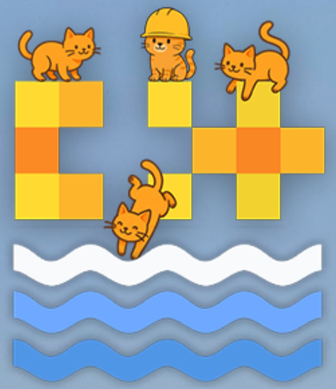


Faster, Safer, Better Ranges

Tristan Brindle

2025

Faster, Safer, Better Ranges

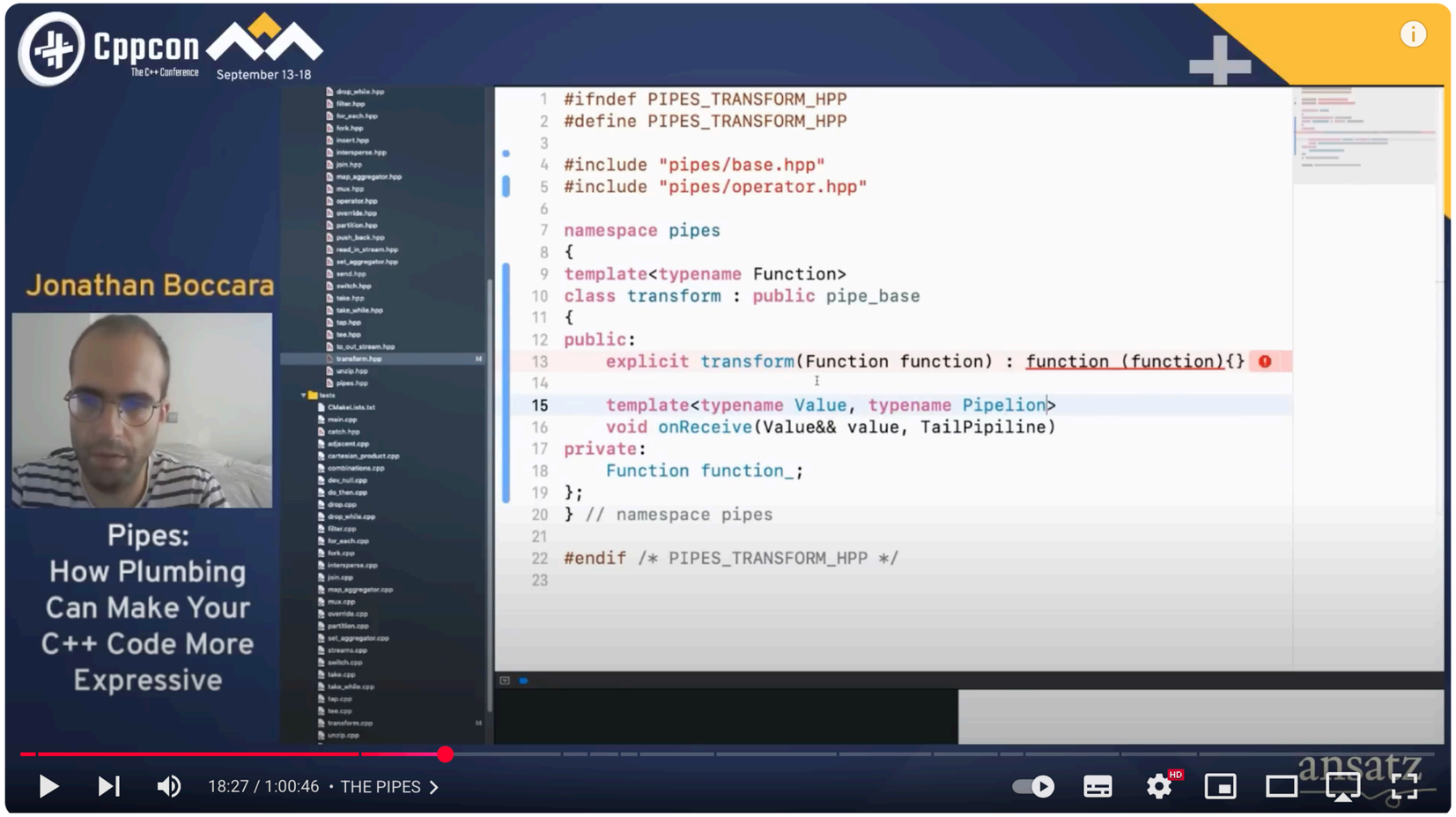
Tristan Brindle

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

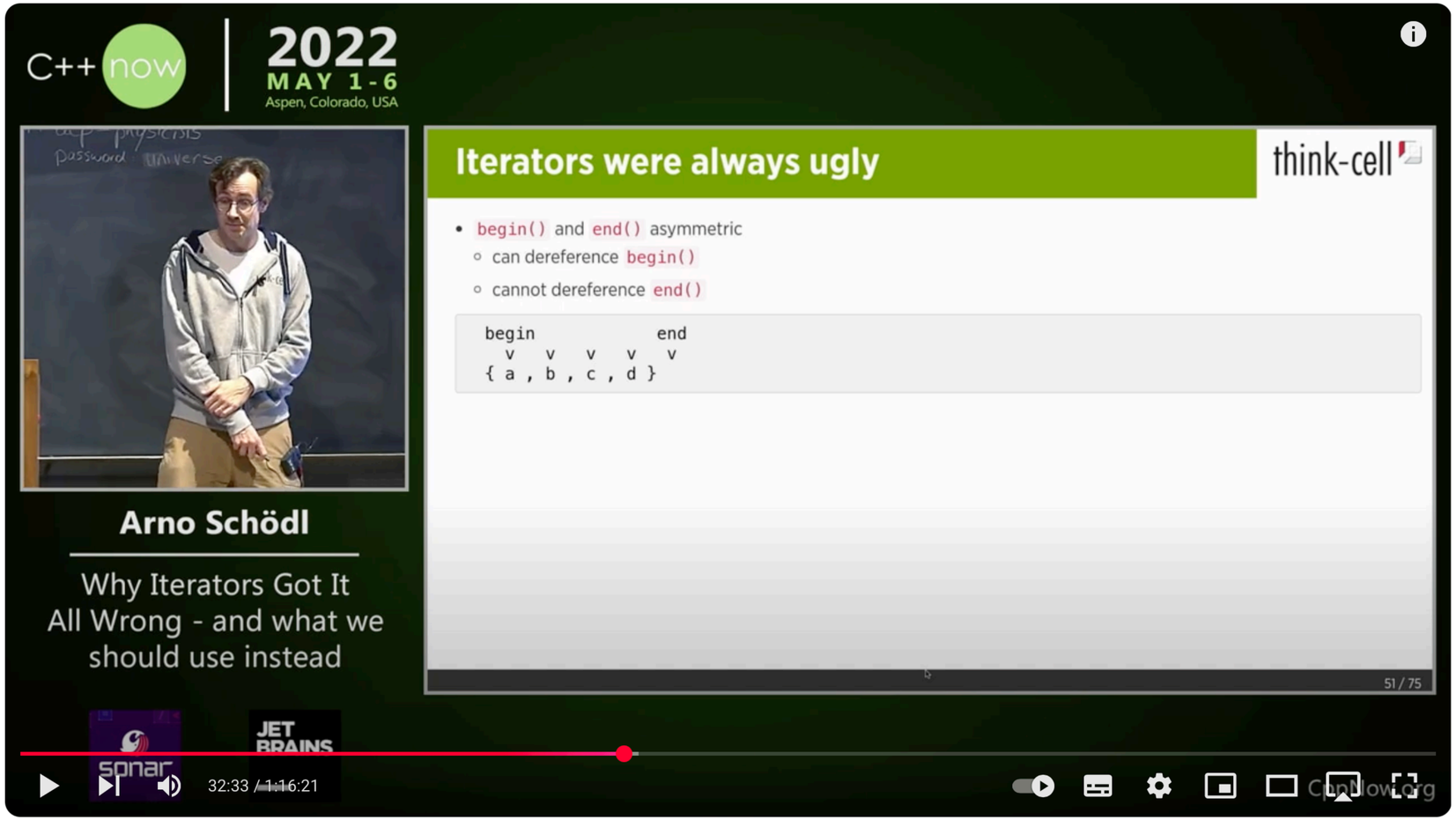
Recent Ranges talks



Recent Ranges talks

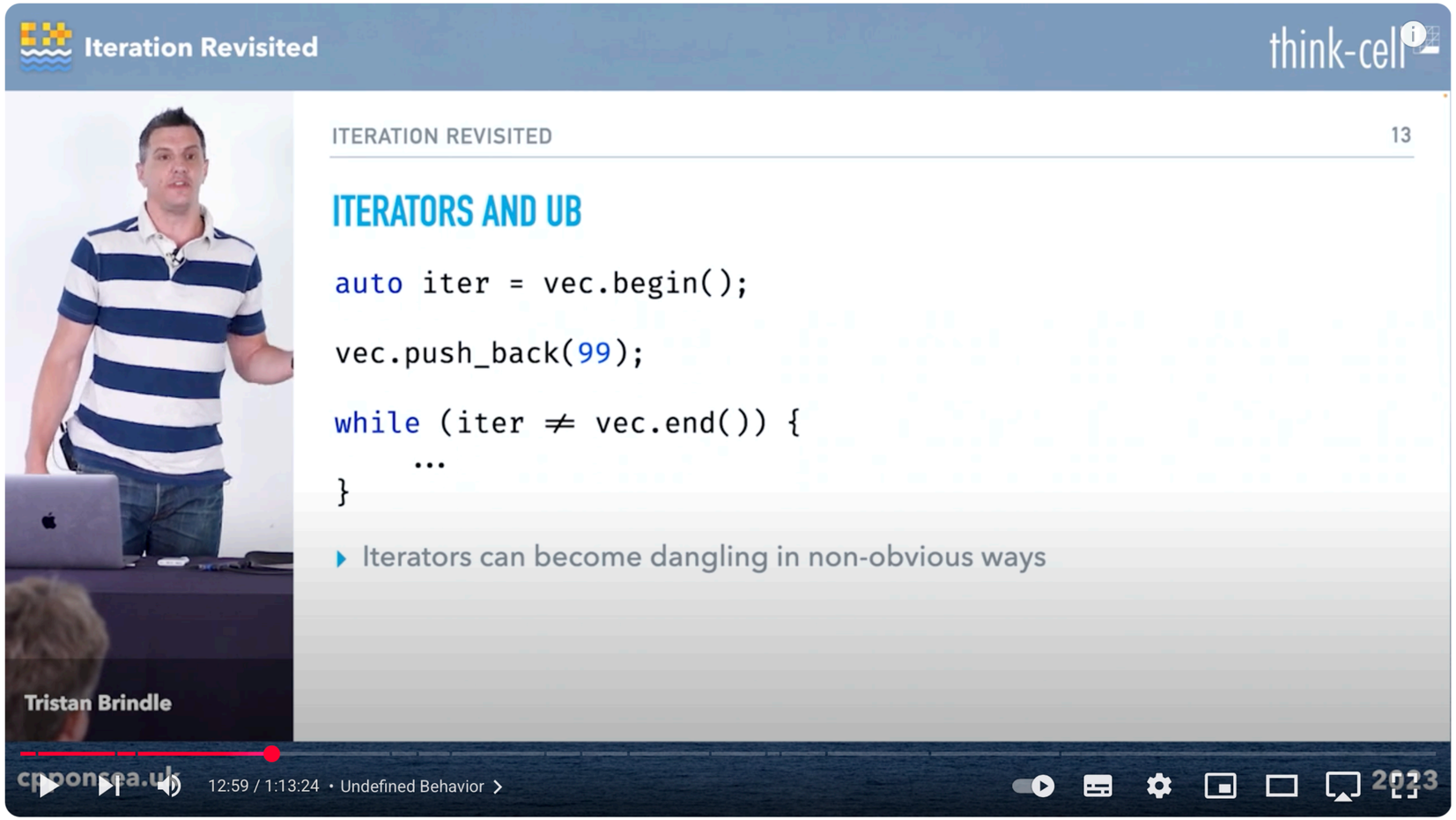
Keynote: Iterators and Ranges: Comparing C++ to D, Rust, and Others - Barry Revzin - CPPP 2021

Recent Ranges talks



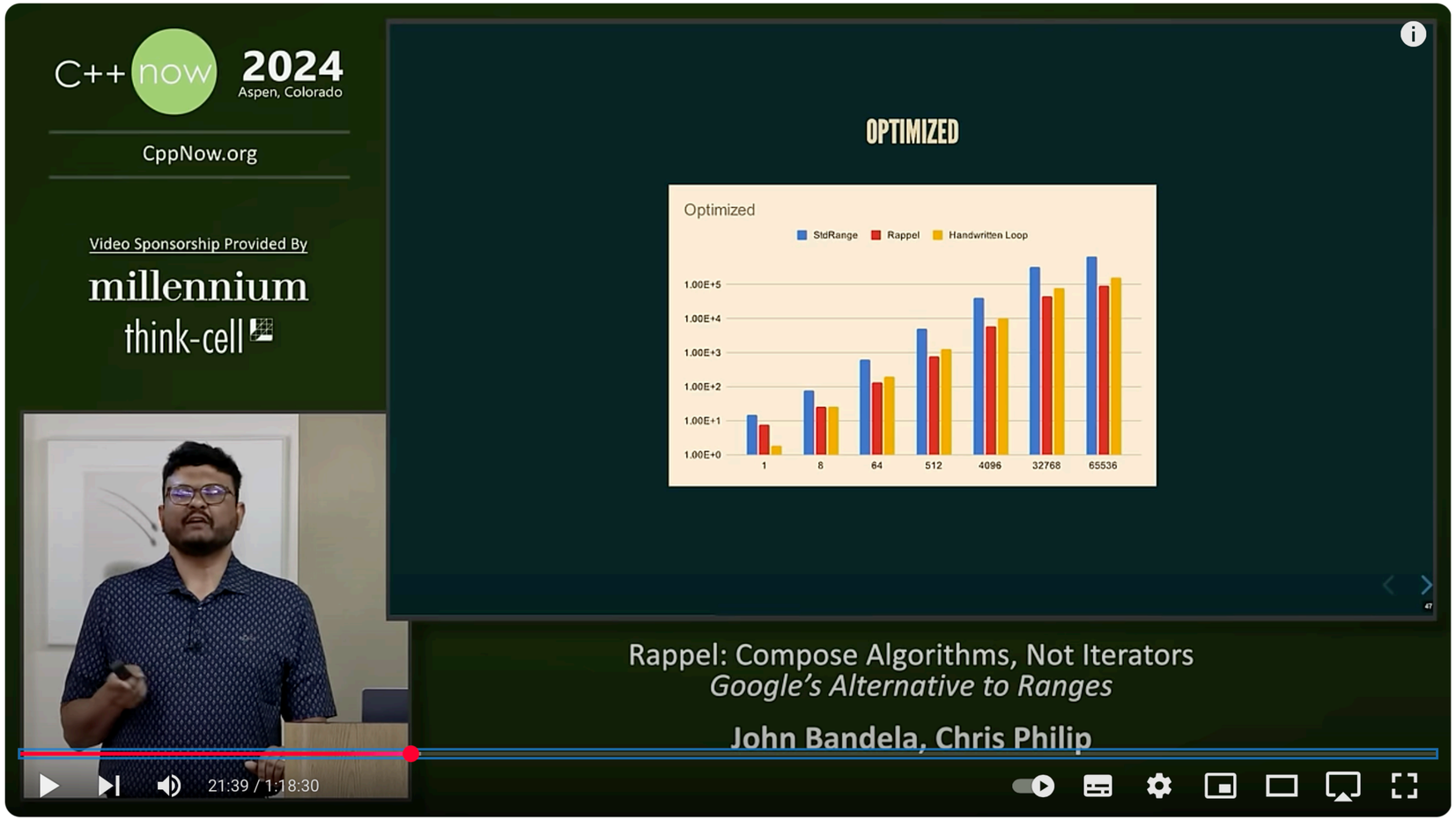
Why Iterators Got It All Wrong — And What We Should Use Instead - Arno Schödl - CppNow 2022

Recent Ranges talks



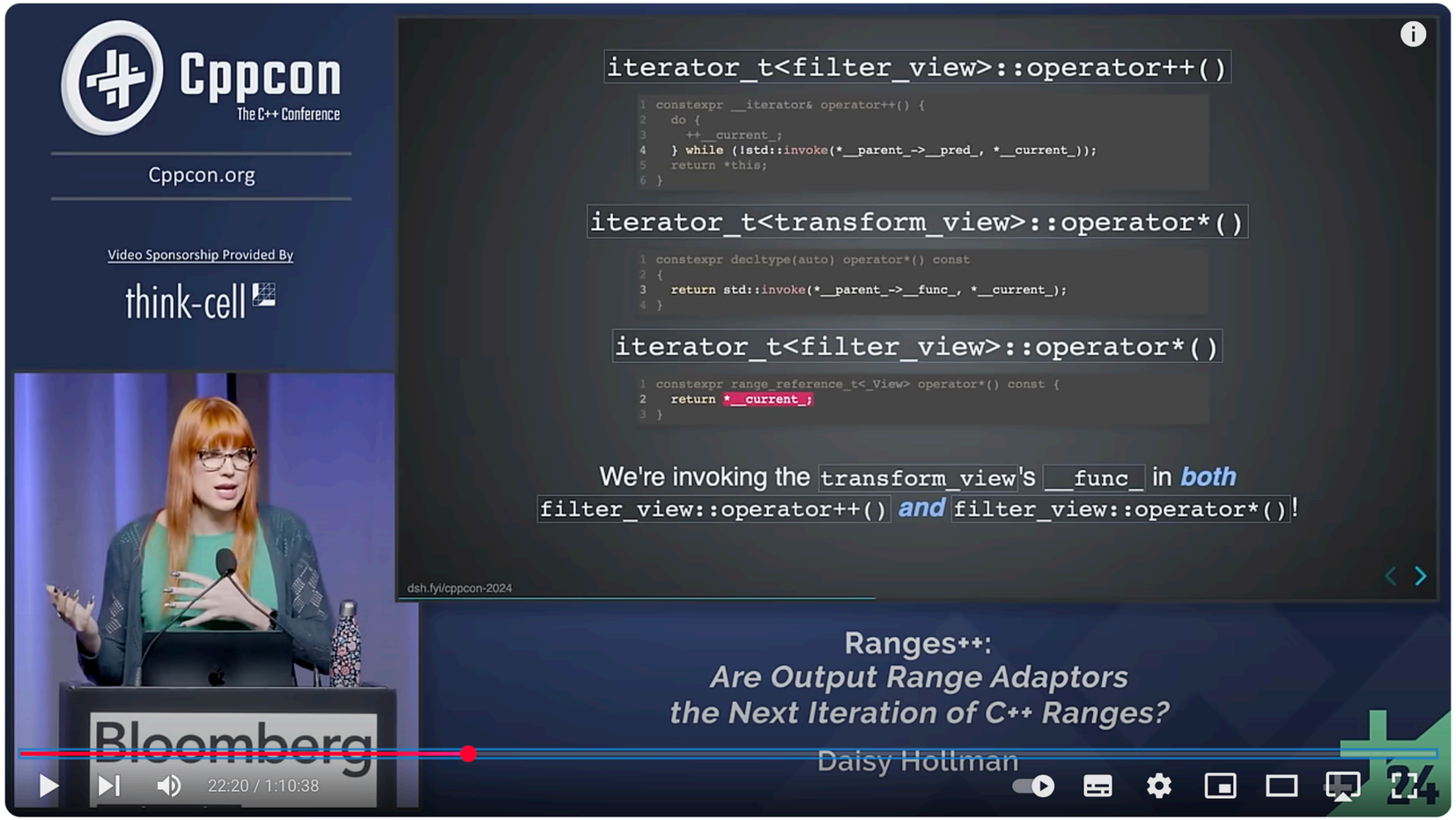
A Safer Iteration Model for C++ - Tristan Brindle - C++ on Sea 2023

Recent Ranges talks



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

Recent Ranges talks



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

Recent Ranges talks

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

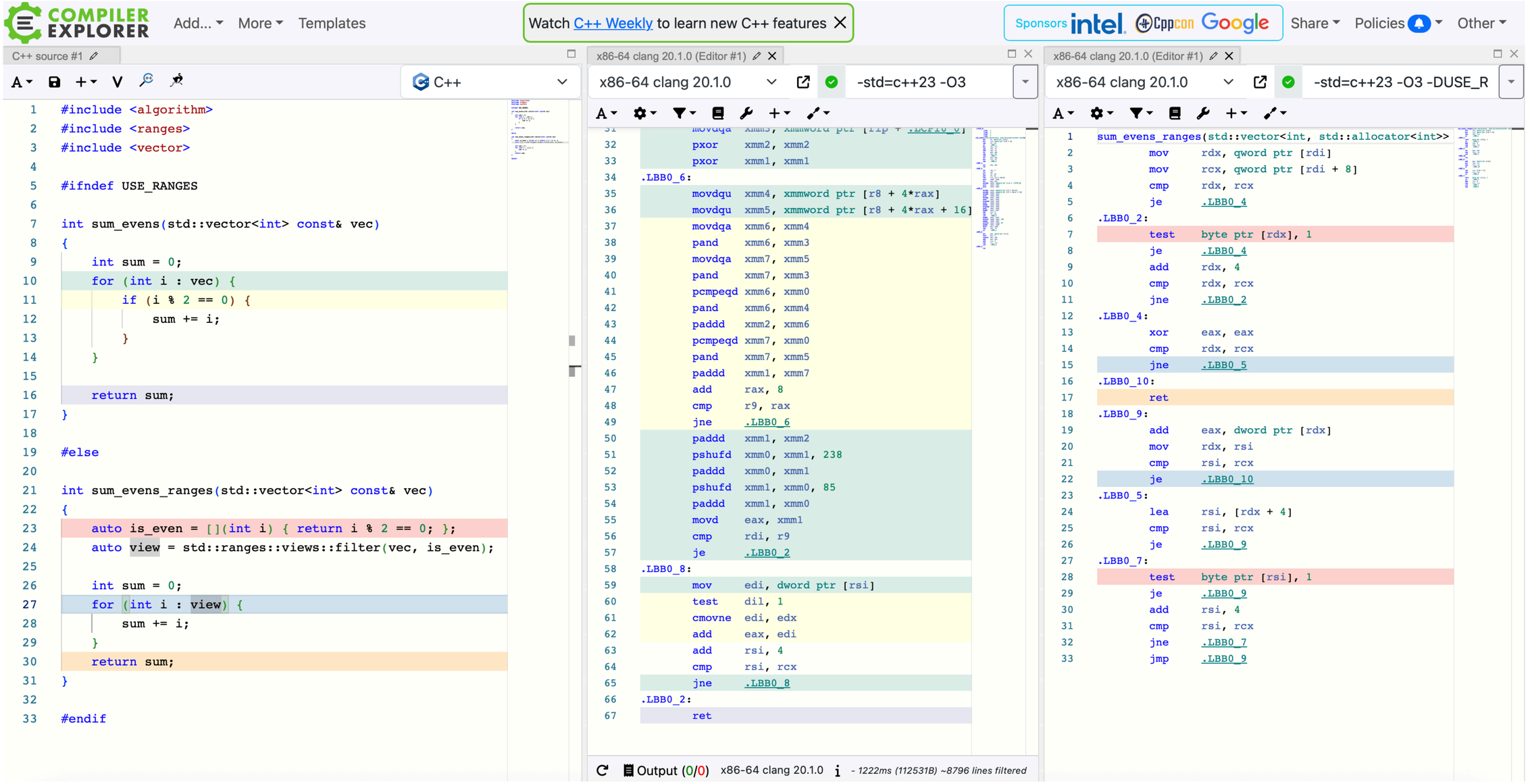
```
void print_evens_v2(std::vector<int> const& vec)
{
    auto is_even = [](int i) { return i % 2 == 0; };

    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




```
#include <algorithm>
#include <ranges>
#include <vector>

#ifndef USE_RANGES

int sum_evens(std::vector<int> const& vec)
{
    int sum = 0;
    for (int i : vec) {
        if (i % 2 == 0) {
            sum += i;
        }
    }

    return sum;
}

#else

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

    int sum = 0;
    for (int i : view) {
        sum += i;
    }
    return sum;
}

#endif
```

```
        pxor    xmm2, xmm2
        pxor    xmm1, xmm1
.LBB0_6:
        movdqu  xmm4, xmmword ptr [r8 + 4*rax]
        movdqu  xmm5, xmmword ptr [r8 + 4*rax + 16]
        movdqa  xmm6, xmm4
        pand    xmm6, xmm3
        movdqa  xmm7, xmm5
        pand    xmm7, xmm3
        pcmpeqd xmm6, xmm0
        pand    xmm6, xmm4
        paddd   xmm2, xmm6
        pcmpeqd xmm7, xmm0
        pand    xmm7, xmm5
        paddd   xmm1, xmm7
        add     rax, 8
        cmp     r9, rax
        jne     .LBB0_6
        paddd   xmm1, xmm2
        pshufd  xmm0, xmm1, 238
        paddd   xmm0, xmm1
        pshufd  xmm1, xmm0, 85
        paddd   xmm1, xmm0
        movd    eax, xmm1
        cmp     rdi, r9
        je      .LBB0_2
.LBB0_8:
        mov     edi, dword ptr [rsi]
        test    dil, 1
        cmovne  edi, edx
        add     eax, edi
        add     rsi, 4
        cmp     rsi, rcx
        jne     .LBB0_8
.LBB0_2:
        ret
```

```
sum_evens_ranges(std::vector<int, std::allocator<int>>
        mov     rdx, qword ptr [rdi]
        mov     rcx, qword ptr [rdi + 8]
        cmp     rdx, rcx
        je      .LBB0_4
.LBB0_2:
        test    byte ptr [rdx], 1
        je      .LBB0_4
        add     rdx, 4
        cmp     rdx, rcx
        jne     .LBB0_2
.LBB0_4:
        xor     eax, eax
        cmp     rdx, rcx
        jne     .LBB0_5
.LBB0_10:
        ret
.LBB0_9:
        add     eax, dword ptr [rdx]
        mov     rdx, rsi
        cmp     rsi, rcx
        je      .LBB0_10
.LBB0_5:
        lea     rsi, [rdx + 4]
        cmp     rsi, rcx
        je      .LBB0_9
.LBB0_7:
        test    byte ptr [rsi], 1
        je      .LBB0_9
        add     rsi, 4
        cmp     rsi, rcx
        jne     .LBB0_7
        jmp     .LBB0_9
```

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 `iterable`s
- 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;
    });
}
```

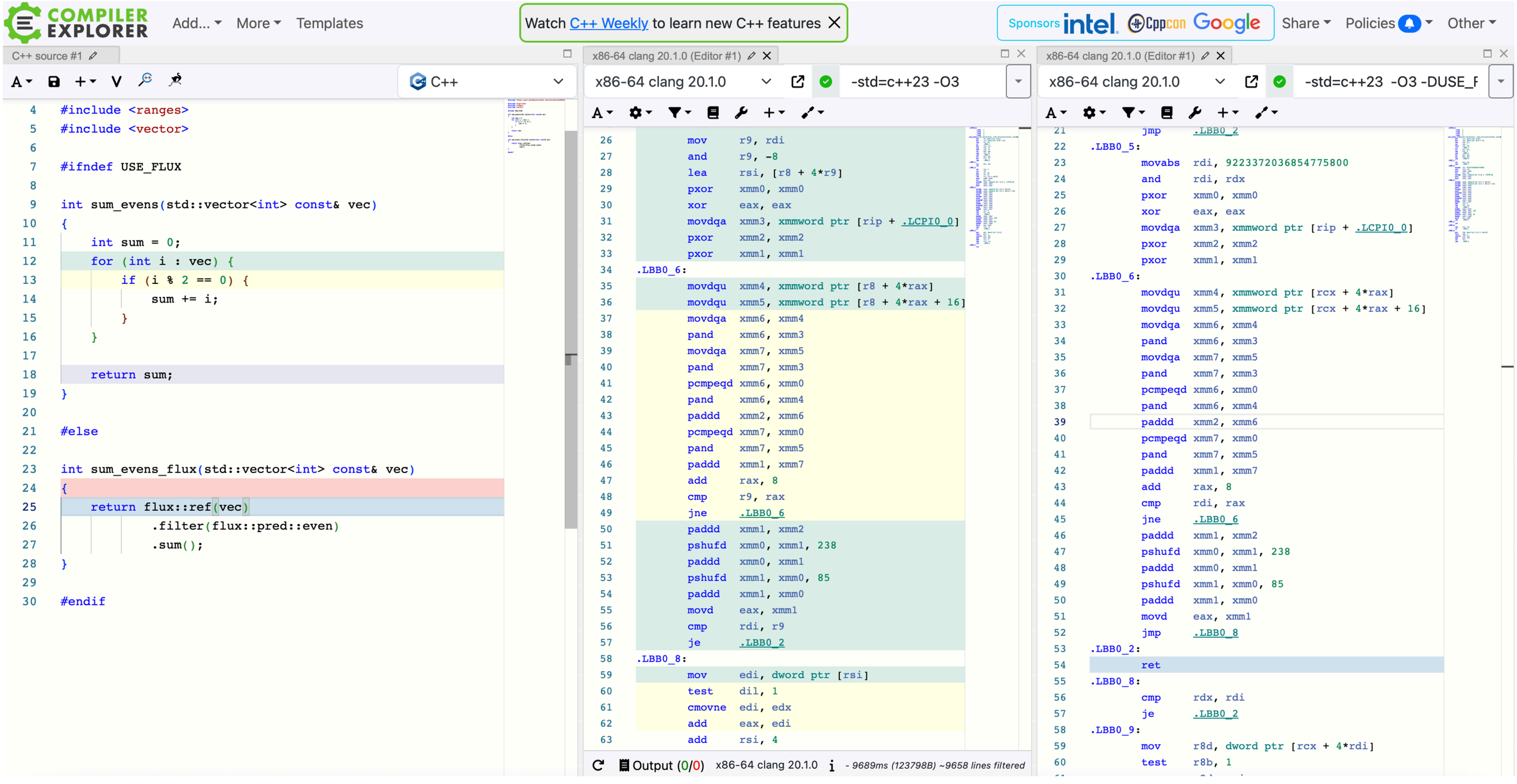
Filter in use

```
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




```
#include <ranges>
#include <vector>

#ifndef USE_FLUX

int sum_evens(std::vector<int> const& vec)
{
    int sum = 0;
    for (int i : vec) {
        if (i % 2 == 0) {
            sum += i;
        }
    }

    return sum;
}

#else

int sum_evens_flux(std::vector<int> const& vec)
{
    return flux::ref(vec)
            .filter(flux::pred::even)
            .sum();
}

#endif
```

```
        mov     r9, rdi
        and     r9, -8
        lea     rsi, [r8 + 4*r9]
        pxor    xmm0, xmm0
        xor     eax, eax
        movdqa  xmm3, xmmword ptr [rip + .LCPI0_0]
        pxor    xmm2, xmm2
        pxor    xmm1, xmm1
.LBB0_6:
        movdqu  xmm4, xmmword ptr [r8 + 4*rax]
        movdqu  xmm5, xmmword ptr [r8 + 4*rax + 16]
        movdqa  xmm6, xmm4
        pand    xmm6, xmm3
        movdqa  xmm7, xmm5
        pand    xmm7, xmm3
        pcmpeqd xmm6, xmm0
        pand    xmm6, xmm4
        paddd   xmm2, xmm6
        pcmpeqd xmm7, xmm0
        pand    xmm7, xmm5
        paddd   xmm1, xmm7
        add     rax, 8
        cmp     r9, rax
        jne     .LBB0_6
        paddd   xmm1, xmm2
        pshufd  xmm0, xmm1, 238
        paddd   xmm0, xmm1
        pshufd  xmm1, xmm0, 85
        paddd   xmm1, xmm0
        movd    eax, xmm1
        cmp     rdi, r9
        je      .LBB0_2
.LBB0_8:
        mov     edi, dword ptr [rsi]
        test    dil, 1
        cmovne  edi, edx
        add     eax, edi
        add     rsi, 4
```

```
        jmp     .LBB0_2
.LBB0_5:
        movabs  rdi, 9223372036854775800
        and     rdi, rdx
        pxor    xmm0, xmm0
        xor     eax, eax
        movdqa  xmm3, xmmword ptr [rip + .LCPI0_0]
        pxor    xmm2, xmm2
        pxor    xmm1, xmm1
.LBB0_6:
        movdqu  xmm4, xmmword ptr [rcx + 4*rax]
        movdqu  xmm5, xmmword ptr [rcx + 4*rax + 16]
        movdqa  xmm6, xmm4
        pand    xmm6, xmm3
        movdqa  xmm7, xmm5
        pand    xmm7, xmm3
        pcmpeqd xmm6, xmm0
        pand    xmm6, xmm4
        paddd   xmm2, xmm6
        pcmpeqd xmm7, xmm0
        pand    xmm7, xmm5
        paddd   xmm1, xmm7
        add     rax, 8
        cmp     rdi, rax
        jne     .LBB0_6
        paddd   xmm1, xmm2
        pshufd  xmm0, xmm1, 238
        paddd   xmm0, xmm1
        pshufd  xmm1, xmm0, 85
        paddd   xmm1, xmm0
        movd    eax, xmm1
        jmp     .LBB0_8
.LBB0_2:
        ret
.LBB0_8:
        cmp     rdx, rdi
        je      .LBB0_2
.LBB0_9:
        mov     r8d, dword ptr [rcx + 4*rdi]
        test    r8b, 1
```

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?