

Missing (and Future?) C++ Range Concepts

Jonathan Müller — @foonathan | Meeting C++ | 2025-11-07

Over two decades of experience building a market-leading C++ add-in for Microsoft Office

- think-cell is the gold standard for creating professional PowerPoint presentations.
- Founded in 2002, we now have 1,200,000+ users at over 30,000 companies.
- Our software is built entirely in C++ and fully integrated into Microsoft Office.
- We are a member of Standard C++ Foundation, run the Berlin C++ Meetup and talk at conferences.
- We are driven by excellence, setting ambitious goals and fostering the growth of exceptional talent.

think-cell standard library: **Heavily** built around ranges.

- Predates `std::ranges` and `range-v3`
- Like `range-v3` evolution from `Boost.Ranges`
- Like `fLux`, implemented in terms of cursors
- Unlike `fLux`, cursors are an implementation detail

Partially public: github.com/think-cell/think-cell-library

See also: C++Now 2023: The New C++ Library: Strong Library Foundation for Future Projects

What this talk is about

- Missing range concepts we have implemented in the think-cell-library

What this talk is about

- Missing range concepts we have implemented in the think-cell-library
- Implementation had to be possible in the C++23 subset implemented by MSVC

What this talk is about

- Missing range concepts we have implemented in the think-cell-library
- Implementation had to be possible in the C++23 subset implemented by MSVC
- Talk will present them using C++23 and C++26 features for convenience

- Missing range concepts we have implemented in the think-cell-library
- Implementation had to be possible in the C++23 subset implemented by MSVC
- Talk will present them using C++23 and C++26 features for convenience
- Talk will use `std::ranges` not `tc ranges` (where there is an equivalent) for easier understanding

- Missing range concepts we have implemented in the think-cell-library
- Implementation had to be possible in the C++23 subset implemented by MSVC
- Talk will present them using C++23 and C++26 features for convenience
- Talk will use `std::ranges` not `tc ranges` (where there is an equivalent) for easier understanding

Disclaimer: No promise that any of this actually makes it into the standard library!

Structure of this talk

Problem #1 Optimizations for better performance

Problem #2 Metaprogramming for compile-time magic

Existing ranges and views:

```
namespace stdv = std::views;  
namespace stdr = std::ranges;
```

Potentially future ranges and views:

```
namespace std2v;  
namespace std2r;
```

Problem #1

Normalize tabs to spaces

Problem: Read a file normalizing all tab characters to four spaces.

Normalize tabs to spaces

Problem: Read a file normalizing all tab characters to four spaces.

```
std::string read_file_normalized(std::string_view path) {  
    return read_file(path)  
        | stdv::transform([](char c) {  
            if (c == '\\t')  
                return std::string(4, ' ');  
            else  
                return std::string(1, c);  
        })  
        | stdv::join  
        | stdr::to<std::string>();  
}
```

```
std::input_range auto read_file(std::string_view path);
```

```
std::input_range auto read_file(std::string_view path);
```

```
class read_file_iterator {  
    const char* _buffer_cur;  
    const char* _buffer_end;  
  
public:  
    char operator*() const { return *_buffer_cur; }  
  
    read_file_iterator& operator++() {  
        if (++_buffer_cur == _buffer_end)  
            read_more();  
        return *this;  
    }  
};
```

Generators

Iterators vs. imperative code

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
while (true) {  
    std::span<const char> buffer = read_more();  
    if (buffer.empty()) break;  
  
    for (char c : buffer) { // ^^^ read_file  
        for (char translated : fn(c)) { // transform | join  
            ...  
        }  
    }  
}
```

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
class join_iterator {  
    transform_iterator    _outer;  
    std::string::iterator _inner;  
};
```

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
class join_iterator {  
    transform_iterator    _outer;  
    std::string::iterator _inner;  
};
```

```
class transform_iterator {  
    read_file_iterator _base;  
};
```

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
class join_iterator {  
    transform_iterator    _outer;  
    std::string::iterator _inner;  
};
```

```
class transform_iterator {  
    read_file_iterator _base;  
};
```

```
class read_file_iterator {  
    const char* _buffer_cur;  
    const char* _buffer_end;  
};
```

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
char join_iterator::operator*() {  
    return *_inner;  
}
```

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
char join_iterator::operator*() {  
    return *_inner;  
}
```

```
std::string transform_iterator::operator*() {  
    return _fn(*_base);  
}
```

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
char join_iterator::operator*() {  
    return *_inner;  
}
```

```
std::string transform_iterator::operator*() {  
    return _fn(*_base);  
}
```

```
char read_file_iterator::operator*() {  
    return *_buffer_cur;  
}
```

Iterator increment

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
auto& join_iterator::operator++() {  
    if (++_inner == stdr::end(*_outer)) {  
        do { ++_outer; } while (stdr::empty(*_outer));  
        _inner = stdr::begin(*_outer);  
    }  
}
```

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
auto& join_iterator::operator++() {  
    if (++_inner == stdr::end(*_outer)) {  
        do { ++_outer; } while (stdr::empty(*_outer));  
        _inner = stdr::begin(*_outer);  
    }  
}
```

```
auto& transform_iterator::operator++() {  
    ++_base;  
}
```

```
auto rng = read_file(path) | stdv::transform(...) | stdv::join;
```

```
auto& join_iterator::operator++() {  
    if (++_inner == stdr::end(*_outer)) {  
        do { ++_outer; } while (stdr::empty(*_outer));  
        _inner = stdr::begin(*_outer);  
    }  
}
```

```
auto& transform_iterator::operator++() {  
    ++_base;  
}
```

```
auto& read_file_iterator::operator++() {  
    if (++_buffer_cur == _buffer_end)  
        read_more();  
}
```

Imperative code:

- Nested loops.

Iterators:

- State machine.
- Loops split into read and advance.
- Arbitrarily deeply nested sub-state machines.

Imperative code:

- Nested loops.

Iterators have overhead.

Iterators:

- State machine.
- Loops split into read and advance.
- Arbitrarily deeply nested sub-state machines.

- `std::join`: logically two nested loops

- `stdv::join`: logically two nested loops
- `stdv::concat`: logically N loops in sequence

- `stdv::join`: logically two nested loops
- `stdv::concat`: logically N loops in sequence
- `stdv::transform(f) | stdv::filter(p)`: logically one loop with `if`

Why state machines?

Pull model: Range consumer is in control.

Why state machines?

Pull model: Range consumer is in control.

```
auto rng = ...;

auto it = stdr::begin(rng);
use(*it);
++it; // skip
use(*it);
use_again(*it);
++it; // skip
++it; // skip
use(*it);
// stop at this point
```

State machine often unnecessary

```
auto rng = ...;  
stdr::for_each(rng, [&](auto&& x) { ... });
```

State machine often unnecessary

```
auto rng = ...;  
std::for_each(rng, [&](auto&& x) { ... });
```

User does not need control.

State machine often unnecessary

```
auto rng = ...;  
std::for_each(rng, [&](auto&& x) { ... });
```

User does not need control.

Every time a range-based for loop is used:

1. The range is represented by a (complex) state machine.
2. The state machine is repeatedly advanced until completion.

Better: Express the loop directly

Push model: Range producer is in control.

- Range pushes value onto the consumer using a sink function object.
- Consumer processes values as they arrive.

Better: Express the loop directly

Push model: Range producer is in control.

- Range pushes value onto the consumer using a sink function object.
- Consumer processes values as they arrive.

Goal: Direct customization of the entire loop.

- Language proposal: P2881 (rejected)
- Reflection token injection: <https://brevzin.github.io/c++/2025/04/03/token-sequence-for/>

Library solution: `for_each_while` customization point

New customization point: `std2r::for_each_while(rng, sink)`

Iterates over `rng` and calls `sink` for each element while the `sink` returns `true`.

Library solution: `for_each_while` customization point

New customization point: `std2r::for_each_while(rng, sink)`

Iterates over `rng` and calls `sink` for each element while the `sink` returns true.

Default implementation:

```
template <std::input_range Rng, typename Sink>
bool for_each_while(Rng&& rng, Sink s) {
    for (auto&& x : rng) {
        if (!s(x)) return false;
    }
    return true;
}
```

for_each_while customization point implementations

```
bool for_each_while(transform_view& rng, auto s) {  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        return s(rng._fn(x));  
    });  
}
```

```
bool for_each_while(transform_view& rng, auto s) {  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        return s(rng._fn(x));  
    });  
}
```

```
bool for_each_while(join_view& rng, auto s) {  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        return std2r::for_each_while(x, s);  
    });  
}
```

```
bool for_each_while(read_file_view& rng, auto s) {  
    while (true) {  
        std::span<const char> buffer = read_more();  
        if (buffer.empty()) break;  
  
        if (!std2r::for_each_while(buffer, s)) return false;  
    }  
    return true;  
}
```

Loops:

- `stdr::for_each`
- `stdr::copy`, `std::move`, `stdr::to`
- `stdr::transform`
- `stdr::min`, `stdr::max`
- `stdr::count`

Short-circuiting loops:

- `stdr::all_of`, `stdr::any_of`,
`stdr::none_of`
- `stdr::equal`
- `stdr::contains`

Range code with `for_each_while`

```
std2r::for_each_while(read_file(path) | stdv::transform(...) | stdv::join, s);
```

After inlining:

```
return std2r::for_each_while(rng._base, [s](std::string&& str) {  
    return std2r::for_each_while(str, s);  
});
```

Range code with `for_each_while`

```
std2r::for_each_while(read_file(path) | stdv::transform(...) | stdv::join, s);
```

After inlining:

```
return std2r::for_each_while(rng._base, [s](std::string&& str) {  
    for (char c : str) {  
        if (!s(c)) return false;  
    }  
    return true;  
});
```

Range code with `for_each_while`

```
std2r::for_each_while(read_file(path) | stdv::transform(...) | stdv::join, s);
```

After inlining:

```
return std2r::for_each_while(rng._base._base, [s](char c) {  
    for (char translated : fn(x)) {  
        if (!s(translated)) return false;  
    }  
    return true;  
});
```

Range code with `for_each_while`

```
std2r::for_each_while(read_file(path) | stdv::transform(...) | stdv::join, s);
```

After inlining:

```
while (true) {  
    std::span<const char> buffer = read_more();  
    if (buffer.empty()) break;  
  
    for (char c : buffer) {  
        for (char translated : fn(c)) {  
            if (!s(translated)) return false;  
        }  
    }  
}  
  
return true;
```

Range code with `for_each_while`

```
std2r::for_each_while(read_file(path) | stdv::transform(...) | stdv::join, s);
```

After inlining:

```
while (true) {  
    std::span<const char> buffer = read_more();  
    if (buffer.empty()) break;  
  
    for (char c : buffer) {  
        for (char translated : fn(c)) {  
            if (!translated) return false;  
        }  
    }  
}  
  
return true;
```

This is the imperative code!

Why not coroutines?

`std::generator`: Write loop with `co_yield`.

```
std::generator<char> read_file_normalized(std::string_view path) {  
    ...  
    while (true) {  
        std::span<const char> buffer = read_more();  
        if (buffer.empty()) break;  
  
        for (char c : buffer) {  
            for (char translated : fn(c))  
                co_yield translated;  
        }  
    }  
}
```

Coroutines are state machines

- Compiler generates iterator state machine for you.
- Still pull model, not push.
- Additional coroutine overhead (heap allocation)

- Compiler generates iterator state machine for you.
- Still pull model, not push.
- Additional coroutine overhead (heap allocation)

We need better optimizations for `std::generator`.

- Compiler generates iterator state machine for you.
- Still pull model, not push.
- Additional coroutine overhead (heap allocation)

We need better optimizations for `std::generator`.

Natural syntax, bad performance:

`for_each_while` is to `co_yield` what `senders/receivers` is to `co_await`.

The rest of the owl

How to optimize `std::ranges`



1. `for_each_while`
customization point

imgflip.com



2. Adopt it everywhere

The rest of the owl

How to optimize `std::ranges`



1. `for_each_while`
customization point

imgflip.com



2. Adopt it everywhere

- `std::reverse`:
`std::ranges::for_each_while_reversed`?

How to optimize `std::ranges`



1. `for_each_while`
customization point

imgflip.com



2. Adopt it everywhere

- `stdv::reverse:`
`std2r::for_each_while_reversed?`
- `stdv::zip:` Can only use
`std2r::for_each_while` once, all
other ranges use iterators, so which one
should use it?

How to optimize `std::ranges`



1. `for_each_while`
customization point

imgflip.com



2. Adopt it everywhere

- `stdv::reverse:`
`std2r::for_each_while_reversed?`
- `stdv::zip:` Can only use
`std2r::for_each_while` once, all
other ranges use iterators, so which one
should use it?
- `stdr::find and variants:`
`std2r::for_each_iterator_while?`

How to optimize `std::ranges`



1. `for_each_while`
customization point

imgflip.com



2. Adopt it everywhere

- `stdv::reverse`:
`std2r::for_each_while_reversed?`
- `stdv::zip`: Can only use
`std2r::for_each_while` once, all
other ranges use iterators, so which one
should use it?
- `stdr::find` and variants:
`std2r::for_each_iterator_while?`
- **Generators**: Ranges that only support
`std2r::for_each_while` but don't
have iterators?

How to optimize `std::ranges`



1. `for_each_while`
customization point

imgflip.com



2. Adopt it everywhere

- `std::reverse`:
`std::ranges::for_each_while_reversed?`
- `std::zip`: Can only use
`std::ranges::for_each_while` once, all
other ranges use iterators, so which one
should use it?
- `std::find` and variants:
`std::ranges::for_each_iterator_while?`
- **Generators**: Ranges that only support
`std::ranges::for_each_while` but don't
have iterators?
- **Optimizations ignore range-based for
loop**

Chunked ranges

Implementing `std::to<std::string>`

```
rng | std::to<std::string>:
```

```
std::string result;  
if constexpr (std::approximately_sized_range<decltype(rng)>)  
    result.reserve(std::reserve_hint(rng));  
else if constexpr (std::forward_range<decltype(rng)>)  
    result.reserve(std::distance(rng));  
else  
    /* reserve is not possible */;  
std::copy(rng, std::back_inserter(result));
```

```
rng | std::to<std::string>:
```

```
std::string result;  
if constexpr (std::approximately_sized_range<decltype(rng)>)  
    result.reserve(std::reserve_hint(rng));  
else if constexpr (std::forward_range<decltype(rng)>)  
    result.reserve(std::distance(rng));  
else  
    /* reserve is not possible */;  
std::copy(rng, std::back_inserter(result));
```

```
contiguous_range | std::to<std::string>:
```

```
std::string result;  
result.append(std::distance(rng), std::data(rng));
```

Contiguous ranges:

- `std::vector<T>`
- `std::array<T, N>`
- `std::string`
- `std::span<T>`

Ranges with contiguous chunks:

- `std::deque<T>`
- `stdv::concat(vec1, vec2)`
- `range_of_spans | stdv::join`
- `read_file(path)`

Some ranges have natural segmentation into chunks.

How to express this chunking?

We already did!

We already did!

```
bool for_each_while(join_view& rng, auto s) {  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        return std2r::for_each_while(x, s);  
    });  
}
```

We already did!

```
bool for_each_while(join_view& rng, auto s) {  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        return std2r::for_each_while(x, s);  
    });  
}
```

```
bool for_each_while(read_file_view& rng, auto s) {  
    while (true) {  
        std::span<const char> buffer = read_more();  
        if (buffer.empty()) break;  
  
        if (!std2r::for_each_while(buffer, s)) return false;  
    }  
    return true;  
}
```

`std::for_each_while(rng, s)` is expression-equivalent to:

1. `s.chunk(rng)` if that is well-formed, otherwise
2. ADL-based `for_each_while(rng, s)` overload if one exists, otherwise

```
3. for (auto&& x : rng) {  
    if (!s(x)) return false;  
}  
return true;
```

`std::for_each_while(rng, s)` is expression-equivalent to:

1. `s.chunk(rng)` if that is well-formed, otherwise
2. ADL-based `for_each_while(rng, s)` overload if one exists, otherwise

```
3. for (auto&& x : rng) {  
    if (!s(x)) return false;  
}  
return true;
```

Crucially: Sink gets to customize first, before the range!

Implementing a sink for string appending

```
struct string_appender_sink {  
    std::string& _result;  
  
    bool operator()(char c) const {  
        _result.push_back(c);  
        return true;  
    }  
  
    bool chunk(std::contiguous_range auto&& chunk) const {  
        _result.append(  
            stdr::distance(chunk),  
            stdr::data(chunk)  
        );  
        return true;  
    }  
};
```

Implementing `std::to<std::string>` more efficiently

```
rng | std::to<std::string>:
```

```
std::string result;
```

```
if constexpr (std::approximately_sized_range<decltype(rng)>)
```

```
    result.reserve(std::reserve_hint(rng));
```

```
else if constexpr (std::forward_range<decltype(rng)>)
```

```
    result.reserve(std::distance(rng));
```

```
else
```

```
    /* reserve is not possible */;
```

```
std::for_each_while(rng, string_appender_sink{result});
```

Using the optimized `std::to<std::string>`

```
std::for_each_while(  
    range_of_spans | std::join,  
    string_appender_sink{result}  
);
```

After inlining:

```
return std2r::for_each_while(rng._base, [s](std::span<const char> x) {  
    return std2r::for_each_while(x, s);  
});
```

Using the optimized `stdr::to<std::string>`

```
stdr::for_each_while(  
    range_of_spans | stdv::join,  
    string_appender_sink{result}  
);
```

After inlining:

```
return std2r::for_each_while(rng._base, [s](std::span<const char> x) {  
    return s.chunk(x);  
});
```

Sinks instead of output iterators?

Output iterators:

- Clunky overloads of `operator++`, `operator*`
- No ability to receive chunks

Sinks:

- Only overload `operator()`
- Can receive chunks

Sinks instead of output iterators?

Output iterators:

- Clunky overloads of `operator++`, `operator*`
- No ability to receive chunks

Sinks:

- Only overload `operator()`
- Can receive chunks

The standard library should support the output iterator interface only for backwards compatibility and also allow sinks.

```
std2r::copy(rng, std2r::append_to(container));
```

Pipelines should express chunking

```
std::string read_file_normalized(std::string_view path) {  
    return read_file(path) // big contiguous chunks  
        | stdv::transform([](char c) {  
            if (c == '\t')  
                return std::string(4, ' ');  
            else  
                return std::string(1, c);  
        }) // not contiguous  
        | stdv::join // tiny contiguous chunks  
        | stdr::to<std::string>();  
}
```

Pipelines should express chunking

```
std::string read_file_normalized(std::string_view path) {  
    return read_file(path)           // big contiguous chunks  
        | stdv::split("\t"sv)        // not contiguous  
        | stdv::join_with("    "sv) // big contiguous chunks  
        | stdr::to<std::string>();  
}
```

```
bool for_each_while(join_with_view& rng, auto s) {  
    auto first = true;  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        if (first)  
            first = false;  
        else  
            if (!std2r::for_each_while(rng._separator, s)) return false;  
  
        return std2r::for_each_while(x, s);  
    });  
}
```

Except it doesn't work

```
std::string read_file_normalized(std::string_view path) {  
    return read_file(path)  
        | stdv::split("\t"sv)  
        | stdv::join_with("    "sv)  
        | stdr::to<std::string>();  
}
```

Except it doesn't work

```
std::string read_file_normalized(std::string_view path) {  
    return read_file(path)  
        | stdv::split("\t"sv)  
        | stdv::join_with(" "sv)  
        | stdr::to<std::string>();  
}
```

But: `stdv::split` requires a forward range, `read_file` is input!

(And `stdv::lazy_split` does not keep the contiguous chunks property.)

Except it doesn't work

```
std::string read_file_normalized(std::string_view path) {  
    return read_file(path)  
        | stdv::split("\t"sv)  
        | stdv::join_with(" "sv)  
        | stdr::to<std::string>();  
}
```

But: `stdv::split` requires a forward range, `read_file` is input!

(And `stdv::lazy_split` does not keep the contiguous chunks property.)

However: We are fine with treating each contiguous chunk separately.

Pipelines should express chunking completely

```
// Returns a range of `std::span<const char>`.  
std::input_range auto read_file_buffers(std::string_view path);  
  
std::input_range auto read_file(std::string_view path)  
{  
    return read_file_buffers(path) | stdv::join;  
}
```

Pipelines should express chunking completely

```
// Returns a range of `std::span<const char>`.
std::input_range auto read_file_buffers(std::string_view path);

std::input_range auto read_file(std::string_view path)
{
    return read_file_buffers(path) | stdv::join;
}

std::string read_file_normalized(std::string_view path) {
    return read_file_buffers(path)
        | stdv::transform([](std::span<const char> chunk) {
            return chunk | stdv::split("\t"sv) | stdv::join_with("    "sv);
        });
    | stdv::join | stdr::to<std::string>();
}
```

A generator implementation of read_file_buffers

```
class read_file_buffers_generator {
    read_file_range _base;

    template <typename Sink>
    struct sink_adaptor {
        Sink s;

        bool chunk(std::span<const char> chunk) const {
            return s(chunk);
        }
    };

    friend bool for_each_while(read_file_buffers_generator& gen, auto s) {
        return stdr::for_each_while(gen._base, sink_adaptor{s});
    }
};
```

Problem #2

Problem: Describe SQL schema in a compile-time DSL and generate SQL statements.

```
using People = Table<
    "people"_tc,
    Column<"id"_tc, "INTEGER PRIMARY KEY"_tc>,
    Column<"name"_tc, "TEXT NOT NULL"_tc>
>;
```

```
constexpr std::string_view stmt = People::Create();
static_assert(stmt
    == "CREATE TABLE people(id INTEGER PRIMARY KEY, name TEXT NOT NULL)"
);
```

Fundamental C++ Limitation: Within a `constexpr` function, arguments aren't `constexpr`.

```
std::array<char, constexpr_strlen("CREATE")> array; // okay
```

```
constexpr void do_sth(const char* str)
{
    std::array<char, constexpr_strlen(str)> array; // error
}
```

Compile-time vs. “compile-time”

Fundamental C++ Limitation: Within a `constexpr` function, arguments aren't `constexpr`.

```
std::array<char, constexpr_strlen("CREATE")> array; // okay
```

```
constexpr void do_sth(const char* str)
{
    std::array<char, constexpr_strlen(str)> array; // error
}
```

Solution: Embed the relevant information in the type of the argument.

Convention: “early compile-time” vs “late compile-time”.

Size of string literal ranges

```
stdr::size("abc")
```

Size of string literal ranges

```
stdr::size("abc")
```

4

Size of string literal ranges

```
stdr::size("abc")
```

4

Solution: Wrap string literals in a custom type.

```
template <typename T, T ... Ts>
struct literal_range {
    static constexpr const T* begin();
    static constexpr const T* end();
};
```

```
template <string_template_param String>
constexpr auto operator""_tc /* _tc == think-cell */ () {
    auto [...Is] = std::make_index_sequence<String.size()>{};
    return literal_range<typename decltype(String)::char_type, String[Is]...>{};
}
```

- `std::size("abc"_tc)` is 3
- Size of `literal_range` parameter is a compile-time expression

Compile-time sized ranges

Concatenating fixed strings

```
template <auto type, auto name>
struct Object {
    static constexpr std::string_view Create() {
        return "CREATE " + type + " " + name; // pseudo code
    }
};
```

```
template <auto type, auto name>
struct Object {
    static constexpr std::string_view Create() {
        return "CREATE " + type + " " + name; // pseudo code
    }
};
```

Wishlist: Just use range algorithms.

```
static constexpr std::string_view Create() {
    static constexpr auto result
        = std::concat("CREATE "_tc, type, " "_tc, name)
        | std::to<std::array>();
    return result;
}
```

std2r::to<std::array>() needs the size early

```
template <template <typename, std::size_t> class Container, typename Rng>
auto to(Rng&& rng)
    -> Container<std::range_value_t<Rng>, std::size(rng)> {
    ...
}
```

Unfortunately, `rng` is a parameter which is not `constexpr` inside the body, so this doesn't work.

std2r::to<std::array>() needs the size early

```
template <template <typename, std::size_t> class Container, typename Rng>
auto to(Rng&& rng)
    -> Container<std::range_value_t<Rng>, std::size(rng)> {
    ...
}
```

Unfortunately, rng is a parameter which is not constexpr inside the body, so this doesn't work.

... or does it?

P2280: Using unknown references in constant expressions | think-cell

P2280 (adopted in C++23) makes it just work:

 rhh6e31E6

```
auto rng = std::concat("abc"_tc, std::array<char, 3>{'d', 'e', 'f'});  
static_assert(std::size(rng) == 6);
```

Essentially: as long as you don't access runtime values, you can use local objects in constexpr!

P2280: Using unknown references in constant expressions | think-cell

P2280 (adopted in C++23) makes it just work:

 rhh6e31E6

```
auto rng = std::concat("abc"_tc, std::array<char, 3>{'d', 'e', 'f'});  
static_assert(std::size(rng) == 6);
```

Essentially: as long as you don't access runtime values, you can use local objects in constexpr!

```
template <typename T, T ... Ts>  
constexpr std::size_t literal_range<T, Ts...>::size() const {  
    return sizeof...(Ts);  
}  
  
template <typename ... Rng>  
constexpr std::size_t concat_view<Rng...>::size() const {  
    auto [...Is] = std::make_index_sequence<sizeof...(Rng)>{};  
    return (std::size(std::get<Is>(_base)) + ...);  
}
```

New concept: Compile-time sized ranges

```
template <typename T>
concept constexpr_sized_range = stdr::sized_range<T> && requires(T& t) {
    typename std::constant_wrapper<stdr::size(t)>;
};
```

Or `compile_time_sized_range`, `statically_sized_range`, `fixed_sized_range`, ...

New concept: Compile-time sized ranges

```
template <typename T>
concept constexpr_sized_range = stdr::sized_range<T> && requires(T& t) {
    typename std::constant_wrapper<stdr::size(t)>;
};
```

Or compile_time_sized_range, statically_sized_range, fixed_sized_range, ...

```
template <constexpr_sized_range Rng>
constexpr auto constexpr_size = stdr::size(std::declval<Rng>());
```

Or compile_time_size, static_size, fixed_size, ...

New concept: Compile-time sized ranges

```
template <typename T>
concept constexpr_sized_range = stdr::sized_range<T> && requires(T& t) {
    typename std::constant_wrapper<stdr::size(t)>;
};
```

Or compile_time_sized_range, statically_sized_range, fixed_sized_range, ...

```
template <constexpr_sized_range Rng>
constexpr auto constexpr_size = decltype([](Rng& rng) {
    return std::cw<std::ranges::size(rng)>;
})(std::declval<Rng&>()))::value;
```

Or compile_time_size, static_size, fixed_size, ...

Compute a constant expression involving unknown references without providing the references.

```
template <auto Fn, typename ... T>
constexpr auto constexpr_declval = decltype([](T&&... args) {
    return std::cw<Fn(args...)>;
})(std::declval<T>()...))::value;
```

Compute a constant expression involving unknown references without providing the references.

```
template <auto Fn, typename ... T>
constexpr auto constexpr_declval = decltype([](T&&... args) {
    return std::cw<Fn(args...)>;
})(std::declval<T>()...))::value;
```

```
template <constexpr_sized_range Rng>
constexpr auto constexpr_size = constexpr_declval<std::size, Rng>;
```

```
template <typename Rng>
constexpr auto constexpr_size = nullptr;

template <typename T, std::size_t N>
constexpr auto constexpr_size<std::array<T, N>> = N;

template <typename Rng>
    requires requires { decltype(std::declval<Rng&>().size())::value; }
constexpr auto constexpr_size<Rng>
    = decltype(std::declval<Rng&>().size())::value;
```

See: Compile-time sizes for range adaptors —

www.think-cell.com/en/career/devblog/compile-time-sizes-for-range-adaptors

Implementing `std2r::to<std::array>()`

```
template <template <typename, std::size_t> class Container, typename Rng>
auto to(Rng&& rng) {
    Container<stdr::range_value_t<Rng>, stdr::size(rng)> result;
    stdr::copy(rng, ptr_appender(result.data()));
    return result;
}
```

Implementing `std2r::to<std::array>()`

```
template <template <typename, std::size_t> class Container, typename Rng>
auto to(Rng&& rng) {
    auto [...Is] = std::make_index_sequence<stdr::size(rng)>{};
    auto it = stdr::begin(rng);
    return Container<stdr::range_value_t<Rng>, stdr::size(rng)>{it[Is]...}{};
}
```

```
template <auto type, auto name>
struct Object {
    static constexpr std::string_view Create() {
        static constexpr auto result
            = stdv::concat("CREATE "_tc, type, " "_tc, name)
              | std2r::to<std::array>();
        return result;
    }
};
```

Heterogeneous generators

Concatening fixed strings with separator

```
template <auto type, auto name>
struct Object {
    static constexpr std::string_view Create() {
        static constexpr auto result
            = stdv::concat("CREATE "_tc, type, " "_tc, name)
              | std2r::to<std::array>();
        return result;
    }
};
```

Concatening fixed strings with separator

```
template <auto type, auto name>
struct Object {
    static constexpr std::string_view Create() {
        static constexpr auto result
            = concat_with(" "_tc, "CREATE"_tc, type, name)
              | std2r::to<std::array>();
        return result;
    }
};
```

Concatening fixed strings with separator

```
template <auto type, auto name>
struct Object {
    static constexpr std::string_view Create() {
        static constexpr auto result
            = concat_with(" "_tc, "CREATE"_tc, type, name)
              | std2r::to<std::array>();
        return result;
    }
};
```

```
auto concat_with(auto&& rngSep, auto&& rng0, auto&&... rngs) {
    return stdv::concat(
        rng0, stdv::concat(rngSep, rngs)...
    );
}
```

Concatening fixed strings with separator

```
template <auto qualifiers, auto type, auto name>
struct Object {
    static constexpr std::string_view Create() {
        static constexpr auto result
            = concat_with(" "_tc, "CREATE"_tc, qualifiers, type, name)
              | std2r::to<std::array>();
        return result;
    }
};
```

Concatenating fixed strings with separator

```
template <auto qualifiers, auto type, auto name>
struct Object {
    static constexpr std::string_view Create() {
        static constexpr auto result
            = concat_nonempty_with(" "_tc, "CREATE"_tc, qualifiers, type, name)
            | std2r::to<std::array>();
        return result;
    }
};
```

Concatenating non-empty ranges with separator

```
auto concat_nonempty_with(auto&& rngSep, auto&&... rngs) {  
    return make_range(rngs...)  
        | stdv::filter([](auto&& rng) {  
            return !stdr::empty(rng);  
        })  
        | stdv::join_with(rngSep);  
}
```

1. Turn the parameter pack into a range.
2. Filter out empty ranges.
3. Join the remaining ranges with the separator.

Concatenating non-empty ranges with separator

```
auto concat_nonempty_with(auto&& rngSep, auto&&... rngs) {  
    return [&](auto&&... non_empty_rngs) {  
        return concat_with(rngSep, non_empty_rngs...);  
    }(/* filter out empty ranges from rngs */...);  
}
```

Concatenating non-empty ranges with separator

```
auto concat_nonempty_with(auto&& rngSep, auto&&... rngs) {  
    return make_range(rngs...) |  
        stdv::filter([](auto&& rng) {  
            return !stdr::empty(rng);  
        }) |  
        stdv::join_with(rngSep);  
}
```

1. Turn the parameter pack into a range.
2. Filter out empty ranges.
3. Join the remaining ranges with the separator.

Implementing make_range

```
template <typename... Rngs>
auto make_range(Rngs&&... rngs) {
    using rng_t = std::common_type_t<stdv::all_t<Rngs>...>;
    return std::array<rng_t, sizeof...(rngs)>{stdv::all(rngs)...};
}
```

```
template <typename... Rngs>
auto make_range(Rngs&&... rngs) {
    using rng_t = std::common_type_t<stdv::all_t<Rngs>...>;
    return std::array<rng_t, sizeof...(rngs)>{stdv::all(rngs)...};
}
```

Problem: Each element of the resulting range has the same type.

```
template <typename... Rngs>
auto make_range(Rngs&&... rngs) {
    using rng_t = std::common_type_t<stdv::all_t<Rngs>...>;
    return std::array<rng_t, sizeof...(rngs)>{stdv::all(rngs)...};
}
```

Problem: Each element of the resulting range has the same type.

```
std::array<std::span<const char>, 3> result
    = make_range("CREATE"_tc, "TABLE"_tc, "people"_tc);
constexpr auto size = stdr::size(result[0]); // error: not a constant expression
```

Better implementation of make_range

```
template <typename ... Rng>
auto make_range(Rng&&... rngs) {
    return std::make_tuple(stdv::all(std::forward<Rng>(rngs))...);
}
```

Better implementation of make_range

```
template <typename ... Rng>
auto make_range(Rng&&... rngs) {
    return std::make_tuple(stdv::all(std::forward<Rng>(rngs))...);
}
```

But a tuple isn't a range!

```
std::tuple<int, float, char> tuple;  
auto x = stdr::begin(tuple)[n];  
// What is the type of x?
```

```
std::tuple<int, float, char> tuple;  
auto x = std::begin(tuple)[n];  
// What is the type of x?
```

```
template <typename ... T>  
auto std::tuple<T...>::iterator::operator*() const  
    -> std::variant<T...>  
{  
    ...  
}
```

Overhead.

```
std::tuple<int, float, char> tuple;  
auto x = std::begin(tuple)[n];  
// What is the type of x?
```

```
template <typename ... T, typename Sink>  
bool for_each_while(std::tuple<T...>& tuple, Sink s) {  
    auto [...Is] = std::make_index_sequence<sizeof...(T)>{};  
    return (s(std::get<Is>(tuple)) && ...);  
}
```

No overhead!

Homogeneous ranges:

- `std::begin/std::end` (and optionally `for_each_while`)
- One common `std::range_value_t` type
- Can use all iterator-based algorithms/views, potentially optimized with `for_each_while`
- Can use range-based `for` loop

Heterogeneous generators:

- Only `for_each_while`, no `std::begin/std::end`
- Variadic `generator_output_t` type list
- Can only use `for_each_while`-based algorithm/views
- Can only use `for_each_while` with a generic lambda

Every homogeneous range is-a heterogeneous generator, but not vice versa.

`std::tuple:`

```
std2r::for_each_while(tuple, [](auto&& x) {  
    // do something with `x`  
});
```

`std::tuple:`

```
std2r::for_each_while(tuple, [](auto&& x) {  
    // do something with `x`  
});
```

`std::index_sequence:`

```
std2r::for_each_while(seq, []<std::size_t I>(std::constant_wrapper<I>) {  
    // do something with `I`  
});
```

`std::tuple:`

```
std2r::for_each_while(tuple, [](auto&& x) {  
    // do something with `x`  
});
```

`std::index_sequence:`

```
std2r::for_each_while(seq, []<std::size_t I>(std::constant_wrapper<I>) {  
    // do something with `I`  
});
```

`boost::mp11::mp_list:`

```
std2r::for_each_while(type_list, []<typename T>(std::type_identity<T>) {  
    // do something with `T`  
});
```

Goal: Compile each opcode of Z80 machine code into a function that executes it.

```
using Z80Func = void (*)(Z80 &);

constexpr std::array<Z80Func*, 256> compiled_opcodes =
    stdv::iota(0, 256)
    | stdv::transform([](auto opcode) {
        constexpr Instruction decoded_instr = decode(opcode);
        return compile(decoded_instr);
    })
    | stdr::to<std::array>{};
```

Not valid C++!

Goal: Compile each opcode of Z80 machine code into a function that executes it.

```
using Z80Func = void (*)(Z80&);

constexpr std::array<Z80Func*, 256> compiled_opcodes =
    std::make_integer_sequence<unsigned char, 256>{}
    | std2v::transform([]<unsigned char Byte>(std::constant_wrapper<Byte>) {
        constexpr Instruction decoded_instr = decode(Byte);
        return compile<decoded_instr>();
    })
    | std2r::to<std::array>{};
```

Valid C++!

Goal: Compile each opcode of Z80 machine code into a function that executes it.

```
using Z80Func = void(*)(Z80&);

constexpr std::array<Z80Func*, 256> compiled_opcodes =
    std::make_integer_sequence<unsigned char, 256>{}
    | std2v::transform([]<unsigned char Byte>(std::constant_wrapper<Byte>) {
        return +[](Z80& z80) {
            return instructions<Opcode(Byte)>.execute(z80);
        };
    })
    | std2r::to<std::array>{};
```

Valid C++!

```
bool for_each_while(filter_view& rng, auto s) {  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        return rng._predicate(x) ? s(x) : true;  
    });  
}
```

```
bool for_each_while(filter_view& rng, auto s) {  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        if constexpr (requires { std::cw<rng._predicate(x)>; }) {  
            if constexpr (rng._predicate(x)) {  
                return s(x);  
            } else {  
                return true;  
            }  
        } else {  
            return rng._predicate(x) ? s(x) : true;  
        }  
    });  
}
```

Using filtering based on type

```
std::make_tuple(1, 2.0, "hello")
| std2v::filter([](auto x) {
    return std::is_same_v<decltype(x), int>;
})
| stdr::to<std::vector>(); // std::vector<int>
```

Concatenating non-empty ranges with separator

```
auto concat_nonempty_with(auto&& rngSep, auto&&... rngs) {  
    return make_range(rngs...)  
        // compile-time sized with compile-time sized elements  
        | stdv::filter([](auto&& rng) {  
            return !stdr::empty(rng);  
        })  
        | stdv::join_with(rngSep);  
}
```

Tuple Generators

Concatenating non-empty ranges with separator

```
auto concat_nonempty_with(auto&& rngSep, auto&&... rngs) {  
    return make_range(rngs...)  
        // compile-time sized with compile-time sized elements  
        | stdv::filter([](auto&& rng) {  
            return !stdr::empty(rng);  
        }) // no longer compile-time sized  
        | stdv::join_with(rngSep);  
}
```

```
template <typename Base, typename Predicate>
std::size_t filter_view<Base, Predicate>::size() const {
    return stdr::count_if(_base, _predicate);
}
```

```
template <typename Base, typename Predicate>
    requires std2r::constexpr_sized_range<Base>
std::size_t filter_view<Base, Predicate>::size() const {
    return stdr::count_if(_base, _predicate); // O(1)
}
```

```
template <typename Base, typename Predicate>
std::size_t filter_view<Base, Predicate>::size() const
    requires std2r::constexpr_sized_range<Base>
        && (stdr::all_of(std2r::generator_output_t<Base>{},
            []<typename T>(std::type_identity<T>) {
                return requires(Predicate p, T t) { std::cw<p(t)>; };
            })))
{
    return stdr::count_if(_base, _predicate); // 0(1), hopefully optimized
}
```

Pre-C++23:

```
template <typename Base, typename Predicate>
    requires std2r::constexpr_sized_range<Base>
    && (stdr::all_of(std2r::generator_output_t<Base>{},
        []<typename T>(std::type_identity<T>) {
            return requires(Predicate p, T t) { std::cw<p(t)>; };
        })))
constexpr auto constexpr_size<filter_view<Base, Predicate>>
    = stdr::count_if(std::declval<Base>(), std::declval<Predicate>()); // ???
```

Conditional range and filter

```
auto gen1 = std::make_tuple(1, 2, 3);  
static_assert(stdr::size(gen1) == 3);  
  
auto gen2 = std::make_tuple(1.0, 2.0, 3.0);  
static_assert(stdr::size(gen2) == 3);
```

```
auto gen1 = std::make_tuple(1, 2, 3);  
static_assert(stdr::size(gen1) == 3);  
  
auto gen2 = std::make_tuple(1.0, 2.0, 3.0);  
static_assert(stdr::size(gen2) == 3);  
  
auto conditional = std2v::conditional_range(getchar() % 2 == 0, gen1, gen2);  
static_assert(stdr::size(conditional) == 3);
```

```
auto gen1 = std::make_tuple(1, 2, 3);  
static_assert(stdr::size(gen1) == 3);
```

```
auto gen2 = std::make_tuple(1.0, 2.0, 3.0);  
static_assert(stdr::size(gen2) == 3);
```

```
auto conditional = std2v::conditional_range(getchar() % 2 == 0, gen1, gen2);  
static_assert(stdr::size(conditional) == 3);
```

```
auto filtered = conditional  
    | std2v::filter([](auto x) { return std::is_same_v<decltype(x), int>; });  
static_assert(stdr::size(filtered) == ???); // error: not a constant expression
```

`std::filter(rng, p)` has a compile-time size if:

1. `rng` has a compile-time size
2. `p` depends only on the type of the elements in `rng`
3. We can get the type of each element at compile-time.

New concept: Tuple generators

Types that are:

1. Generators: `std2r::for_each_while(gen, sink)`
2. Compile-time sized: `std::cw<stdr::size(gen)>`
3. Tuple-like: `std2r::get<Idx>(gen)`

New concept: Tuple generators

Types that are:

1. Generators: `std2r::for_each_while(gen, sink)`
2. Compile-time sized: `std::cw<stdr::size(gen)>`
3. Tuple-like: `std2r::get<Idx>(gen)`

Most views of tuple-like generators can themselves become tuple-like generators.

Generators:

```
std2r::for_each_while(gen, [](auto&& x) {  
    ...  
});
```

Tuple generators:

```
template for (auto&& x : gen) {  
    ...  
}
```

Ranges:

```
for (auto&& x : rng) {  
    ...  
}
```

Sized filter view

```
template <typename Base, typename Predicate>
std::size_t filter_view<Base, Predicate>::size() const
    requires std2r::tuple_generator<Base>
        && (stdr::all_of(std2r::generator_output_t<Base>{},
            []<typename T>(std::type_identity<T>) {
                return requires(Predicate p, T t) { std::cw<p(t)>; };
            })))
{
    return constexpr_declval<[]>(Base b, Predicate p) {
        return std2r::count_if(std::make_index_sequence<stdr::size(b)>{},
            [&]<std::size_t I>(std::constant_wrapper<I>) {
                return p(std2r::get<I>(b));
            }
        );
    }, Base, Predicate>;
}
```

```
template <std::size_t I, typename Base, typename Predicate>
decltype(auto) get(filter_view<Base, Predicate>& rng) {
    constexpr auto base_i
        = /* index accounting for skipped elements */;
    return std2r::get<base_i>(rng._base);
}
```

Concatenating non-empty ranges with separator

```
auto concat_nonempty_with(auto&& rngSep, auto&&... rngs) {  
    return make_range(rngs...)  
        | std2v::filter([](auto&& rng) {  
            return !stdr::empty(rng);  
        })  
        // compile-time sized with compile-time sized elements  
        | stdv::join_with(rngSep);  
        // no longer compile-time sized  
}
```

```
template <typename RngRng>
    requires stdr::sized_range<RngRng>
           && all_same_constexpr_size<stdr::generator_output_t<RngRng>>
std::size_t join_view<RngRng>::size() const
{
    auto common_constexpr_size = ...;
    return stdr::size(_base) * common_constexpr_size;
}
```

```
template <typename RngRng>
    requires stdr::sized_range<RngRng>
        && (std2r::tuple_generator<RngRng>
            || all_same_constexpr_size<std2r::generator_output_t<RngRng>>)
std::size_t join_view<RngRng>::size() const
{
    if constexpr (std2r::tuple_generator<RngRng>) {
        auto [...Is] = std::make_index_sequence<stdr::size(_base)>{};
        return (stdr::size(std2r::get<Is>(_base)) + ...);
    } else {
        auto common_constexpr_size = ...;
        return stdr::size(_base) * common_constexpr_size;
    }
}
```

```
template <typename RngRng>
    requires stdr::sized_range<RngRng>
    && std2r::tuple_generator<RngRng>
std::size_t join_view<RngRng>::size() const
{
    auto [...Is] = std::make_index_sequence<stdr::size(_base)>{};
    return (stdr::size(std2r::get<Is>(_base)) + ...);
}
```

```
template <typename RngRng>
    requires stdr::sized_range<RngRng>
    && std2r::tuple_generator<RngRng>
std::size_t join_view<RngRng>::size() const
{
    auto [...Is] = std::make_index_sequence<stdr::size(_base)>{};
    return (stdr::size(std2r::get<Is>(_base)) + ...);
}
```

```
template <typename ... Rng>
std::size_t concat_view<Rng...>::size() const {
    auto [...Is] = std::make_index_sequence<sizeof...(Rng)>{};
    return (stdr::size(std::get<Is>(_base)) + ...);
}
```

```
bool for_each_while(join_view& rng, auto s) {  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        return std2r::for_each_while(x, s);  
    });  
}
```

```
bool for_each_while(join_view& rng, auto s) {  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        return std2r::for_each_while(x, s);  
    });  
}
```

```
bool for_each_while(concat_view& rng, auto s) {  
    return std2r::for_each_while(rng._base, [s](auto&& x) {  
        return std2r::for_each_while(x, s);  
    });  
}
```

concat is just a join of a tuple!

concat is just a join of a tuple!

```
template <typename ... Rng>
auto concat(Rng&&... rng) {
    return std2v::join(std::make_tuple(stdv::all(rng)...));
}
```

- tuple-based join_view that corresponds to concat_view
- range-based join_view that corresponds to traditional join_view
- generator join_view that is shared and just implements for_each_while

Concatenating non-empty ranges with separator

```
auto concat_nonempty_with(auto&& rngSep, auto&&... rngs) {  
    return [&](auto&&... non_empty_rngs) {  
        return concat_with(rngSep, non_empty_rngs...);  
    }(/* filter out empty ranges from rngs */...);  
}
```

```
auto concat_nonempty_with(auto&& rngSep, auto&&... rngs) {  
    return make_range(rngs...)  
        | std2v::filter([](auto&& rng) {  
            return !std::empty(rng);  
        })  
        | std2v::join_with(rngSep);  
}
```

Concatenating non-empty ranges with separator

```
auto concat_nonempty_with(auto&& rngSep, auto&&... rngs) {  
    return [&](auto&&... non_empty_rngs) {  
        return concat_with(rngSep, non_empty_rngs...);  
    }(/* filter out empty ranges from rngs */...);  
}
```

```
auto concat_nonempty_with(auto&& rngSep, auto&&... rngs) {  
    return make_range(rngs...) |  
        std2v::filter([&](auto&& rng) {  
            return !std::empty(rng);  
        }) |  
        std2v::join_with(rngSep);  
}
```

They're the same algorithm!

Library infrastructure updates:

84 files changed, 969 insertions(+), 730 deletions(-)

Library infrastructure updates:

84 files changed, 969 insertions(+), 730 deletions(-)

Make `concat_nonempty_with` work:

```
template<typename RngSep, typename... Rngs>  
-   auto concat_nonempty_with(RngSep&& rngSep, Rngs&&... rngs) {  
+   constexpr auto concat_nonempty_with(RngSep&& rngSep, Rngs&&... rngs) {
```

Conclusion

What should be included in `std::ranges`

- `std2v::conditional_range`

What should be included in `std::ranges`

- `std2v::conditional_range`
- `std2r::for_each_while` customization point as a hidden optimization

What should be included in `std::ranges`

- `std2v::conditional_range`
- `std2r::for_each_while` customization point as a hidden optimization
- Sink-based appenders as better output iterators

What should be included in `std::ranges`

- `std2v::conditional_range`
- `std2r::for_each_while` customization point as a hidden optimization
- Sink-based appenders as better output iterators
- Compile-time sized ranges, `std2r::to<std::array>`

(Heterogeneous) Generators are probably not a good fit | think-cell

(Heterogeneous) Generators are a wild departure from existing range concepts:

- No iterators.
- No (regular) for loop.
- Not a single value type.

(Heterogeneous) Generators are probably not a good fit | think-cell

(Heterogeneous) Generators are a wild departure from existing range concepts:

- No iterators.
- No (regular) for loop.
- Not a single value type.

Can be somewhat emulated using a range of `std::meta::info`.

Proposals welcome!

We're hiring: think-cell.com/career/dev

Bonus

```
int fib(int n) { ... }

int smallest_fib_above(int x) {
    auto all_fibonacci_numbers =
        stdv::iota(0)           // [0, 1, 2, 3, ...]
        | stdv::transform(fib); // [0, 1, 1, 2, ...]

    return *stdr::find_if(
        all_fibonacci_numbers,
        [x](int f) { return f > x; }
    );
}
```

[iterator.requirements.general]/8

Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A range is an iterator and a sentinel that designate the beginning and end of the computation [...]

[iterator.requirements.general]/8

Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A range is an iterator and a sentinel that designate the beginning and end of the computation [...]

[iterator.requirements.general]/9

An iterator and a sentinel denoting a range are comparable. A range $[i, s)$ is empty if $i == s$; otherwise, $[i, s)$ refers to the elements in the data structure starting with the element pointed to by i and up to but not including the element, if any, pointed to by the first iterator j such that $j == s$.

[iterator.requirements.general]/10

A sentinel s is called reachable from an iterator i if and only if there is a finite sequence of applications of the expression $++i$ that makes $i == s$.

[iterator.requirements.general]/10

*A sentinel s is called reachable from an iterator i if and only if there is a **finite sequence** of applications of the expression $++i$ that makes $i == s$. [...]*

[iterator.requirements.general]/10

*A sentinel s is called reachable from an iterator i if and only if there is a **finite sequence** of applications of the expression $++i$ that makes $i == s$. [...]*

```
auto rng = std::iota(0u);
auto i = rng.begin();
auto s = rng.end();
for (;;)
{
    assert(i != s);
    ++i;
}
```

[iterator.requirements.general]/10

*A sentinel s is called reachable from an iterator i if and only if there is a **finite sequence** of applications of the expression $++i$ that makes $i == s$. [...]*

```
auto rng = std::iota(0u);
auto i = rng.begin();
auto s = rng.end();
for (;;)
{
    assert(i != s);
    ++i;
}
```

`decltype(s)` is `std::unreachable_sentinel_t`.

[iterator.requirements.general]/10

[...] If s is reachable from i , $[i, s)$ denotes a valid range.

[iterator.requirements.general]/10

[...] If s is reachable from i , $[i, s)$ denotes a valid range.

Hmmm...

[iterator.requirements.general]/10

[...] If s is reachable from i , $[i, s)$ denotes a valid range.

Hmmm...

[iterator.requirements.general]/12

The result of the application of library functions to invalid ranges is undefined.

[iterator.requirements.general]/10

[...] If s is reachable from i , $[i, s)$ denotes a valid range.

Hmmm...

[iterator.requirements.general]/12

The result of the application of library functions to invalid ranges is undefined.

Oh.

```
auto rng = stdv::iota(0u);  
stdr::find_if(rng, p); // UB
```

```
auto rng = stdv::iota(0u);  
stdr::find_if(rng, p); // UB
```

```
auto rng = stdv::iota(0u);  
rng | stdv::take(10); // UB
```

```
auto rng = stdv::iota(0u);  
stdr::find_if(rng, p); // UB
```

```
auto rng = stdv::iota(0u);  
rng | stdv::take(10); // UB
```

```
auto rng = stdv::iota(0u);  
for (auto x : rng) { /* ... */ } // UB (calls `begin()`)
```

```
auto rng = stdv::iota(0u);  
stdr::find_if(rng, p); // UB
```

```
auto rng = stdv::iota(0u);  
rng | stdv::take(10); // UB
```

```
auto rng = stdv::iota(0u);  
for (auto x : rng) { /* ... */ } // UB (calls `.begin()`)
```

```
auto rng = stdv::iota(0u); // UB (calls `~iota_view()`)
```

This is undesirable

The standard library should not provide facilities that only cause undefined behavior.

This is undesirable

The standard library should not provide facilities that only cause undefined behavior.

SF	F	N	A	SA
9	4	1	0	0

Actual poll had different wording.

This is undesirable

The standard library should not provide facilities that only cause undefined behavior.

SF	F	N	A	SA
9	4	1	0	0

Actual poll had different wording.

But how to fix it?

True invalid ranges:

- `stdr::subrange(array + 10, array)`
- `stdr::subrange(vec1.begin(), vec2.end())`

True invalid ranges:

- `stdr::subrange(array + 10, array)`
- `stdr::subrange(vec1.begin(), vec2.end())`

True infinite ranges:

- `stdv::repeat(42)`
- `stdv::iota(unsigned(0))`

True invalid ranges:

- `stdr::subrange(array + 10, array)`
- `stdr::subrange(vec1.begin(), vec2.end())`

True infinite ranges:

- `stdv::repeat(42)`
- `stdv::iota(unsigned(0))`

Ranges where `.end()` lies:

- `stdv::iota(int(0))`
- `stdr::subrange(ptr, stdr::unreachable_sentinel) // trust me`

True invalid ranges:

- `stdr::subrange(array + 10, array)`
- `stdr::subrange(vec1.begin(), vec2.end())`

True infinite ranges:

- `stdv::repeat(42)`
- `stdv::iota(unsigned(0))`

Ranges where `.end()` lies:

- `stdv::iota(int(0))`
- `stdr::subrange(ptr, stdr::unreachable_sentinel) // trust me`

Ranges where we don't know:

- `stdv::istream`
- `std::generator<T>`

What even is infinity?

Infinite range:

```
for (auto x : stdv::repeat(42)) {  
    std::print("{}\n", x);  
}
```

What even is infinity?

Infinite range:

```
for (auto x : stdv::repeat(42)) {  
    std::print("{}\n", x);  
}
```

Infinite range?

```
for (auto c : stdv::istream<char>(std::cin)) {  
    std::print("{}\n", c);  
}
```

What even is infinity?

Infinite range:

```
for (auto x : stdv::repeat(42)) {  
    std::print("{}\n", x);  
}
```

Infinite range?

```
for (auto c : stdv::istream<char>(std::cin)) {  
    std::print("{}\n", c);  
}
```

Infinite range?

```
bool operator==(iterator it, sentinel s) {  
    return dice_roll() == 6;  
}
```

Infinite usage:

```
for (auto x : input) {  
    output(process(x));  
}
```

Finite usage:

```
stdv::zip(rng, stdv::iota(0))
```

```
stdr::transform(rng, stdv::iota(0), out, fn);
```

```
return *stdr::find_if(  
    all_fibonacci_numbers,  
    [x](int f) { return f > x; }  
);
```

Infinity sentinel: No matter how many times you do `++i`, you will not encounter undefined behavior, and `i == s` is false.

- `stdv::repeat(42)`
- `stdv::iota(unsigned(0))`
- potentially `stdv::istream<T>`
- potentially `std::generator<T>`

Infinity sentinel: No matter how many times you do `++i`, you will not encounter undefined behavior, and `i == s` is false.

- `stdv::repeat(42)`
- `stdv::iota(unsigned(0))`
- potentially `stdv::istream<T>`
- potentially `std::generator<T>`

But not:

- `stdv::iota(int(0))`
- `stdr::subrange(ptr, stdr::unreachable_sentinel)`
- `stdr::subrange(array + 10, array)`
- `stdr::subrange(vec1.begin(), vec2.end())`

Unbounded sentinel: No matter how many times you do `++i`, `i == s` is false, but you can only do it a finite amount of times before encountering undefined behavior; the true range is a finite prefix of `[i, s)`.

- `stdv::iota(int(0))`
- `stdr::subrange(ptr, stdr::unreachable_sentinel)`
- potentially `std::generator<T>`

Unbounded sentinel: No matter how many times you do `++i`, `i == s` is false, but you can only do it a finite amount of times before encountering undefined behavior; the true range is a finite prefix of `[i, s)`.

- `std::iota(int(0))`
- `std::subrange(ptr, std::unreachable_sentinel)`
- potentially `std::generator<T>`

But also (!):

- `std::subrange(array + 10, array)`
- `std::subrange(vec1.begin(), vec2.end())`
- any iterator-sentinel-pair where `i == s` compiles

Unbounded sentinel: `s` has type `std::unreachable_sentinel_t` but you can only do `++i` a finite amount of times before encountering undefined behavior; the true range is a finite prefix of `[i, s)`.

- `stdv::iota(int(0))`
- `stdr::subrange(ptr, stdr::unreachable_sentinel)`

But not:

- `stdr::subrange(array + 10, array)`
- `stdr::subrange(vec1.begin(), vec2.end())`

New definition attempt #2: Unbounded sentinel

Unbounded sentinel: `s` has type `std::unreachable_sentinel_t` but you can only do `++i` a finite amount of times before encountering undefined behavior; the true range is a finite prefix of `[i, s)`.

- `stdv::iota(int(0))`
- `stdr::subrange(ptr, stdr::unreachable_sentinel)`

But not:

- `stdr::subrange(array + 10, array)`
- `stdr::subrange(vec1.begin(), vec2.end())`

But also not (!):

- `std::generator<T>`

Do we care about unbounded generators?

```
std::generator<int> infinite_range() {  
    while (true) {  
        co_yield 0;  
    }  
}
```

```
std::generator<int> unbounded_range() {  
    for (auto i = 0; true; ++i) {  
        if (i == 10) undefined_behavior();  
        co_yield 0;  
    }  
}
```

What about the difference type?

After an infinite number of `++i`, `i - begin()` does not fit in `ptrdiff_t`.

What about the difference type?

After an infinite number of `++i`, `i - begin()` does not fit in `ptrdiff_t`.

Do we care?

- Finite usage: does not matter.
- Infinite usage: maybe?

What about the difference type?

After an infinite number of `++i`, `i - begin()` does not fit in `ptrdiff_t`.

Do we care?

- Finite usage: does not matter.
- Infinite usage: maybe?

`ptrdiff_t` on 32 bit system:

```
std::span<unsigned char> memory = allocate_virtual_memory(3 * 1024 * 1024 * 1024);  
memory.end() - memory.begin();
```

What about cyclic ranges?

```
void foo(stdr::iota_view<unsigned> rng, stdr::iota_view<unsigned>::iterator it) +  
    assert(*rng.begin() == 0);  
    assert(*it == 4);  
    std::print("{}\n", it - rng.begin());  
}
```

What is the difference?

- 4?
- `UINT_MAX + 4`?
- `10 * UINT_MAX + 4`?

- `stdr::sized_range`: Range has a known finite size.
- `std2r::infinite_range`: Range has an infinite size.
- `stdr::range`: Range could be finite, infinite, unbounded — we don't know statically.

- `stdr::sized_range`: Range has a known finite size.
- `std2r::infinite_range`: Range has an infinite size.
- `stdr::range`: Range could be finite, infinite, unbounded — we don't know statically.

```
stdr::transform(std::execution::par, rng, stdv::iota(0u), out, fn);
```

- `stdr::sized_range`: Range has a known finite size.
- `std2r::infinite_range`: Range has an infinite size.
- `stdr::range`: Range could be finite, infinite, unbounded — we don't know statically.

```
stdr::transform(std::execution::par, rng, stdv::iota(0u), out, fn);
```

```
infinite_range | stdv::take(10) // could be optimized
```

- `stdr::sized_range`: Range has a known finite size.
- `std2r::infinite_range`: Range has an infinite size.
- `stdr::range`: Range could be finite, infinite, unbounded — we don't know statically.

```
stdr::transform(std::execution::par, rng, stdv::iota(0u), out, fn);
```

```
infinite_range | stdv::take(10) // could be optimized
```

Proposed by P3555R0.