptors from `std::views::` return borrowed ranges:

```
                                             // Lifetime dependencies
std::vector<int> vec{1,2,7,3};               //  ┐ ┐
                                             //  │ │
auto v = vec | std::views::take(3);          //  ┘ │  ┐
                                             //    │  ─   not dependent on v!
auto it = v.begin();                         //    ┘  ┘
```

Range adaptors - indirection

Some adaptors from `std::views::` return borrowed ranges:

```
std::vector<int> vec{1,2,7,3};                   // Lifetime dependencies
                                                  //  ┐ ┐
auto v = vec | std::views::take(3);               //  │ │
                                                  //  ┘ │  ┐
auto it = v.begin();                              //    │  ─   not dependent on v!
                                                  //    ┘  ┘
```

But other adaptors from `std::views::` do not:

```
std::vector<int> vec{1,2,7,3};                   // Lifetime dependencies
auto fn = [] (int i) { return i > 2; };          //  ┐ ┐
auto v = vec | std::views::filter(fn);           //  │ │
                                                  //  ┘ │  ┐
auto it = v.begin();                              //    │  │   is dependent on v!
                                                  //    ┘  ┘
```

Range adaptors - indirection

Implications:

```
std::vector<int> vec{1,2,7,3};
int mint   = *sr::min_element(vec | std::views::take(3));     // well-formed: 1

auto fn    = [] (int i) { return i > 2; };
//int minf = *sr::min_element(vec | std::views::filter(fn));  // ill-formed
```

Range adaptors - indirection

Implications:

```
std::vector<int> vec{1,2,7,3};
int mint    = *sr::min_element(vec | std::views::take(3));     // well-formed: 1

auto fn     = [] (int i) { return i > 2; };
//int minf = *sr::min_element(vec | std::views::filter(fn));  // ill-formed

auto v      = vec | std::views::filter(fn);                    // create temp. var
int minf    = *sr::min_element(v);                             // well-formed: 3
```

Range adaptors - indirection

Implications:

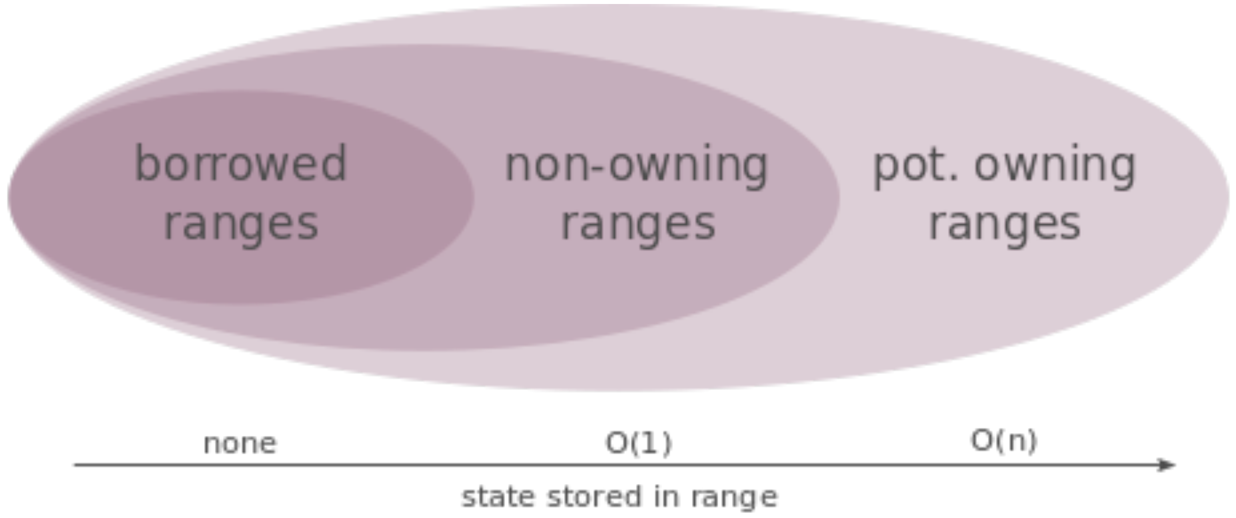
```
std::vector<int> vec{1,2,7,3};
int mint   = *sr::min_element(vec | std::views::take(3));     // well-formed: 1

auto fn    = [] (int i) { return i > 2; };
//int minf = *sr::min_element(vec | std::views::filter(fn));  // ill-formed

auto v     = vec | std::views::filter(fn);                    // create temp. var
int minf   = *sr::min_element(v);                             // well-formed: 3
```

- This is a design choice!
- We could store all state in the iterators: every (indirect) adaptor becomes borrowed.
- But the iterators could become much larger.

Range adaptors - indirection



Borrowed Range
contains only iterator-sentinel-pair; its iterators do not refer back to it.

⊂ **Non-owning Range**
may contain state, but not proportional to #elements (e.g. a functor).

⊂ **Range**
may contain the elements or state proportional to #elements.

👾 Multiple degrees of ownership/indirection

Status quo:

- Borrowed ranges are a well-defined concept, but not all adaptors-on-lvalues return borrowed ranges.
- "Non-owning" ranges that are not borrowed are more difficult to define.

👾 Multiple degrees of ownership/indirection

Status quo:

- Borrowed ranges are a well-defined concept, but not all adaptors-on-lvalues return borrowed ranges.
- "Non-owning" ranges that are not borrowed are more difficult to define.

Pro:

- Keeps iterators small.

Con:

- Prevents using iterators standalone.
- This is a common pattern, also in the standard library.
- Makes the mental model more complicated.

Range adaptors - the big three topics

1. Range adaptors that cache begin.
2. Different forms of indirection.
3. The dilution of the `view` concept.

Range adaptors - views

Me: »Please summarise in one sentence how views in C++ differ from other ranges!«

ChatGPT5:

Range adaptors - views

Me: »Please summarise in one sentence how views in C++ differ from other ranges!«

ChatGPT5: »In C++, *views* are lightweight, non-owning adaptors that provide a window over existing ranges or containers without copying or owning the underlying elements, unlike other ranges that may store or own their data.«

Range adaptors - views

Me: »Please summarise in one sentence how views in C++ differ from other ranges!«

ChatGPT5: »In C++, *views* are lightweight, non-owning adaptors that provide a window over existing ranges or containers without copying or owning the underlying elements, unlike other ranges that may store or own their data.«

⚠️ **Wrong** ⚠️

Range adaptors - views

Me: »Please summarise in one sentence how views in C++ differ from other ranges!«

ChatGPT5: »In C++, *views* are lightweight, non-owning adaptors that provide a window over existing ranges or containers without copying or owning the underlying elements, unlike other ranges that may store or own their data.«

⚠️ **Wrong** ⚠️

```
auto view1 = std::views::iota(0, 10);                     // not a range adaptor
auto view2 = std::string{"foobar"} | std::views::take(3); // owns the string
```

Range adaptors - views

`std::ranges::view`:

- `std::semiregular` (default-constructible, copyable)
- copyable in O(1) ("non-owning")

Range adaptors - views

`std::ranges::view`:

- ~~`std::semiregular` (default-constructible, copyable)~~
- ~~copyable in O(1) ("non-owning")~~

Changed by [P2415](P2415) post C++20.

Range adaptors - views

`std::ranges::view`:

- ~~`std::semiregular` (default-constructible, copyable)~~
- ~~copyable in O(1) ("non-owning")~~

Changed by [P2415](P2415) post C++20.

```
auto view2 = std::string{"foobar"} | std::views::take(3); // returns sr::owning_view
```

Range adaptors - views

`std::ranges::view`:

- ~~`std::semiregular` (default-constructible, copyable)~~
- ~~copyable in O(1) ("non-owning")~~

Changed by P2415 post C++20.

```
auto view2 = std::string{"foobar"} | std::views::take(3); // returns sr::owning_view
```

Eric Niebler Over a year ago

I was going to hold my tongue but ... " `owning_view` " was not part of the original design. Previously, you couldn't use the adaptors on a temporary container. The existence of " `owning_view` " muddies the design and contributes to this confusion. I'm sad it was ever added.

Range adaptors - views

Type	"Adaptor"	Destruct	Copy	⟲	⏲	View
`sr::transform_view<ref_view<…>, ...>`	yes	O(1)	O(1)	yes	yes	yes
`sr::subrange<int*, int*>`	yes	O(1)	O(1)	no	yes	yes
`sr::owning_view<...>`	yes	O(n)	n/a	yes	no	yes
`std::generator<int>`	?	?	n/a	yes	?	yes
`sr::iota_view<Val, Bound>`	no	O(1)	O(1)	no	no	yes
`sr::repeat_view<Val, Bound>`	no	O(1)	O(1)	yes	no	yes
`std::vector<int>`	no	O(n)	O(n)	yes	no	no

⟲: whether iterators depend on lifetime of range itself (`!sr::borrowed_range<>`)
⏲: whether iterators depend on lifetime of another range

👾 The view concept

Status quo:

- Views can be non-owning (indirect) or owning.

👾 The view concept

Status quo:

- Views can be non-owning (indirect) or owning.

Pro:

- You can move containers into views.

Con:

- Nobody can explain what "view" means.

Introduction

Basic range concepts

Indirections and lifetime

Range adaptors

Summary

Summary

- `std::ranges::range` probably encompasses more than it should
- `std::ranges::forward_range` probably encompasses more than it should
- `std::ranges::view` probably encompasses more than it should

Summary

- `std::ranges::range` probably encompasses more than it should
- `std::ranges::forward_range` probably encompasses more than it should
- `std::ranges::view` probably encompasses more than it should

80% of features with 20% of complexity possible,
but if you want 100% of features, you get 100% of complexity!

Summary

- `std::ranges::range` probably encompasses more than it should
- `std::ranges::forward_range` probably encompasses more than it should
- `std::ranges::view` probably encompasses more than it should

80% of features with 20% of complexity possible,
but if you want 100% of features, you get 100% of complexity!

The "mental model" for many terms might be muddy, but they still work quite well in practice!

Summary

```
void printLongWordsUppercase(std::string_view const str)
{
  auto notPunct = [] (unsigned char c) -> bool { return !std::ispunct(c); };
  auto toUpper  = [] (unsigned char c) -> char { return std::toupper(c); };
  auto isLong   = [] (auto && w) -> bool { return std::ranges::distance(w) > 4; };

  auto view = str                              // "This is an average, boring sen…"
            | std::views::filter(notPunct)     // "This is an average boring sent…"
            | std::views::transform(toUpper)   // "THIS IS AN AVERAGE BORING SENT…"
            | std::views::split(' ')           // ["THIS", "IS", "AN", "AVERAGE", …]
            | std::views::filter(isLong);      // ["AVERAGE", "BORING", "SENTENCE"]


  std::print("{::s}", view);
}
```

```
printLongWordsUppercase("This is a an average, boring sentence.");
// [AVERAGE, BORING, SENTENCE]
```

Thanks for attending the talk!

Questions?

My own ranges library:
https://github.com/h-2/radr

Blog:
https://hannes.hauswedell.net

LinkedIn:
https://www.linkedin.com/in/hannes-hauswedell/