

Faster, Safer,  
Better Ranges

**Tristan Brindle**

2025

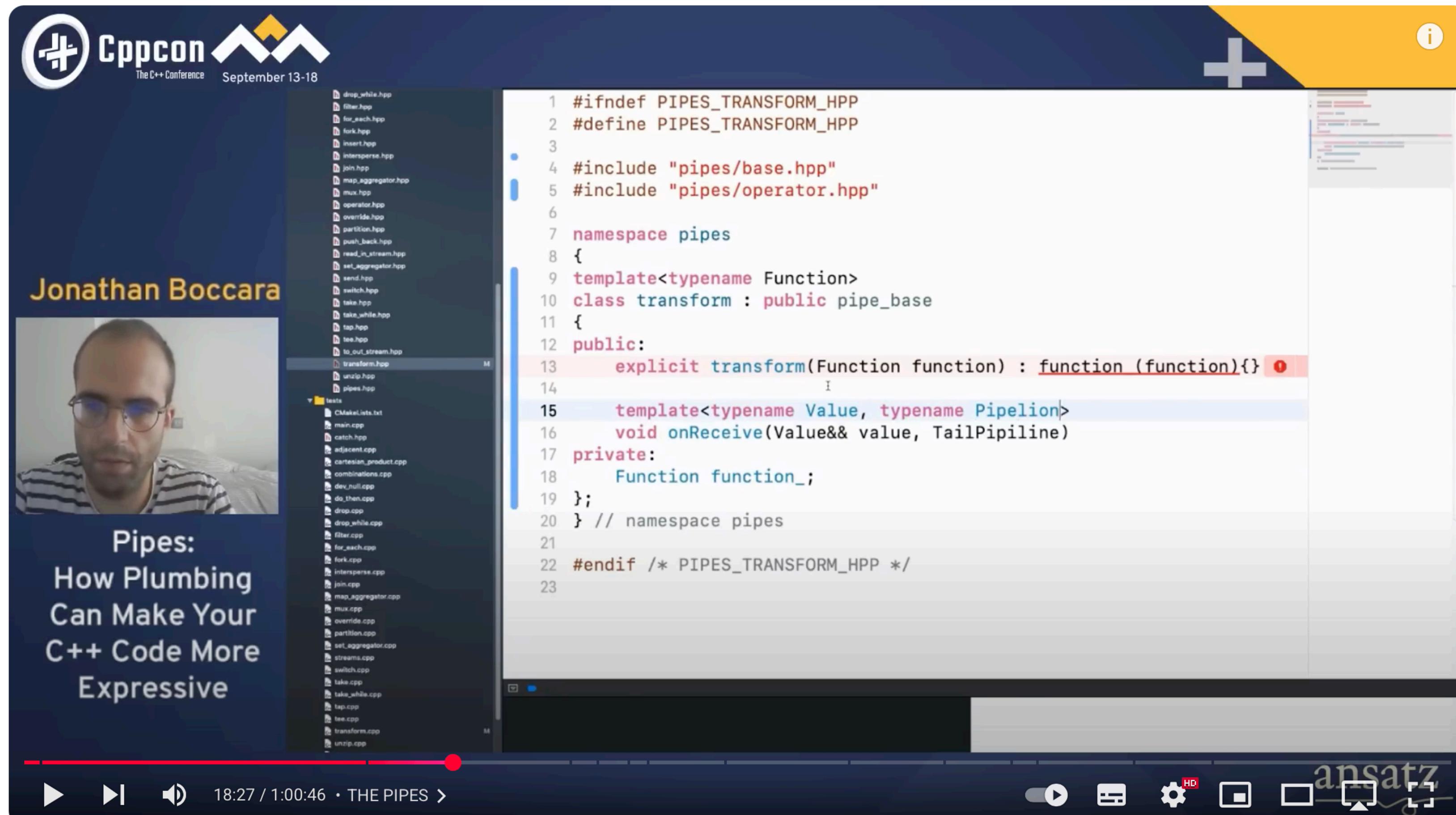
# Faster, Safer, Better Ranges

Tristan Brindle  
C++ on Sea 2025

# Recent Ranges talks

- If you watch a lot of conference talks, you may have noticed a theme developing over the last few years

# Recent Ranges talks



# Recent Ranges talks

## Iterators and Ranges: Comparing C++ to D, Rust, and Others

### Implementing filter with Streams

```
template <stream S, predicate<stream_reference_t<S>> F>
class filter_stream {
    S stream;
    F filt;

public:
    using reference = stream_reference_t<S>;

    template <predicate<reference> P>
    auto while_(P pred) -> bool {
        return stream.while_([&](reference elem){
            if (invoke(filt, elem)) {
                return invoke(pred, elem);
            } else {
                return true;
            }
        });
    }
};
```



Barry Revzin



CODE RECKONS

Science to the CORE



1:06:37 / 1:31:27



Keynote: Iterators and Ranges: Comparing C++ to D, Rust, and Others - Barry Revzin - CPPP 2021

# Recent Ranges talks

The screenshot shows a video player interface for a talk titled "Why Iterators Got It All Wrong – And What We Should Use Instead" by Arno Schödl at CppNow 2022.

**Top Left:** C++ now | 2022 MAY 1-6 Aspen, Colorado, USA

**Top Right:** i

**Speaker Image:** A photo of Arno Schödl standing on stage, wearing a light-colored hoodie and glasses. Behind him is a chalkboard with the text "up physcis Password Universe".

**Title Bar:** Iterators were always ugly think-cell

**Content Area:**

- `begin()` and `end()` asymmetric
  - can dereference `begin()`
  - cannot dereference `end()`

```
begin           end
  v   v   v   v   v
{ a , b , c , d }
```

**Bottom Left:** sonar JET BRAINS

**Bottom Center:** 32:33 / 1:16:21

**Bottom Right:** CppNow.org

Why Iterators Got It All Wrong – And What We Should Use Instead - Arno Schödl - CppNow 2022

# Recent Ranges talks

The screenshot shows a video player interface. At the top, there's a logo with three colored squares (blue, yellow, red) followed by the text "Iteration Revisited". To the right is the "think-cell" logo with a small circular icon. Below the title, there's a thumbnail image of a man with short dark hair, wearing a white and blue striped polo shirt, standing and gesturing with his right hand. He is positioned next to a silver laptop on a desk. The main content area has a light gray background with faint, semi-transparent text that reads "ITERATION REVISITED", "ITERATORS AND UB", and "auto iter = vec.begin();". On the right side of the slide, there are several small, illegible text snippets. Below the slide, the video player interface includes a progress bar with a red slider, the text "Tristan Brindle", and the URL "cpponsea.us". The bottom of the screen shows a dark bar with various control icons (play, volume, settings, etc.) and the text "12:59 / 1:13:24 · Undefined Behavior >".

ITERATION REVISITED

ITERATORS AND UB

```
auto iter = vec.begin();
vec.push_back(99);
while (iter != vec.end()) {
    ...
}
```

▶ Iterators can become dangling in non-obvious ways

Tristan Brindle

12:59 / 1:13:24 · Undefined Behavior >

2023

A Safer Iteration Model for C++ - Tristan Brindle - C++ on Sea 2023

# Recent Ranges talks

The screenshot shows a video player interface for a talk at C++ Now 2024. The top left corner displays the event logo and details: "C++ now 2024 Aspen, Colorado" and "CppNow.org". Below this, it says "Video Sponsorship Provided By millennium think-cell". The main video frame shows a man with glasses and a beard, wearing a blue patterned shirt, speaking. To his right is a presentation slide with the title "OPTIMIZED" and a subtitle "Optimized". The slide contains a bar chart comparing three methods: StdRange (blue), Rappel (red), and Handwritten Loop (yellow). The x-axis represents data sizes: 1, 8, 64, 512, 4096, 32768, and 65536. The y-axis is logarithmic, ranging from 1.00E+0 to 1.00E+5. The chart shows that StdRange generally performs best, followed by Rappel, and then the Handwritten Loop. The slide also includes the text "Rappel: Compose Algorithms, Not Iterators" and "Google's Alternative to Ranges" along with the speakers' names, John Bandela and Chris Philip.

Optimized

StdRange Rappel Handwritten Loop

Data Size	StdRange	Rappel	Handwritten Loop
1	~10	~10	~5
8	~100	~50	~50
64	~1000	~200	~200
512	~5000	~1000	~1000
4096	~30000	~8000	~8000
32768	~150000	~100000	~100000
65536	~200000	~150000	~150000

Rappel: Compose Algorithms, Not Iterators  
Google's Alternative to Ranges

John Bandela, Chris Philip

21:39 / 1:18:30

Rappel: Compose Algorithms, Not Iterators - Google's Alternative to Ranges - C++Now 2024

# Recent Ranges talks

The screenshot shows a video player interface for a talk titled "Ranges++: Are Output Range Adaptors the Next Iteration of C++ Ranges? - Daisy Hollman - CppCon 2024".

**Left Side:** The video player has a dark blue header with the Cppcon logo and "The C++ Conference" text. Below it is the URL "Cppcon.org". A "Video Sponsorship Provided By" section features the "think-cell" logo. A thumbnail image of the speaker, Daisy Hollman, is visible.

**Right Side:** The main content area displays three code snippets related to range adaptors:

- iterator\_t<filter\_view>::operator++()**

```
1 constexpr __iterator& operator++() {  
2     do {  
3         ++__current_;  
4     } while (!std::invoke(*__parent_>__pred_, *__current_));  
5     return *this;  
6 }
```
- iterator\_t<transform\_view>::operator\*()**

```
1 constexpr decltype(auto) operator*() const  
2 {  
3     return std::invoke(*__parent_>__func_, *__current_);  
4 }
```
- iterator\_t<filter\_view>::operator\*()**

```
1 constexpr range_reference_t<_View> operator*() const {  
2     return *__current_;  
3 }
```

A text overlay in the center-right states: "We're invoking the `transform_view`'s `__func__` in **both** `filter_view::operator++()` **and** `filter_view::operator*()`!"

The bottom right corner of the video player shows the Cppcon logo and the number "24".

Ranges++: Are Output Range Adaptors the Next Iteration of C++ Ranges? - Daisy Hollman - CppCon 2024

# Recent Ranges talks

The curse of pull

12

```
std::vector<int> rng;
int res;

auto is_even = [](auto x) { return x%2 == 0; };
auto x3 = [](auto x) { return x*3; };
void dst(int);

void push()
{
    for(auto x: rng)
        if(is_even(x)) dst(x3(x));
}

void pull()
{
    for (auto x:
        rng | filter(is_even) | transform(x3))
        dst(x);
}
```

<https://godbolt.org/z/o4KTKeF59>

`_Z4pushv:`  
push r14  
push rbx  
push rax  
mov rbx, qword ptr [rip + rng]  
mov r14, qword ptr [rip + rng+8]  
.LBB1\_1:  
jmp .LBB1\_4:  
add rbx, 4  
.LBB1\_1:  
rbx, r14  
cmp je .LBB1\_5:  
mov eax, dword ptr [rbx]  
test al, 1  
jne .LBB1\_4:  
lea edi, [rax + 2\*rax]  
call \_Z3dsti@PLT  
.LBB1\_5:  
jmp .LBB1\_4:  
add rsp, 8  
pop rbx  
pop r14  
ret

SCALIAN Qualcomm Bloomberg JFrog CONAN Fundación UC3 Fundación Universidad Carlos III SECURITAS Direct UNE think-cell

▶ ▶ ⏪ 12:58 / 51:16

Push is faster - Joaquín M López Muñoz

# Recent Ranges talks



# Ranges alternatives

- Ranges offers a great set of features bringing sequence-orientated programming to C++
- But it has certain problems in terms of safety, performance and ease of use
- These largely stem from the underlying iterator model

# Ranges problems

```
void print_evens(std::vector<int> const& vec)
```

# Ranges problems

```
void print_evens(std::vector<int> const& vec)
{
    for (int i : vec) {
        if (i % 2 == 0) {
            std::println("{}", i);
        }
    }
}
```

# Ranges problems

```
void print_evens_v2(std::vector<int> const& vec)
{
    auto is_even = [] (int i) { return i % 2 == 0; };
    auto view = std::views::filter(vec, is_even);

    for (int i : view) {
        std::println("{}", i);
    }
}
```

# Ranges problems

```
void print_evens_v2(std::vector<int> const& vec)
{
    auto is_even = [](int i) { return i % 2 == 0; };
    auto view = std::views::filter(vec, is_even);

    auto view_iter = view.begin();
    auto view_last = view.end();

    while (view_iter != view_last) {
        std::println("{}", *view_iter);
        ++view_iter;
    }
}
```

# Ranges problems

```
void print_evens_v2(std::vector<int> const& vec)
{
    auto is_even = [] (int i) { return i % 2 == 0; };

    auto vec_iter = std::ranges::find_if(vec, is_even);
    auto vec_last = values.end();

    while (vec_iter != vec_last) {
        std::println("{}", *vec_iter);
        vec_iter = std::ranges::find_if(++vec_iter, vec_last, is_even);
    }
}
```

# Ranges problems

```
void print_evens_v2(std::vector<int> const& vec)
{
    auto is_even = [] (int i) { return i % 2 == 0; };

    auto vec_iter = values.begin();
    auto vec_last = values.end();

    while (vec_iter != vec_last) {
        if (is_even(*vec_iter)) {
            break;
        }
        ++vec_iter;
    }

    while (vec_iter != vec_last) {
        std::println("{}", *vec_iter);
        vec_iter = std::ranges::find_if(++vec_iter, vec_last, is_even);
    }
}
```

# Ranges problems

```
void print_evens_v2(std::vector<int> const& vec)
{
    auto is_even = [] (int i) { return i % 2 == 0; };

    auto vec_iter = values.begin();
    auto vec_last = values.end();

    while (vec_iter != vec_last) {
        if (is_even(*vec_iter)) {
            break;
        }
        ++vec_iter;
    }

    while (vec_iter != vec_last) {
        std::println("{}", *vec_iter);

        ++vec_iter;
        while (vec_iter != vec_last) {
            if (is_even(*vec_iter)) {
                break;
            }
            ++vec_iter;
        }
    }
}
```

```
void print_evens_v1(std::vector<int> const& vec)
{
    auto vec_iter = vec.begin();
    auto vec_last = vec.end();

    while (vec_iter != vec_last) {
        int i = *vec_iter;
        if (i % 2 == 0) {
            std::println("{}", i);
        }
        ++vec_iter;
    }
}
```

# Ranges problems

- Could a sufficiently smart compiler see through the extra loops?
- Maybe, but not today

# Ranges problems

Compiler Explorer interface showing three panes comparing C++ code and its assembly output for different range-based algorithms.

**Left Pane (C++ Source #1):**

```
1 #include <algorithm>
2 #include <ranges>
3 #include <vector>
4
5 #ifndef USE_RANGES
6
7 int sum_evens(std::vector<int> const& vec)
8 {
9     int sum = 0;
10    for (int i : vec) {
11        if (i % 2 == 0) {
12            sum += i;
13        }
14    }
15
16    return sum;
17 }
18
19 #else
20
21 int sum_evens_ranges(std::vector<int> const& vec)
22 {
23     auto is_even = [](int i) { return i % 2 == 0; };
24     auto view = std::views::filter(vec, is_even);
25
26     int sum = 0;
27     for (int i : view) {
28         sum += i;
29     }
30     return sum;
31 }
32
33 #endif
```

**Middle Pane (Assembly Output - std=c++23 -O3):**

```
1 .LBB0_6:
2     movdqa xmm5, xmmword ptr [rip + _ZELF10_0]
3     pxor    xmm2, xmm2
4     pxor    xmm1, xmm1
5
6     movdqu xmm4, xmmword ptr [r8 + 4*rax]
7     movdqu xmm5, xmmword ptr [r8 + 4*rax + 16]
8
9     movdqa xmm6, xmm4
10    pand   xmm6, xmm3
11    movdqa xmm7, xmm5
12    pand   xmm7, xmm3
13    pcmpeqd xmm6, xmm0
14    pand   xmm6, xmm4
15    paddd  xmm2, xmm6
16    pcmpeqd xmm7, xmm0
17    pand   xmm7, xmm5
18    paddd  xmm1, xmm7
19    add    rax, 8
20    cmp    r9, rax
21    jne    .LBB0_6
22    paddd  xmm1, xmm2
23    pshufd xmm0, xmm1, 238
24    paddd  xmm0, xmm1
25    pshufd xmm1, xmm0, 85
26    paddd  xmm1, xmm0
27    movd   eax, xmm1
28    cmp    rdi, r9
29    je     .LBB0_2
30
31    .LBB0_8:
32    mov    edi, dword ptr [rsi]
33    test   dil, 1
34    cmovne edi, edx
35    add    eax, edi
36    add    rsi, 4
37    cmp    rsi, rcx
38    jne    .LBB0_8
39
40    .LBB0_2:
41    ret
```

**Right Pane (Assembly Output - std=c++23 -O3 -DUSE\_R):**

```
1 sum_evens_ranges(std::vector<int>, std::allocator<int>)
2     mov    rdx, qword ptr [rdi]
3     mov    rcx, qword ptr [rdi + 8]
4     cmp    rdx, rcx
5     je    .LBB0_4
6
7     test   byte ptr [rdx], 1
8     je    .LBB0_4
9     add    rdx, 4
10    cmp   rdx, rcx
11    jne    .LBB0_2
12
13    xor    eax, eax
14    cmp    rdx, rcx
15    jne    .LBB0_5
16
17    ret
18
19    .LBB0_9:
20    add    eax, dword ptr [rdx]
21    mov    rdx, rsi
22    cmp    rsi, rcx
23    je    .LBB0_10
24
25    lea    rsi, [rdx + 4]
26    cmp    rsi, rcx
27    je    .LBB0_9
28
29    test   byte ptr [rsi], 1
30    je    .LBB0_9
31    add    rsi, 4
32    cmp    rsi, rcx
33    jne    .LBB0_7
34    jmp    .LBB0_9
35
36    .LBB0_10:
37    ret
38
39    .LBB0_7:
40    test   byte ptr [rsi], 1
41    je    .LBB0_9
42    add    rsi, 4
43    cmp    rsi, rcx
44    jne    .LBB0_7
45    jmp    .LBB0_9
46
47    .LBB0_5:
48    lea    rsi, [rdx + 4]
49    cmp    rsi, rcx
50    je    .LBB0_9
51
52    .LBB0_7:
53    test   byte ptr [rsi], 1
54    je    .LBB0_9
55    add    rsi, 4
56    cmp    rsi, rcx
57    jne    .LBB0_7
58
59    .LBB0_2:
60    ret
```

Output (0/0) x86-64 clang 20.1.0 - 1222ms (112531B) ~8796 lines filtered

# Ranges problems

```
// From P3725, courtesy of Nico Josuttis
std::vector<std::string> cities{"Amsterdam", "Berlin", "Cologne", "LA"};

// move long strings in reverse order to another container:
auto large = [](const auto& s) { return s.size() > 5; };
auto sub = cities | std::views::filter(large)
                  | std::views::reverse
                  | std::views::as_rvalue
                  | std::ranges::to<std::vector>();
```

- Expected result: **sub** contains ["Cologne", "Berlin", "Amsterdam"]
- Actual result: iteration overruns the bounds of **cities** and starts reading arbitrary data on the heap

# A ranges alternative

- I want a sequence-orientated programming library that:
  - Is competitive in performance with hand-written loops
  - Offers good safety by default
  - Is easy to use, predictable, with no unpleasant surprises
  - Is as compatible as possible with existing STL containers and algorithms

# Flux v1

- Flux v1 was my first attempt at writing a library to meet these goals
- It was based around the idea of bounds-checked cursors as an alternative to iterators
- As an optimisation, it also provided a “fast path” using *internal iteration* for certain combinations of adaptors and algorithms
- All Flux sequences were automatically C++20 ranges of the equivalent category
- All C++20 contiguous ranges were automatically Flux contiguous sequences
- For ranges of lower categories, Flux v1 offered no additional safety or performance over using the ranges library directly

# Aside: internal iteration

- *External or pull-based* iteration is the interface style used by the majority of programming languages, including C++ .
  - We *pull* a value from a sequence using a function like `next()` or `operator++()`, use it, and then pull the next value and so on
- *Internal or push-based* iteration is more like `for_each()`: we pass a function that we want to be called for each element in turn
- Broad generalisation: external iteration is more powerful, but internal iteration is more easily optimised
- Read more: <https://journal.stuffwithstuff.com/2013/01/13/iteration-inside-and-out/>

# Flux v1: lessons learnt

- Internal iteration results in fantastic performance
  - ...but could not be used with algorithms which need to pause and resume iteration
  - ...it was easy to accidentally fall off the “fast path”
- Cursors are a great interface for multi-pass sequences (aka forward ranges)
  - ...but as with ranges, trying to generalise a “position-based” interface back to single-pass sequences leads to problems
- Lack of support for node-based containers was a major barrier to adoption
- But people *really* like chaining with dots instead of pipes!

# Flux v2

- The big idea: provide distinct interfaces for single- and multi-pass sequence access
- **iterable**: single-pass iteration using a new and improved internal iteration scheme
- **collection**: multi-pass iteration using a streamlined version of the existing `flux::multipass_sequence` interface

# Disclaimer

# Disclaimer

- The following represents the current development version of Flux
- Naming and other details may change between now and the final release
- But if you have feedback or suggestions, now would be a great time

# Iteration contexts

- At the heart of the new single-pass iteration model is the *iteration context*
- They are responsible for actually performing the work of iterating
- An iteration context must provide:
  - An `element_type` `typedef`
  - A member function template `run_while()` taking a unary predicate and returning a `flux::iteration_result`

# Iteration contexts

- The `run_while()` function invokes the given predicate for each element in turn
  - If the predicate returns `false`, iteration stops and `run_while()` returns `iteration_result::incomplete`
  - Otherwise, when the sequence is exhausted `run_while()` returns `iteration_result::complete`
- If iteration is incomplete, the next call to `run_while()` will resume iteration from the next element
  - Iteration contexts are expected to keep track of any necessary state to allow this to work

# Iteration contexts

- An iteration context typically holds a reference to its parent iterable, along with any necessary iteration state
- Contexts are typically *non-copyable* and *non-movable*
  - Helps to prevent references escaping “into the wild”
  - Iteration contexts are mostly an implementation detail: end users will typically use higher-level algorithms and adaptors on iterables

# Iteration context example

```
struct ints_iteration_context : flux::immovable {
    int current;
    int const bound;

    ints_iteration_context(int from, int to)
        : current(from), bound(to) {}

    using element_type = int;

    auto run_while(auto&& pred) -> flux::iteration_result
    {
        while (current < bound) {
            if (!pred(current++)) {
                return flux::iteration_result::incomplete;
            }
        }
        return flux::iteration_result::complete;
    }
};
```

# Iteration context example 2

```
template <std::ranges::forward_range R>
struct range_iteration_context : immovable {
    std::ranges::iterator_t<R> iter;
    std::ranges::sentinel_t<R> const sent;

    range_iteration_context(R& rng)
        : iter(std::ranges::begin(rng)),
        sent(std::ranges::end(rng))
    {}

    using element_type = std::ranges::range_reference_t<R>;

    iteration_result run_while(auto&& pred)
    {
        while (iter != sent) {
            if (!pred(*iter++)) {
                return iteration_result::incomplete;
            }
        }
        return iteration_result::complete;
    }
};
```

# Iteration context example 2

```
template <std::ranges::input_range R>
struct range_iteration_context : immovable {
    std::ranges::iterator_t<R> iter;
    std::ranges::sentinel_t<R> const sent;
    bool inc_next = false;

    range_iteration_context(R& rng)
        : iter(std::ranges::begin(rng)),
          sent(std::ranges::end(rng))
    {}

    using element_type = std::ranges::range_reference_t<R>;

    iteration_result run_while(auto&& pred)
    {
        if (inc_next && iter != sent) {
            ++iter;
            inc_next = false;
        }

        while (iter != sent) {
            if (!pred(*iter)) {
                inc_next = true;
                return iteration_result::incomplete;
            }
            ++iter;
        }
        return iteration_result::complete;
    }
};
```

# Iteration context functions

```
auto flux::run_while(auto& ctx, auto&& pred) -> iteration_result;
```

- Returns `ctx.run_while(pred)`

```
auto flux::next_element(Ctx& ctx) -> optional<Ctx::element_type>;
```

- Equivalent to:

```
optional<Ctx::element_type> result = nullopt;
run_while(ctx, [&](auto&& val) {
    result.emplace(FWD(val));
    return false;
});
return result;
```

- Allows single-stepping through sequences
- We can easily switch from *push-based* to *pull-based* iteration!

# Iterables

- Iteration contexts perform the low-level operations, but a design goal is that they should be mostly be considered an implementation detail
- Rather, end users will typically use algorithms and adaptors on higher-level **iterables**
- An **iterable** must provide an **iterate()** function, which returns an **iteration\_context**
- Optionally, it can also provide a const-qualified overload of **iterate()** to enable const iteration

# Iterable example

```
struct ints {
    int from;
    int to;
};

template <>
struct flux::iterable_traits<ints> {
    static auto iterate(ints const& i) {
        return ints_iteration_context{i.from, i.to};
    }
};
```

# Iterable example

```
struct ints {
    int from;
    int to;

    auto iterate() const {
        return ints_iteration_context{from, to};
    }
};
```

# Iterable example

```
struct ints {
    int from;
    int to;

    struct context_type : flux::immovable {
        int current;
        int const bound;

        using element_type = int;

        auto run_while(auto&& pred) -> flux::iteration_result
        {
            while (current < bound) {
                if (!pred(current++)) {
                    return flux::iteration_result::incomplete;
                }
            }
            return flux::iteration_result::complete;
        }
    };
};

auto iterate() const -> context_type {
    return {.current = from, .bound = to};
};
```

# Iterable algorithms

```
template <iterable It, typename Func, typename Init = iterable_value_t<It>>
    requires foldable<...>
auto fold(It&& it, Func func, Init init = {}) -> Init
{
    auto ctx = iterate(it);
    run_while(ctx, [&](auto&& elem) {
        init = std::invoke(func, move(init), FWD(elem));
        return true;
    });
    return init;
}
```

# Iterable algorithms

```
template <iterable It1, iterable It2>
    requires std::equality_comparable_with<iterable_element_t<It1>, iterable_element_t<It2>>
auto equal(It1&& it1, It2&& it2) -> bool
{
    iteration_context auto ctx1 = iterate(it1);
    iteration_context auto ctx2 = iterate(it2);

    while (true) {
        auto opt1 = next_element(ctx1);
        auto opt2 = next_element(ctx2);

        if (opt1 && opt2) {
            if (*opt1 != *opt2) {
                return false;
            }
        } else if (opt1 || opt2) {
            return false;
        } else {
            return true;
        }
    }
}
```

# Iterable adaptors

- As ever, writing lazy adaptors requires a bit more work than terminal algorithms
  - Though generally less so than sequence adaptors in today's Flux
  - and much less than most C++20 range adaptors
- An adaptor will normally provide an iteration context which wraps the context of its base iterable
- The adaptor context's `run_while()` will typically call `run_while()` on the base context, passing it a modified predicate

# Iterable adaptor example

```
template <iterable Base, typename FilterFn>
struct filter_adaptor {
    Base base;
    FilterFn filter_fn;

    template <typename BaseCtx, typename Fn>
    struct context_type {
        BaseCtx base_ctx;
        Fn filter_fn;

        using element_type = typename BaseCtx::element_type;

        auto run_while(auto&& pred) -> iteration_result
        {
            return base_ctx.run_while([&](auto&& elem) {
                if (filter_fn(elem)) {
                    return pred(FWD(filter));
                } else {
                    return true; // continue
                }
            });
        }
    };
};
```

# Iterable adaptor example

```
template <iterable Base, typename FilterFn>
struct filter_adaptor {

    // continued...

    auto iterate()
    {
        return context_type{.base_ctx = flux::iterate(base),
                           .filter_fn = std::ref(filter_fn)};
    }

    auto iterate() const requires iterable<Base const>
    {
        return context_type{.base_ctx = flux::iterate(base),
                           .filter_fn = std::ref(filter_fn)};
    }
};
```

# Filter in use

```
void print_evens(std::vector<int> const& vec)
{
    for (int i : vec) {
        if (i % 2 == 0) {
            std::println("{}", i);
        }
    }
}
```

# Filter in use

```
void print_evens_v3(std::vector<int> const& vec)
{
    auto it = flux::filter(flux::ref(vec), flux::pred::even);

    flux::for_each(it, [](int i) {
        std::println("{}", i);
    });
}
```

# Filter in use

```
void print_evens_v3(std::vector<int> const& vec)
{
    auto it = flux::filter(flux::ref(vec), flux::pred::even);

    filter_iteration_context filter_ctx = flux::iterate(it);

    filter_ctx.run_while([&](int i) {
        std::println("{}", i);
        return true;
    });
}
```

# Filter in use

```
void print_evens_v3(std::vector<int> const& vec)
{
    range_iteration_context rng_ctx = flux::iterate(vec);

    rng_ctx.run_while([&](int i) {
        if (flux::pred::even(i)) {
            return [&](int i) {
                std::println("{}", i);
                return true;
            }(i);
        } else {
            return true;
        }
    });
}
```

# Filter in use

```
void print_evens_v3(std::vector<int> const& vec)
{
    range_iteration_context rng_ctx = flux::iterate(vec);

    rng_ctx.run_while([&](int i) {
        if (flux::pred::even(i)) {
            std::println("{}", i);
        }
        return true;
    });
}
```

# Filter in use

```
void print_evens_v3(std::vector<int> const& vec)
{
    auto iter = std::ranges::begin(vec);
    auto last = std::ranges::end(vec);

    while (iter != last) {
        auto pred = [&](int i) {
            if (flux::pred::even(i)) {
                std::println("{}", i);
            }
        }
        return true;
    );
    if (!pred(*iter++)) {
        break;
    }
}
```

# Filter in use

```
void print_evens_v3(std::vector<int> const& vec)
{
    auto iter = std::ranges::begin(vec);
    auto last = std::ranges::end(vec);

    while (iter != last) {
        int i = *iter++;
        if (i % 2 == 0) {
            std::println("{}", i);
        }
    }
}
```

# Filter in use

Compiler Explorer Add... More Templates Watch C++ Weekly to learn new C++ features Sponsors intel Cppcon Google Share Policies Other

C++ source #1 x86-64 clang 20.1.0 (Editor #1) x86-64 clang 20.1.0 (Editor #1) x86-64 clang 20.1.0 (Editor #1)

A + V 🔍 🔍 -std=c++23 -O3 -std=c++23 -O3 -DUSE\_F

```
4 #include <ranges>
5 #include <vector>
6
7 #ifndef USE_FLUX
8
9 int sum_evens(std::vector<int> const& vec)
10 {
11     int sum = 0;
12     for (int i : vec) {
13         if (i % 2 == 0) {
14             sum += i;
15         }
16     }
17
18     return sum;
19 }
20
21 #else
22
23 int sum_evens_flux(std::vector<int> const& vec)
24 {
25     return flux::ref(vec)
26         .filter(flux::pred::even)
27         .sum();
28 }
29
30 #endif
```

x86-64 clang 20.1.0 (Editor #1) -std=c++23 -O3

A 🔍 🔍 -std=c++23 -O3

```
26     mov    r9, rdi
27     and    r9, -8
28     lea    rsi, [r8 + 4*r9]
29     pxor   xmm0, xmm0
30     xor    eax, eax
31     movdqa xmm3, xmmword ptr [rip + .LCPI0_0]
32     pxor   xmm2, xmm2
33     pxor   xmm1, xmm1
34 .LBB0_6:
35     movdqa xmm4, xmmword ptr [r8 + 4*rax]
36     movdqa xmm5, xmmword ptr [r8 + 4*rax + 16]
37     movdqa xmm6, xmm4
38     pand   xmm6, xmm3
39     movdqa xmm7, xmm5
40     pand   xmm7, xmm3
41     pcmpeqd xmm6, xmm0
42     pand   xmm6, xmm4
43     paddd  xmm2, xmm6
44     pcmpeqd xmm7, xmm0
45     pand   xmm7, xmm5
46     paddd  xmm1, xmm7
47     add    rax, 8
48     cmp    r9, rax
49     jne    .LBB0_6
50     paddd  xmm1, xmm2
51     pshufd xmm0, xmm1, 238
52     paddd  xmm0, xmm1
53     pshufd xmm1, xmm0, 85
54     paddd  xmm1, xmm0
55     movd   eax, xmm1
56     cmp    rdi, r9
57     je     .LBB0_2
58 .LBB0_8:
59     mov    edi, dword ptr [rsi]
60     test   dil, 1
61     cmovne edi, edx
62     add    eax, edi
63     add    rsi, 4
```

x86-64 clang 20.1.0 (Editor #1) -std=c++23 -O3 -DUSE\_F

A 🔍 🔍 -std=c++23 -O3 -DUSE\_F

```
21     jmp   .LBB0_2
22 .LBB0_5:
23     movabs rdi, 9223372036854775800
24     and    rdi, rdx
25     pxor   xmm0, xmm0
26     xor    eax, eax
27     movdqa xmm3, xmmword ptr [rip + .LCPI0_0]
28     pxor   xmm2, xmm2
29     pxor   xmm1, xmm1
30 .LBB0_6:
31     movdqa xmm4, xmmword ptr [rcx + 4*rax]
32     movdqa xmm5, xmmword ptr [rcx + 4*rax + 16]
33     movdqa xmm6, xmm4
34     pand   xmm6, xmm3
35     movdqa xmm7, xmm5
36     pand   xmm7, xmm3
37     pcmpeqd xmm6, xmm0
38     pand   xmm6, xmm4
39     paddd  xmm2, xmm6
40     pcmpeqd xmm7, xmm0
41     pand   xmm7, xmm5
42     paddd  xmm1, xmm7
43     add    rax, 8
44     cmp    rdi, rax
45     jne    .LBB0_6
46     paddd  xmm1, xmm2
47     pshufd xmm0, xmm1, 238
48     paddd  xmm0, xmm1
49     pshufd xmm1, xmm0, 85
50     paddd  xmm1, xmm0
51     movd   eax, xmm1
52     jmp   .LBB0_8
53 .LBB0_2:
54     ret
55 .LBB0_8:
56     cmp    rdx, rdi
57     je     .LBB0_2
58 .LBB0_9:
59     mov    r8d, dword ptr [rcx + 4*rdi]
60     test   r8b, 1
```

C Output (0/0) x86-64 clang 20.1.0 - 9689ms (123798B) ~9658 lines filtered

# Reverse iteration

- Reverse iteration in today's Flux works in almost exactly the same way as with STL bidirectional iterators
- With the new iteration model, we can dramatically simply things
- An `iterable` is additionally `reverse_iterable` if it provides a `reverse_iterate()` function which returns an `iteration_context`
  - ...one that just happens to generate elements from back to front

# Reverse iteration context example

```
template <std::ranges::bidirectional_range R>
    requires std::ranges::common_range<R>
struct range_reverse_iteration_context : immovable{
    using Iter = std::ranges::iterator_t<R>;
    Iter const start;
    Iter iter;
    using element_type = std::iter_reference_t<Iter>;
    auto run_while(auto&& pred) -> iteration_result
    {
        while (iter != start) {
            if (!pred(*--iter)) {
                return iteration_result::incomplete;
            }
        }
        return iteration_result::complete;
    }
};
```

# Iterable adaptor example

```
template <iterable Base, typename FilterFn>
struct filter_adaptor {
    Base base;
    FilterFn filter_fn;

    template <typename BaseCtx, typename Fn>
    struct context_type {
        BaseCtx base_ctx;
        Fn filter_fn;

        using element_type = typename BaseCtx::element_type;

        auto run_while(auto&& pred) -> iteration_result
        {
            return base_ctx.run_while([&](auto&& elem) {
                if (filter_fn(elem)) {
                    return pred(FWD(filter));
                } else {
                    return true; // continue
                }
            });
        }
    };
};
```

# Reverse iteration: adaptors

```
template <iterable Base, typename FilterFn>
struct filter_adaptor {
    // continued...

    auto iterate()
    {
        return context_type{.base_ctx = flux::iterate(base),
                           .filter_fn = std::ref(filter_fn)};
    }

    auto iterate() const requires iterable<Base const>
    {
        return context_type{.base_ctx = flux::iterate(base),
                           .filter_fn = std::ref(filter_fn)};
    }

    auto reverse_iterate() requires reverse_iterable<Base>
    {
        return context_type{.base_ctx = flux::reverse_iterate(base),
                           .filter_fn = std::ref(filter_fn)};
    }

    auto reverse_iterate() const requires reverse_iterable<Base const>
    {
        return context_type{.base_ctx = flux::reverse_iterate(base),
                           .filter_fn = std::ref(filter_fn)};
    }
};
```

# Reverse adaptor

```
template <reverse_iterable Base>
struct reverse_adaptor {
    Base base;

    auto iterate() { return flux::reverse_iterate(base); }

    auto iterate() const requires reverse_iterable<Base const>
    {
        return flux::reverse_iterate(base);
    }

    auto reverse_iterate() { return flux::iterate(base); }

    auto reverse_iterate() const requires iterable<Base const>
    {
        return flux::iterate(base);
    }
};
```

# Collections

# Collections

- The iterable concept is all about single-pass iteration
- But some algorithms require that we visit the same location in a sequence more than once
  - that is, multi-pass iteration
- This is the purpose of the **collection** concept
- All collections are automatically **iterable** — we provide a default **iteration\_context** implemented using the collection API

# Positions

- Every collection has an associated **position** type
- A position is like a generalisation of an array index
  - In contrast to iterators, which are a generalisation of array pointers
  - Unlike iterators, position objects are not “smart”: they do not know how to increment or dereference themselves
  - Their only role is to serve as a way of denoting an offset from the start of a collection

# Positions

- Position objects must be *regular types* – that is, default constructible, copyable and equality comparable
- Additionally, positions must have a strict total order
  - If  $p1 < p2$ , then  $p1$  represents an earlier position in a sequence than  $p2$
  - With ordered positions, we can easily and uniformly perform bounds checking for all collections
  - In particular, we can easily perform bounds checking for arbitrary *slices* of collections

# Collections

- To define a type as a collection, users need to provide implementations of four functions
- `auto begin_pos(collection const&)` returns the start position of the collection
- `auto end_pos(collection const&)` returns the end position
- `void inc_pos(collection const&, position_t&)` increments the position so that it points to the next element
- `decltype(auto)read_at_unchecked(collection&, position_t const& p)` returns the element at the given position
  - The implementor may assume that `p >= begin_pos() && p < end_pos()` on entry to this function
  - This allows us to implement e.g. `try_read_at()` without double bounds checking

# Collection example

```
template <typename T, std::size_t N>
struct my_array {
    T data[N];

    auto begin_pos() const -> std::size_t { return 0; }

    auto end_pos() const -> std::size_t { return N; }

    void inc_pos(std::size_t& pos) const { ++pos; }

    auto read_at_unchecked(std::size_t pos) -> T& {
        return data[pos];
    }

    auto read_at_unchecked(std::size_t pos) const -> T const& {
        return data[pos];
    }
};
```

# Collections

- A position is *valid* if it is greater than or equal to `begin_pos()` and less than or equal to `end_pos()`
- A position is *readable* if it is a valid position not equal to `end_pos()`
- These can be checked with `is_valid_pos()` and `is_readable_pos()` respectively
- The `read_at(col, pos)` function performs checked element access, raising a runtime error if the given position is not readable
- `try_read_at(col, pos)` returns an optional rather than trapping

# Collection refinements

- `permutable_collection` additionally provides a `swap_at(pos1, pos2)` method which swaps the elements at the given positions
- `bidirectional_collection` provides a `dec_pos()` method allowing us to step backwards through a collection
  - A `bidirectional_collection` is automatically `reverse_iterable`
- `random_access_collection` provides two additional functions:
  - `distance(pos1, pos2)` returns the (possibly negative) distance between the two positions in constant time
  - `offset_pos(pos, n)` adjusts pos by n places (which may be negative) in constant time
- `contiguous_collection` permits unsafe access to the underlying storage for low-level optimisations

# Ranges compatibility

- A good interoperability story with existing STL containers and algorithms is essential
- Every C++20 input range is automatically a **flux::iterable**
  - ...and every bidirectional range is automatically **reverse\_iterable**
- Every **flux::iterable** can be converted to a C++20 range via the **as\_range** adaptor
  - ...but range-for loops work without needing an adaptor
- Every **flux::collection** is automatically C++20 forward range or better
- Every sized, random access range is automatically a **flux::collection**

# Faster, safer, better?

- Flux today offers unrivalled performance for a library of its kind when on the internal iteration “fast path”
- The **iterable** protocol in Flux v2 allows us to expand the use of internal iteration in a wider range of algorithms, while still offering pull-based iteration where necessary
- Iteration contexts know their own bounds, and do not overstep them
- The **collection** protocol offers safe, bounds checked, generic element access for multipass sequence

# Thank you!

- Github: <https://github.com/tcbrindle/flux>

# Questions?