Iteration Revisited

TRISTAN BRINDLE

ITERATION BASICS

```
int arr[] = {1, 2, 3, 4, 5};
size_t sz = 5;

for (size_t i = 0; i < sz; i++) {
    do_something(arr[i]);
}
```

ITERATION BASICS

```
for (size_t i = 0; i < sz; i++) {
    do_something(arr[i]);
}
```

ITERATION BASICS

```
for (int* p = arr; p != arr + sz; ++p) {
    do_something(*p);
}
```

ITERATION BASICS

```
for (int* p = std::begin(arr);
     p ≠ std::end(arr);
     p++) {
    do_something(*p);
}
```

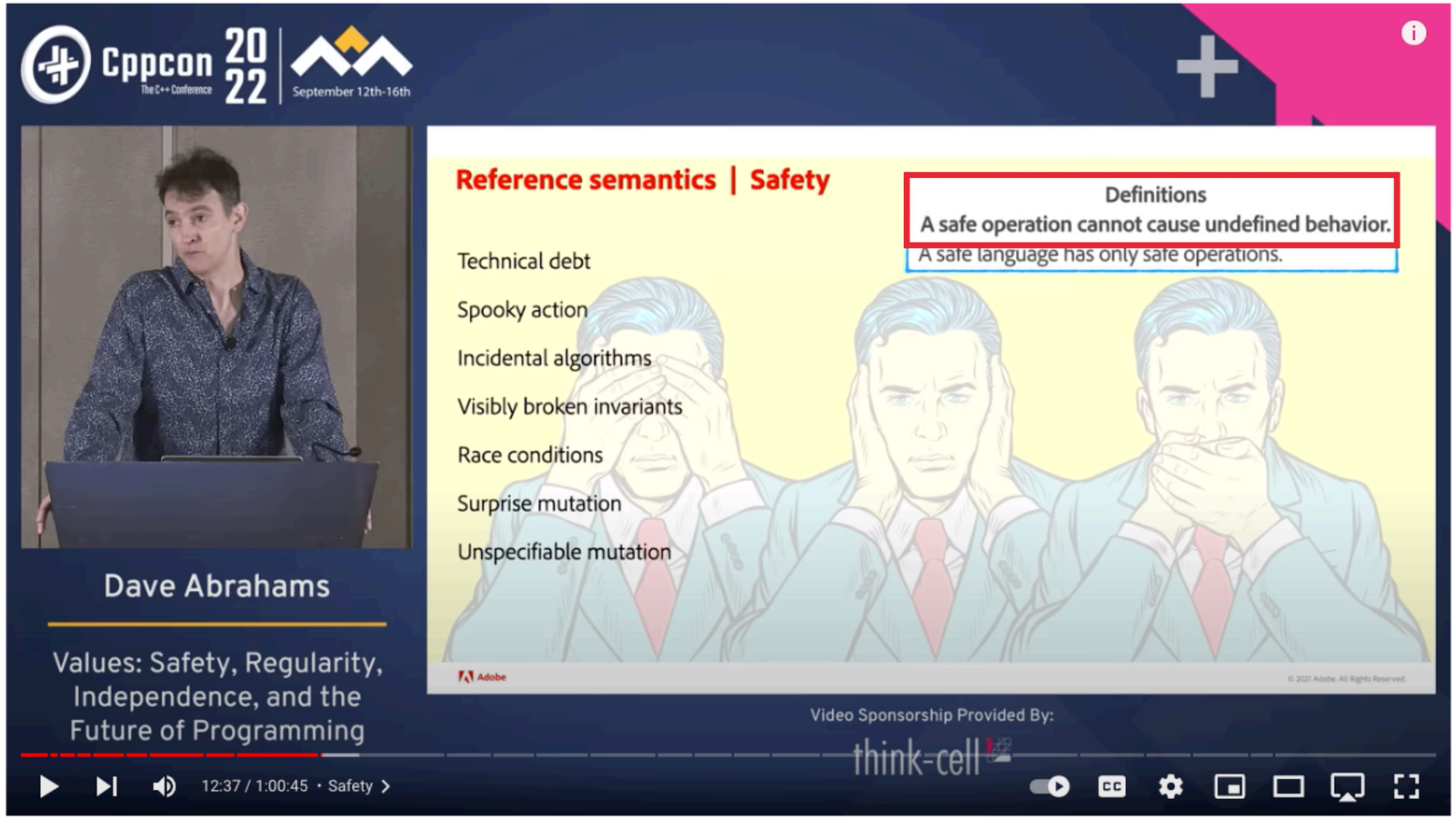
ITERATION BASICS

```
for (auto iter = std::begin(arr);
     iter ≠ std::end(arr);
     ++iter) {
    do_something(*iter);
}
```

STL ITERATORS

- STL iterators are a *generalisation* of array pointers
- Advantages:
  - Powerful
  - Low overhead
  - “Natural” syntax
- Disadvantages:
  - Iterators are **just as unsafe** as raw pointers

WHAT DO I MEAN BY SAFETY?



Value Semantics: Safety, Independence, Projection, & Future of Programming - Dave Abrahams CppCon 22

CppCon
129K subscribers
Subscribed
318
Share
Clip
Save

A SAFE OPERATION CANNOT CAUSE UNDEFINED BEHAVIOUR.

Dave Abrahams

ITERATORS AND UB

```
auto iter = std::end(rng);
do_something(*iter);
```

- *Dereferencing* a past-the-end iterator is UB

ITERATORS AND UB

```
auto iter = std::end(rng);
++iter;
```

- *Incrementing* a past-the-end iterator is UB
  - So is decrementing a begin() iterator
  - So are out-of-bounds random–access jumps

ITERATORS AND UB

```
auto min_iter(const std::vector<int>& vec)
{
    return std::min_element(vec.begin(), vec.end());
}

auto iter = min_iter(get_vector());
std::cout << "Minimum is " << *iter << std::endl;
```

- **Dangling iterators** can occur easily
  - **Any** operation on a dangling iterator is UB

ITERATORS AND UB

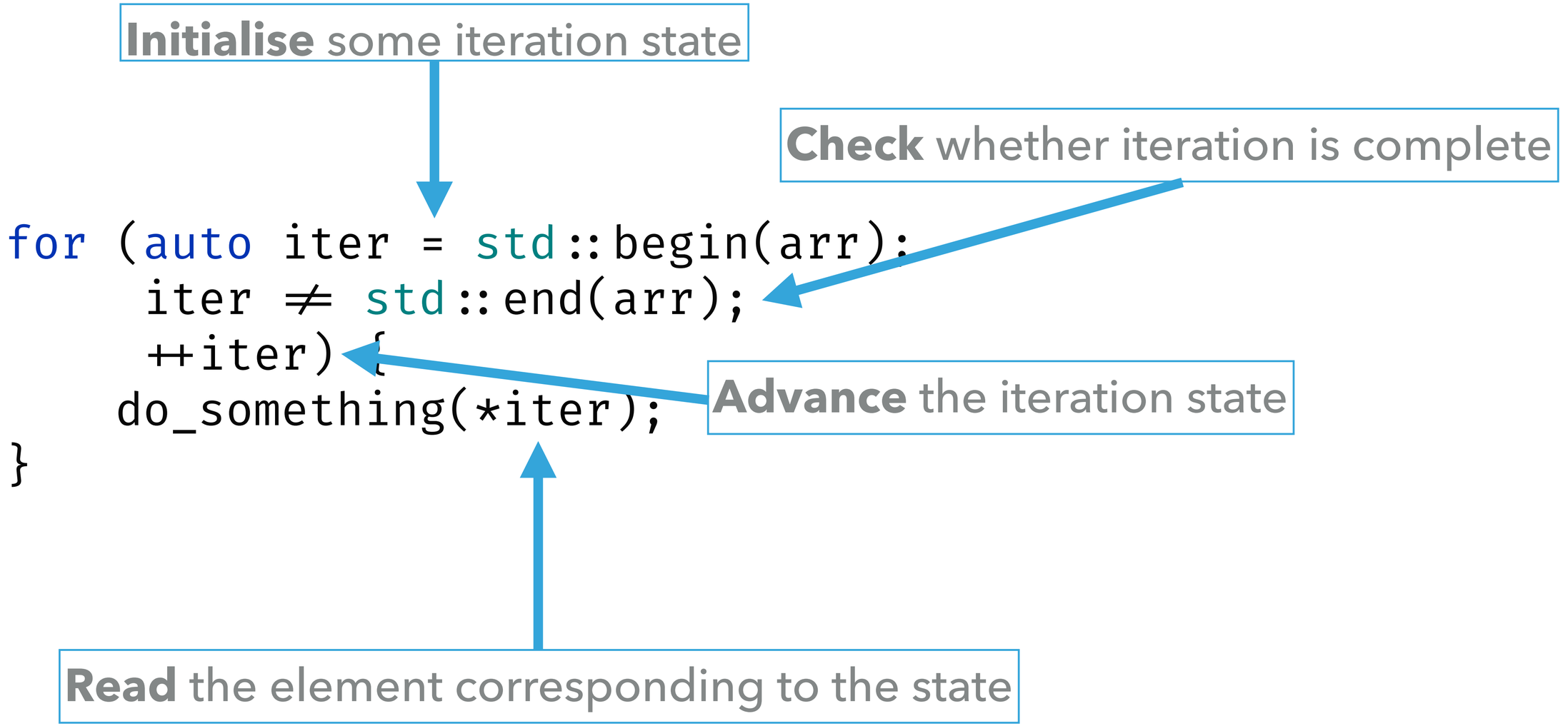
```
auto iter = vec.begin();

vec.push_back(99);

while (iter ≠ vec.end()) {
    ...
}
```

- Iterators can become dangling in non-obvious ways
  - All part of the C++ experience!

ITERATORS AND UB

- “A safe operation cannot cause undefined behaviour”
- Most iterator operations can cause UB
- => STL iterators are a ***fundamentally unsafe*** abstraction

ITERATION FUNDAMENTALS



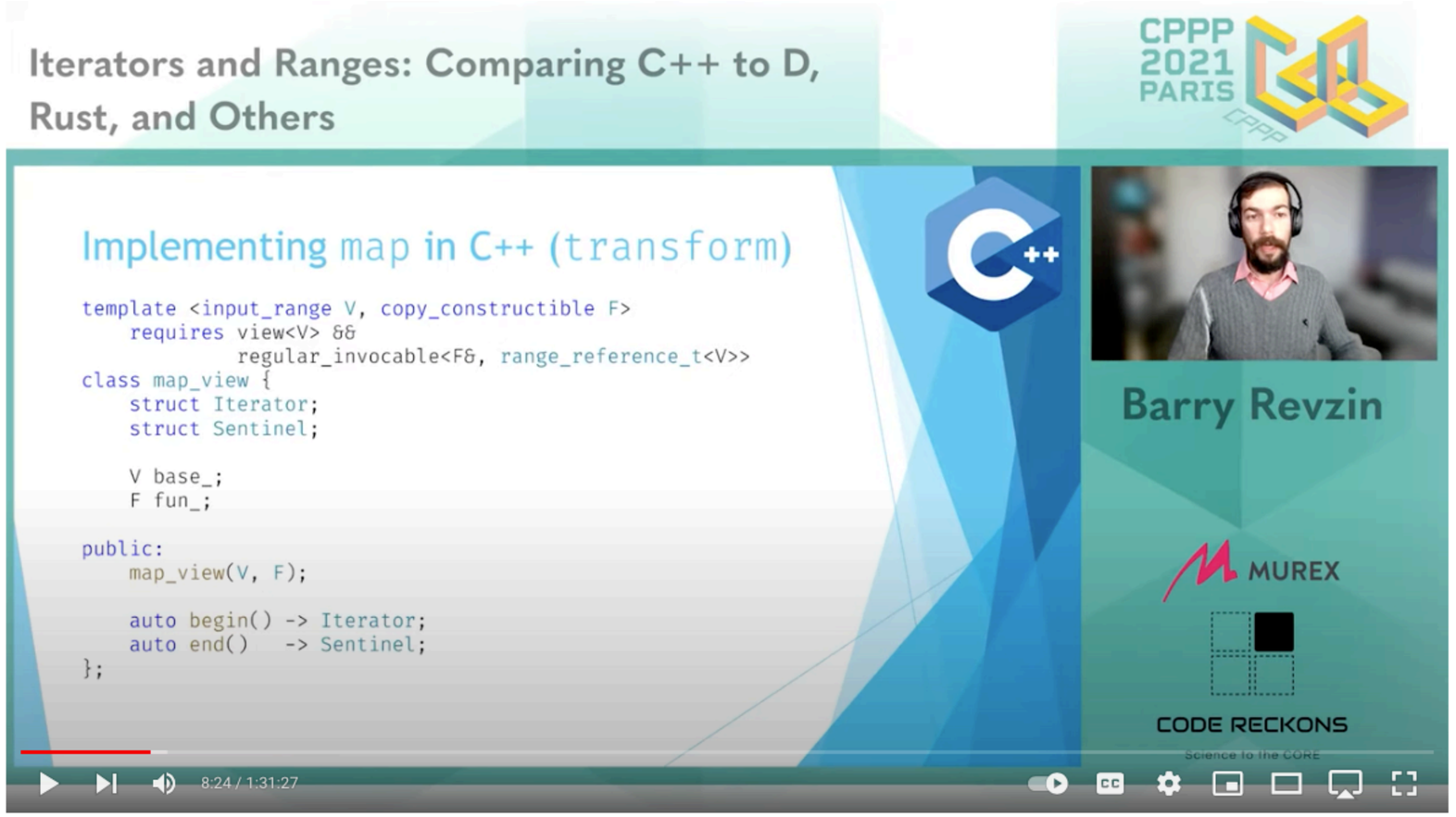
ITERATION FUNDAMENTALS

- **Initialise** some iteration state
- **Check** whether iteration is complete
- **Read** the element corresponding to the state
- **Advance** the iteration state

Rust iterators:

```
next() -> optional<T>
```

ITERATION FUNDAMENTALS



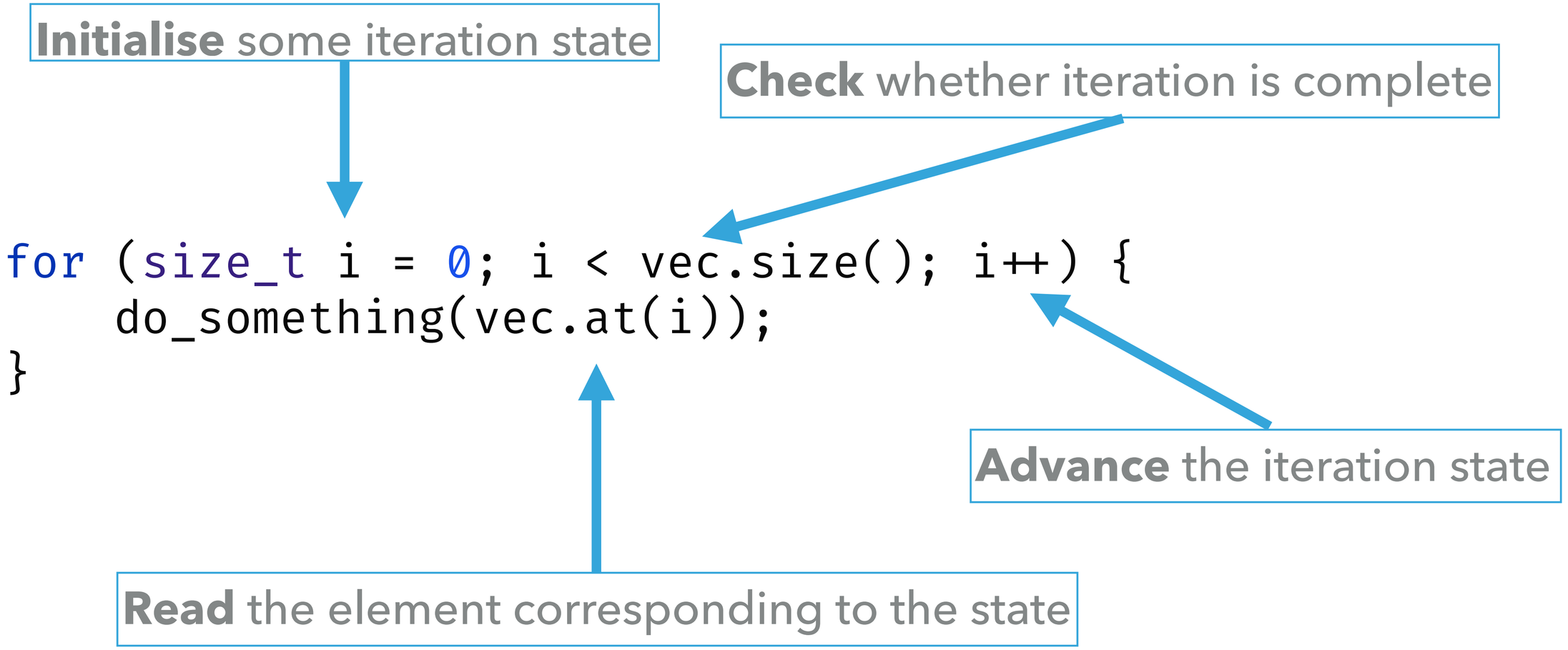
Keynote: Iterators and Ranges: Comparing C++ to D, Rust, and Others - Barry Revzin - CPPP 2021

CPPP Conference
816 subscribers
Subscribe
47
Share
Clip
Save

ITERATION WITH INDICES

```
std::vector<int> vec{1, 2, 3, 4, 5};

for (size_t i = 0; i < vec.size(); i++) {
    do_something(vec.at(i));
}
```

ITERATION WITH INDICES



ITERATION WITH INDICES

```
auto idx = std::size(vec);
do_something(vec.at(idx));
```

- Reading via a past-the-end index is bounds checked
  - No UB!

ITERATION WITH INDICES

```
auto idx = std::size(vec);
++idx;
```

- *Incrementing* a past-the-end index is fine
  - (Assuming unsigned indices or checked operations on signed indices)

ITERATION WITH INDICES

```
size_t min_idx(const std::vector<int>& vec) {
    ...
}

auto idx = min_idx(get_vector());
// std::cout << "Minimum is " << ??? << std::endl;
```

- “Dangling indices” are safe **by design**

ITERATION WITH INDICES

```
size_t idx = 0;

vec.push_back(99);

while (idx < vec.size()) {
    ...
}
```

- “Index invalidation” problems are no longer UB
  - Caught by bounds check at next attempted read

ITERATION WITH INDICES

- Index-based iteration **dramatically reduces** the potential for UB
- …but is still **just as powerful**
- The Big Idea: generalise iteration by *index*, rather than by *pointer*

FLUX

- **Flux** (github.com/tcbrindle/flux) is a C++20 library exploring these ideas
- Aims:
  - Much improved safety by default vs STL iterators
  - Improved ease-of-use and fewer “gotchas”
  - Equal or better runtime performance vs C++20 ranges
  - Compatibility with existing STL code

ITERATION OPERATIONS WITH FLUX

- `flux::first(seq)` returns a *cursor*, which represents a sequence position
  - For contiguous sequences, this is just an integer index
- `flux::is_last(seq, cur)` returns whether the cursor is past-the-end
- `flux::read_at(seq, cur)` accesses the element at the cursor position
- `flux::inc(seq, cur)` increments the cursor

FLUX CONCEPTS

- A type which provides these four operations models the `sequence` concept
- Sequences are single-pass by default
- *Multipass* sequences are those which allow simultaneous iteration by independent cursors
- A sequence whose cursor is a *regular type* is assumed to be multipass
  - Regular = default constructible, copyable, equality comparable
  - Sequences can opt out of being multipass if necessary

FLUX CONCEPTS

- *Bidirectional sequences* are multipass sequences which additionally allow their cursors to be *decremented*:
  - `flux::dec(seq, cur)`
- *Random-access sequences* provide additional operations:
  - `flux::inc(seq, cur, offset)`
  - `flux::distance(seq, from_cur, to_cur)`
  - Cursors are `std::totally_ordered`
- *Contiguous sequences* are random-access, backed by an in-memory array

FLUX SEQUENCE ALGORITHMS

- As of October 2023

all
any
compare
contains
count
count_eq
count_if
contains
ends_with
equal
fill
find
find_if
find_if_not
find_max
find_min
find_minmax
fold
fold_first
for_each
for_each_while
inplace_reverse
max
min
minmax
none
output_to
product
search
sort
starts_with
sum
swap_elements
to
write_to
zip_find_if
zip_fold
zip_for_each
zip_for_each_while

FLUX PIPELINES

- Like C++20 Ranges, Flux provides a variety of *sequence adaptors* performing lazy evaluation
- Typically these are composed into pipelines
- A pipeline begins with some sequence (the *source*), passes through zero or more *adaptors*, and then finishes with a call to an *algorithm*

FLUX PIPELINE EXAMPLE

```
std::vector<int> vec = get_vector();

int max_even_sq = flux::ref(vec)                  // iterate over a reference to vec
                    .filter(flux::pred::even) // keep only even numbers
                    .map([](int i) { return i * i; }) // square each int
                    .max()                        // find the maximum value
                    .value_or(0);                 // handle empty sequences
```

FLUX PIPELINE EXAMPLE

```
std::vector<int> vec = get_vector();

int max_even_sq = flux::ref(vec)                  // iterate over a reference to vec
                    ._(flux::filter, flux::pred::even) // keep only even numbers
                    ._(flux::map, [](int i) { return i * i; }) // square each int
                    ._(flux::max)                 // find the maximum value
                    .value_or(0);                 // handle empty sequences
```

FLUX SEQUENCE ADAPTORS

- As of October 2023

adjacent
adjacent_filter
adjacent_map
cache_last
cartesian_product
cartesian_product_with
chain
chunk
chunk_by
cursors
cycle
deduce
drop
drop_while
filter
flatten
map
mask
pairwise
pairwise_map
prescan
read_only
reverse
scan
scan_first
set_difference
set_intersection
set_symmetric_difference
set_union
slide
stride
split
take
take_while
unchecked
zip

IMPLEMENTING THE SEQUENCE PROTOCOL

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

template <typename T, std::size_t N>
struct flux::sequence_traits<my_array<T, N>> {
    using self_t = my_array<T, N>;

    static size_t first(const self_t&) { return 0; }
    static bool is_last(const self_t&, size_t cur) { return cur >= N; }
    static void inc(const self_t&, size_t& cur) { ++cur; }
    static decltype(auto) read_at(auto& self, size_t cur) {
        flux::bounds_check(cur < N);
        return self.data[cur];
    }
};
static_assert(flux::multipass_sequence<my_array<int, 10>>);
```

IMPLEMENTING THE SEQUENCE PROTOCOL

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

    struct flux_sequence_traits {
        static size_t first(const my_array&) { return 0; }
        static bool is_last(const my_array&, size_t cur) { return cur >= N; }
        static void inc(const my_array&, size_t& cur) { ++cur; }
        static decltype(auto) read_at(auto& self, size_t cur)
        {
            flux::bounds_check(cur < N);
            return self.data[cur];
        }
    };
};
static_assert(flux::multipass_sequence<my_array<int, 10>>);
```

IMPLEMENTING THE SEQUENCE PROTOCOL

- Considerably easier to implement vs STL ranges
  - Even more so for higher sequence/range categories
- No `iterator`/`const_iterator` confusion: the same cursor type is used for const and non-const access

RANGES COMPATIBILITY

- Every C++20 `contiguous_range` is automatically a `flux::contiguous_sequence`
  - Contiguous => “cheaply indexable”
  - Covers vector, array, string, string_view, span…
- Other ranges can be wrapped in a sequence implementation using `flux::from_range(rng)`, with reduced safety guarantees
  - No worse than using ranges directly

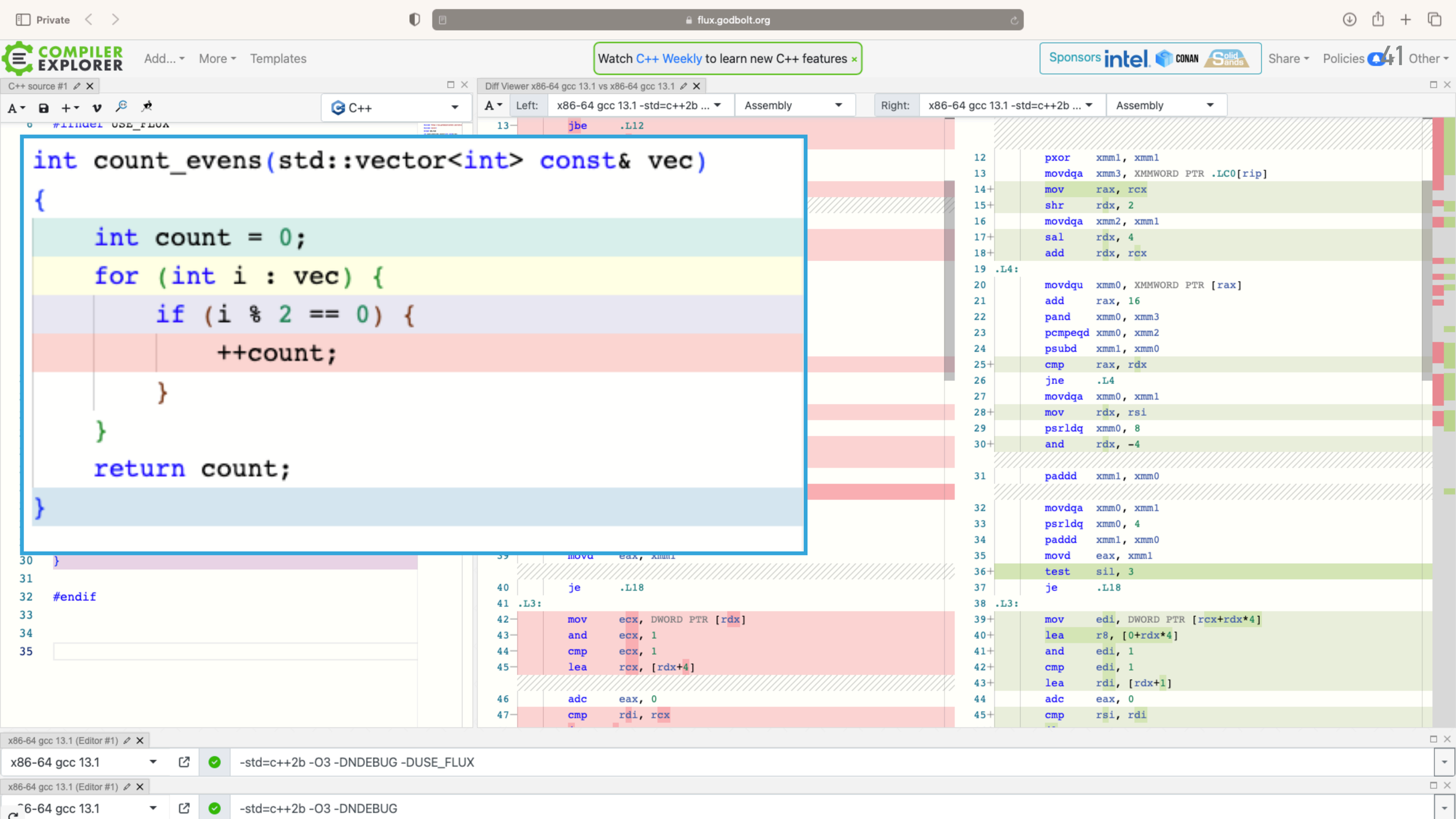
RANGES COMPATIBILITY

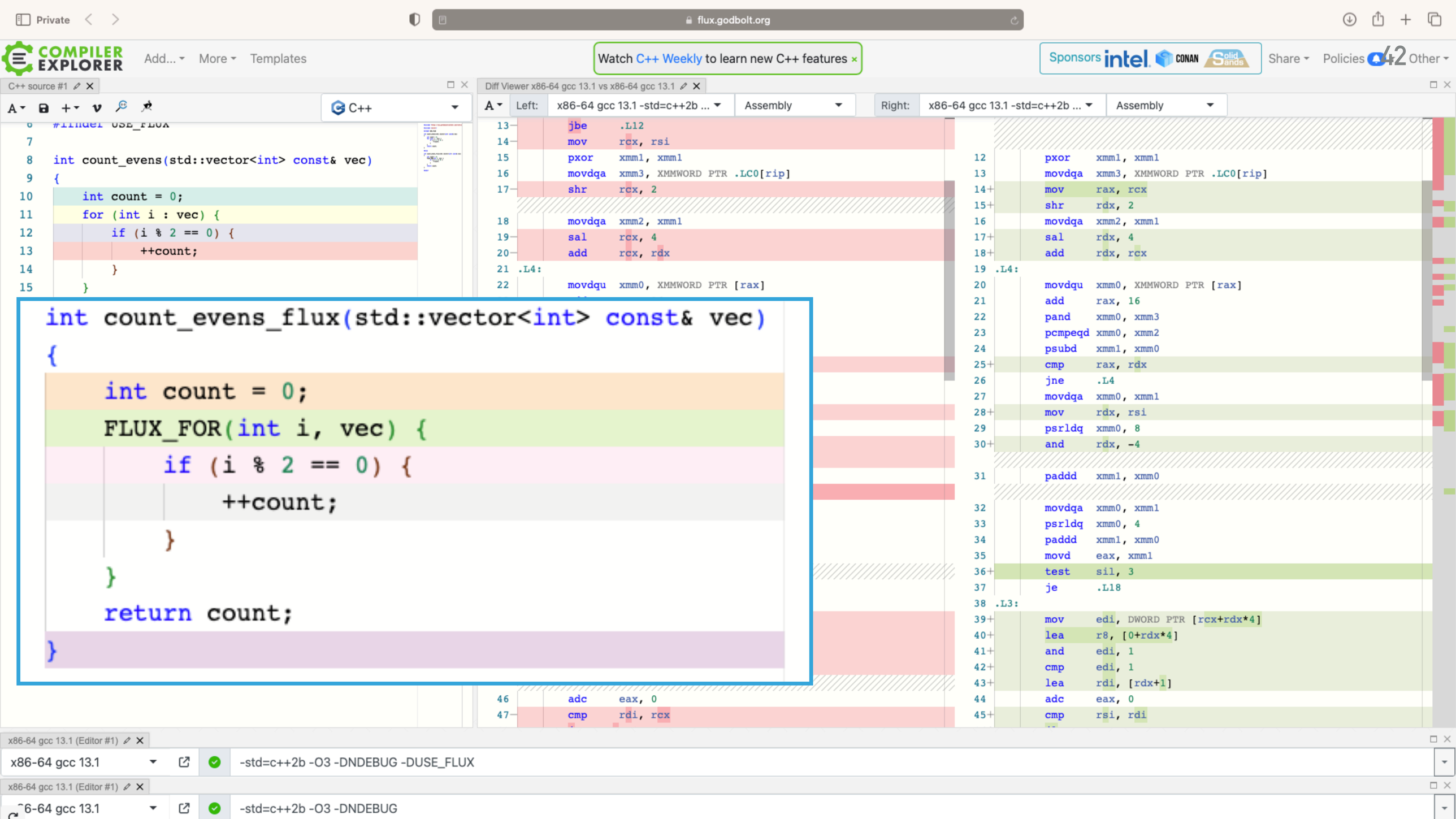
- Conversely, every `flux::sequence` provides STL-compatible iterators
  - So we can use them with C++20 algorithms and range adaptors
  - ...and range-for loops

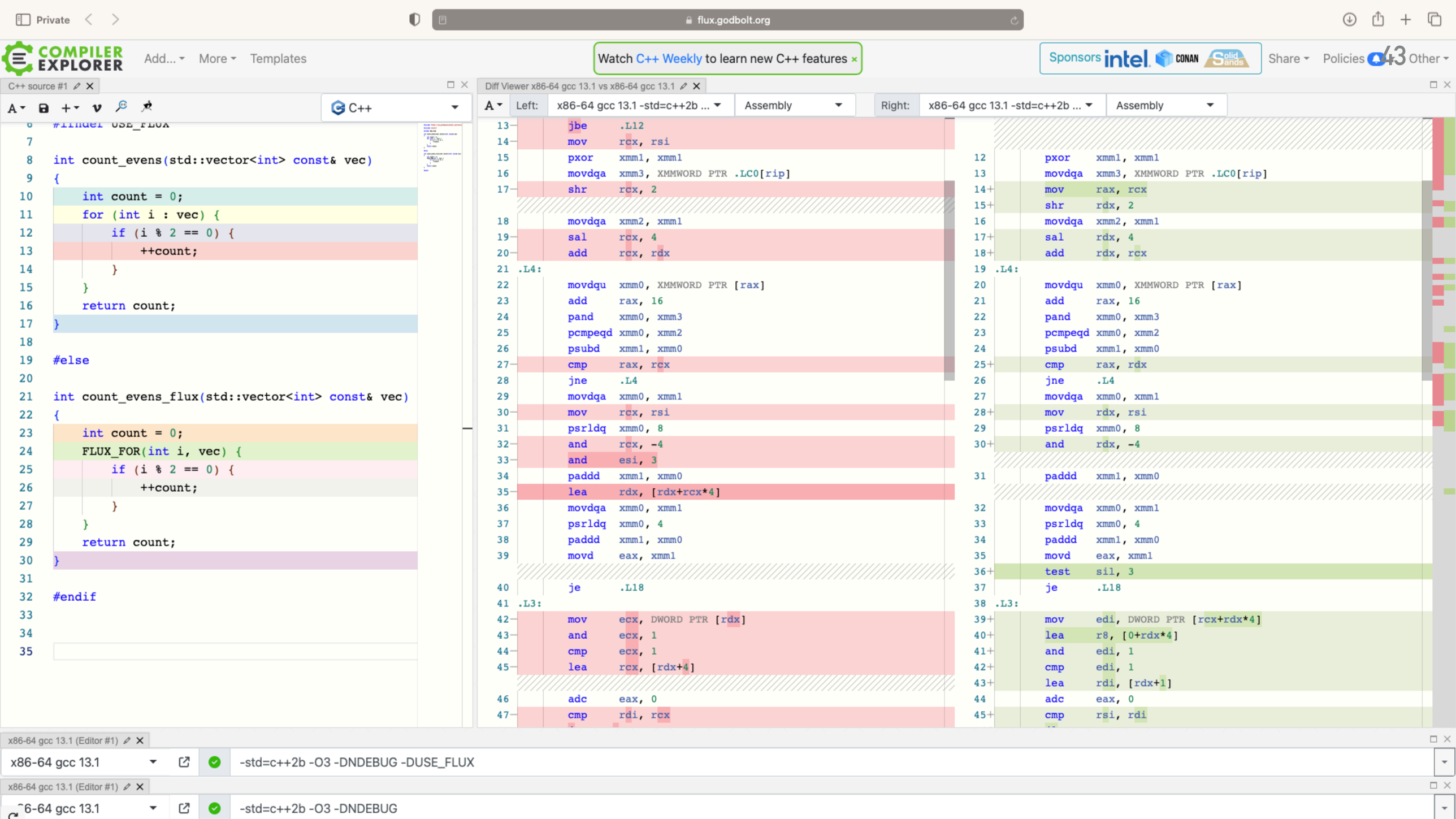
```
template <flux::sequence Seq>
struct sequence_iterator {
    Seq* seq;
    flux::cursor_t<Seq> cur;

    sequence_iterator& operator++() {
        flux::inc(*seq, cur);
        return *this;
    }

    // etc…
};
```

PERFORMANCE

- Compilers are **incredibly good** at eliding bounds checks
- Many (most?) new languages use bounds checking universally
- => Optimisers can recognise those patterns in very many cases
- For example: https://flux.godbolt.org/z/v35dsd7sx


Private
flux.godbolt.org
COMPILER EXPLORER
Add...
More
Templates
Watch C++ Weekly to learn new C++ features
Sponsors intel CONAN Solid Sands
Share
Policies
Other
C++ source #1
C++
Diff Viewer x86-64 gcc 13.1 vs x86-64 gcc 13.1
Left:
x86-64 gcc 13.1 -std=c++2b ...
Assembly
Right:
x86-64 gcc 13.1 -std=c++2b ...
Assembly
int count_evens(std::vector<int> const& vec)
{
int count = 0;
for (int i : vec) {
if (i % 2 == 0) {
++count;
}
}
return count;
}
#endif
x86-64 gcc 13.1 (Editor #1)
x86-64 gcc 13.1
-std=c++2b -O3 -DNDEBUG -DUSE_FLUX
x86-64 gcc 13.1 (Editor #1)
-std=c++2b -O3 -DNDEBUG


Private
flux.godbolt.org
COMPILER EXPLORER
Add...
More
Templates
Watch C++ Weekly to learn new C++ features
Sponsors intel CONAN Solid Sands
Share
Policies
42
Other
C++ source #1
C++
Diff Viewer x86-64 gcc 13.1 vs x86-64 gcc 13.1
Left:
x86-64 gcc 13.1 -std=c++2b ...
Assembly
Right:
x86-64 gcc 13.1 -std=c++2b ...
Assembly
#ifndef USE_FLUX
int count_evens(std::vector<int> const& vec)
{
int count = 0;
for (int i : vec) {
if (i % 2 == 0) {
++count;
}
}
int count_evens_flux(std::vector<int> const& vec)
{
int count = 0;
FLUX_FOR(int i, vec) {
if (i % 2 == 0) {
++count;
}
}
return count;
}
jbe .L12
mov rcx, rsi
pxor xmm1, xmm1
movdqa xmm3, XMMWORD PTR .LC0[rip]
shr rcx, 2
movdqa xmm2, xmm1
sal rcx, 4
add rcx, rdx
.L4:
movdqu xmm0, XMMWORD PTR [rax]
adc eax, 0
cmp rdi, rcx
pxor xmm1, xmm1
movdqa xmm3, XMMWORD PTR .LC0[rip]
mov rax, rcx
shr rdx, 2
movdqa xmm2, xmm1
sal rdx, 4
add rdx, rcx
.L4:
movdqu xmm0, XMMWORD PTR [rax]
add rax, 16
pand xmm0, xmm3
pcmpeqd xmm0, xmm2
psubd xmm1, xmm0
cmp rax, rdx
jne .L4
movdqa xmm0, xmm1
mov rdx, rsi
psrldq xmm0, 8
and rdx, -4
paddd xmm1, xmm0
movdqa xmm0, xmm1
psrldq xmm0, 4
paddd xmm1, xmm0
movd eax, xmm1
test sil, 3
je .L18
.L3:
mov edi, DWORD PTR [rcx+rdx*4]
lea r8, [0+rdx*4]
and edi, 1
cmp edi, 1
lea rdi, [rdx+1]
adc eax, 0
cmp rsi, rdi
x86-64 gcc 13.1 (Editor #1)
x86-64 gcc 13.1
-std=c++2b -O3 -DNDEBUG -DUSE_FLUX
x86-64 gcc 13.1 (Editor #1)
-std=c++2b -O3 -DNDEBUG


Private
flux.godbolt.org
COMPILER EXPLORER
Add...
More
Templates
Watch C++ Weekly to learn new C++ features
Sponsors intel CONAN Solid Sands
Share
Policies
43
Other
C++ source #1
C++
#ifndef USE_FLUX
int count_evens(std::vector<int> const& vec)
{
int count = 0;
for (int i : vec) {
if (i % 2 == 0) {
++count;
}
}
return count;
}
#else
int count_evens_flux(std::vector<int> const& vec)
{
int count = 0;
FLUX_FOR(int i, vec) {
if (i % 2 == 0) {
++count;
}
}
return count;
}
#endif
Diff Viewer x86-64 gcc 13.1 vs x86-64 gcc 13.1
Left:
x86-64 gcc 13.1 -std=c++2b ...
Assembly
Right:
x86-64 gcc 13.1 -std=c++2b ...
Assembly
jbe .L12
mov rcx, rsi
pxor xmm1, xmm1
movdqa xmm3, XMMWORD PTR .LC0[rip]
shr rcx, 2
movdqa xmm2, xmm1
sal rcx, 4
add rcx, rdx
.L4:
movdqu xmm0, XMMWORD PTR [rax]
add rax, 16
pand xmm0, xmm3
pcmpeqd xmm0, xmm2
psubd xmm1, xmm0
cmp rax, rcx
jne .L4
movdqa xmm0, xmm1
mov rcx, rsi
psrldq xmm0, 8
and rcx, -4
and esi, 3
paddd xmm1, xmm0
lea rdx, [rdx+rcx*4]
movdqa xmm0, xmm1
psrldq xmm0, 4
paddd xmm1, xmm0
movd eax, xmm1
je .L18
.L3:
mov ecx, DWORD PTR [rdx]
and ecx, 1
cmp ecx, 1
lea rcx, [rdx+4]
adc eax, 0
cmp rdi, rcx
pxor xmm1, xmm1
movdqa xmm3, XMMWORD PTR .LC0[rip]
mov rax, rcx
shr rdx, 2
movdqa xmm2, xmm1
sal rdx, 4
add rdx, rcx
.L4:
movdqu xmm0, XMMWORD PTR [rax]
add rax, 16
pand xmm0, xmm3
pcmpeqd xmm0, xmm2
psubd xmm1, xmm0
cmp rax, rdx
jne .L4
movdqa xmm0, xmm1
mov rdx, rsi
psrldq xmm0, 8
and rdx, -4
paddd xmm1, xmm0
movdqa xmm0, xmm1
psrldq xmm0, 4
paddd xmm1, xmm0
movd eax, xmm1
test sil, 3
je .L18
.L3:
mov edi, DWORD PTR [rcx+rdx*4]
lea r8, [0+rdx*4]
and edi, 1
cmp edi, 1
lea rdi, [rdx+1]
adc eax, 0
cmp rsi, rdi
x86-64 gcc 13.1 (Editor #1)
x86-64 gcc 13.1
-std=c++2b -O3 -DNDEBUG -DUSE_FLUX
x86-64 gcc 13.1 (Editor #1)
-std=c++2b -O3 -DNDEBUG

PERFORMANCE

- This implementation is using the `FLUX_FOR()` macro to force iteration using Flux primitives (`first`, `is_last`, `read_at`, `inc`)
- Idiomatic use of Flux would be to use a named algorithm instead, i.e. `flux::count_if()`

C++ source #1

```cpp
#ifndef USE_FLUX

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

#else

int count_evens_flux(std::vector<int> const& vec)
{
    return flux::count_if(vec, flux::pred::even);
}

#endif
```

Diff Viewer x86-64 clang 16.0.0 vs x86-64 clang 16.0.0

Left: x86-64 clang 16.0.0 -std=c+... | Assembly

```
.LCPI0_0:
        .long   1                               # 0x1
        .long   1                               # 0x1
        .long   1                               # 0x1
        .long   1                               # 0x1
count_evens_flux(std::vector<int, std::allocator<int> > const&): # @
        mov     r8, qword ptr [rdi]
        mov     rcx, qword ptr [rdi + 8]
        cmp     r8, rcx
        je      .LBB0_1
        mov     rsi, rcx
        sub     rsi, r8
        add     rsi, -4
        xor     eax, eax
        cmp     rsi, 28
        jae     .LBB0_4
        mov     rdx, r8
        jmp     .LBB0_7
.LBB0_1:
        xor     eax, eax
        ret
.LBB0_4:
        shr     rsi, 2
        inc     rsi
        mov     rdi, rsi
        and     rdi, -8
        lea     rdx, [r8 + 4*rdi]
        pxor    xmm0, xmm0
        xor     eax, eax
        movdqa  xmm2, xmmword ptr [rip + .LCPI0_0] # xmm2 = [1,1,1,1
        pxor    xmm1, xmm1
.LBB0_5:                                # =>This Inner Loop Header: I
        movdqu  xmm3, xmmword ptr [r8 + 4*rax]
        movdqu  xmm4, xmmword ptr [r8 + 4*rax + 16]
        pandn   xmm3, xmm2
        paddd   xmm0, xmm3
        pandn   xmm4, xmm2
        paddd   xmm1, xmm4
```

Right: x86-64 clang 16.0.0 -std=c+... | Assembly

```
.LCPI0_0:
        .long   1                               # 0x1
        .long   1                               # 0x1
        .long   1                               # 0x1
        .long   1                               # 0x1
count_evens(std::vector<int, std::allocator<int> > const&):     # @c
        mov     r8, qword ptr [rdi]
        mov     rcx, qword ptr [rdi + 8]
        cmp     r8, rcx
        je      .LBB0_1
        mov     rsi, rcx
        sub     rsi, r8
        add     rsi, -4
        xor     eax, eax
        cmp     rsi, 28
        jae     .LBB0_4
        mov     rdx, r8
        jmp     .LBB0_7
.LBB0_1:
        xor     eax, eax
        ret
.LBB0_4:
        shr     rsi, 2
        inc     rsi
        mov     rdi, rsi
        and     rdi, -8
        lea     rdx, [r8 + 4*rdi]
        pxor    xmm0, xmm0
        xor     eax, eax
        movdqa  xmm2, xmmword ptr [rip + .LCPI0_0] # xmm2 = [1,1,1,1
        pxor    xmm1, xmm1
.LBB0_5:                                # =>This Inner Loop Header: I
        movdqu  xmm3, xmmword ptr [r8 + 4*rax]
        movdqu  xmm4, xmmword ptr [r8 + 4*rax + 16]
        pandn   xmm3, xmm2
        paddd   xmm0, xmm3
        pandn   xmm4, xmm2
        paddd   xmm1, xmm4
```

x86-64 clang 16.0.0 (Editor #1)

x86-64 clang 16.0.0 -std=c++2b -O3 -DNDEBUG -DUSE_FLUX

x86-64 clang 16.0.0 (Editor #1)

6-64 clang 16.0.0 -std=c++2b -O3 -DNDEBUG

INTERNAL ITERATION

- The Flux sequence protocol has an extra, optional customisation point: `for_each_while(predicate)`
- This instructs the sequence to perform *internal iteration* until the predicate returns `false`
- Many sequence adaptors can perform internal iteration more efficiently than by a check/read/advance loop
- Many algorithms can be written in terms for `for_each_while` (e.g. `fold()`, `find_if()`, `min()`)
- => Flux pipelines can in many cases generate more efficient code than equivalent C++20 Ranges pipelines

C++ source #1

```cpp
#include <cstddef>
#include <stdint.h>
#include <vector>

#ifdef USE_FLUX

#include "https://raw.githubusercontent.com/tcbrindle/flux/main/single_i

int sum_sq_even_flux(std::vector<int> const& vec)
{
    return flux::ref(vec)
            filter(flux::pred::even)
```

```cpp
int sum_sq_even_ranges(std::vector<int> const& vec)
{
    auto view = vec
        | std::views::filter([](int i) { return i % 2 == 0; })
        | std::views::transform([](int i) { return i * i; });
    return std::accumulate(view.begin(), view.end(), 0);
}
```

x86-64 clang 16.0.0 (Editor #1)

x86-64 clang 16.0.0 -std=c++20 -O3 -DNDEBUG -m

```
        vpxor   xmm4, xmm4, xmm4
LBB0_5:                                 # =>This Inner Loop Header: Depth=1
        vmovdqu ymm5, ymmword ptr [r8 + 4*rax]
        vmovdqu ymm6, ymmword ptr [r8 + 4*rax + 32]
        vmovdqu ymm7, ymmword ptr [r8 + 4*rax + 64]
        vmovdqu ymm8, ymmword ptr [r8 + 4*rax + 96]
        vptestnmd       k1, ymm5, ymm1
        vptestnmd       k2, ymm6, ymm1
        vpshufd xmm1, xmm0, 238                 # xmm1 = xmm0[2,3,2,3]
        vpaddd  xmm0, xmm0, xmm1
        vpshufd xmm1, xmm0, 85                  # xmm1 = xmm0[1,1,1,1]
        vpaddd  xmm0, xmm0, xmm1
        vmovd   eax, xmm0
        cmp     rdi, r9
        je      .LBB0_8
LBB0_7:                                 # =>This Inner Loop Header: Depth=1
        mov     edi, dword ptr [rsi]
        mov     r8d, edi
        imul    r8d, edi
        test    dil, 1
        cmovne  r8d, edx
        add     eax, r8d
        add     rsi, 4
        cmp     rsi, rcx
        jne     .LBB0_7
```

x86-64 clang 16.0.0 (Editor #1)

x86-64 clang 16.0.0 -std=c++20 -O3 -DNDEBUG -R

```
sum_sq_even_ranges(std::vector<int, std::allocator<int> > const&): # @sum_
        mov     rcx, qword ptr [rdi]
        mov     rdx, qword ptr [rdi + 8]
        cmp     rcx, rdx
        je      .LBB0_4
.LBB0_1:                                # =>This Inner Loop Header: Depth=
        test    byte ptr [rcx], 1
        je      .LBB0_4
        add     rcx, 4
        cmp     rcx, rdx
        jne     .LBB0_1
        jmp     .LBB0_3
.LBB0_4:
        cmp     rcx, rdx
        je      .LBB0_3
        mov     esi, dword ptr [rcx]
        xor     eax, eax
.LBB0_6:                                # =>This Loop Header: Depth=1
        mov     edi, eax
        imul    esi, esi
        mov     eax, esi
        add     eax, edi
.LBB0_7:                                #   Parent Loop BB0_6 Depth=1
        add     rcx, 4
        cmp     rcx, rdx
        je      .LBB0_10
        mov     esi, dword ptr [rcx]
        test    sil, 1
        jne     .LBB0_7
        cmp     rcx, rdx
        jne     .LBB0_6
.LBB0_10:
        ret
.LBB0_3:
        xor     eax, eax
        ret
```

C++ source #1

```
int sum_sq_even_flux(std::vector<int> const& vec)
{
    return flux::ref(vec)
        .filter(flux::pred::even)
        .map([](int i) { return i * i; })
        .sum();
}
```

```
#include <cstddef>

#ifdef __cpp_lib_ranges
#include <numeric>
#include <ranges>
#include <vector>

int sum_sq_even_ranges(std::vector<int> const& vec)
{
    auto view = vec
        | std::views::filter([](int i) { return i % 2 == 0; })
        | std::views::transform([](int i) { return i * i; });
    return std::accumulate(view.begin(), view.end(), 0);
}

#endif
```

x86-64 clang 16.0.0 (Editor #1)

x86-64 clang 16.0.0 -std=c++20 -O3 -DNDEBUG -m

```
        vpaddd  ymm4 {k4}, ymm4, ymm8
        add     rax, 32
        cmp     r9, rax
        jne     .LBB0_5
        vpaddd  ymm0, ymm2, ymm0
        vpaddd  ymm1, ymm4, ymm3
        vpaddd  ymm0, ymm1, ymm0
        vextracti128    xmm1, ymm0, 1
        vpaddd  xmm0, xmm0, xmm1
        vpshufd xmm1, xmm0, 238                 # xmm1 = xmm0[2,3,2,3]
        vpaddd  xmm0, xmm0, xmm1
        vpshufd xmm1, xmm0, 85                  # xmm1 = xmm0[1,1,1,1]
        vpaddd  xmm0, xmm0, xmm1
        vmovd   eax, xmm0
        cmp     rdi, r9
        je      .LBB0_8
.LBB0_7:                                # =>This Inner Loop Header: Depth=1
        mov     edi, dword ptr [rsi]
        mov     r8d, edi
        imul    r8d, edi
        test    dil, 1
        cmovne  r8d, edx
        add     eax, r8d
        add     rsi, 4
        cmp     rsi, rcx
        jne     .LBB0_7
```

x86-64 clang 16.0.0 (Editor #1)

x86-64 clang 16.0.0 -std=c++20 -O3 -DNDEBUG -R

```
sum_sq_even_ranges(std::vector<int, std::allocator<int> > const&): # @sum_
        mov     rcx, qword ptr [rdi]
        mov     rdx, qword ptr [rdi + 8]
        cmp     rcx, rdx
        je      .LBB0_4
.LBB0_1:                                # =>This Inner Loop Header: Depth=
        test    byte ptr [rcx], 1
        je      .LBB0_4
        add     rcx, 4
        cmp     rcx, rdx
        jne     .LBB0_1
        jmp     .LBB0_3
.LBB0_4:
        cmp     rcx, rdx
        je      .LBB0_3
        mov     esi, dword ptr [rcx]
        xor     eax, eax
.LBB0_6:                                # =>This Loop Header: Depth=1
        mov     edi, eax
        imul    esi, esi
        mov     eax, esi
        add     eax, edi
.LBB0_7:                                #   Parent Loop BB0_6 Depth=1
        add     rcx, 4
        cmp     rcx, rdx
        je      .LBB0_10
        mov     esi, dword ptr [rcx]
        test    sil, 1
        jne     .LBB0_7
        cmp     rcx, rdx
        jne     .LBB0_6
.LBB0_10:
        ret
.LBB0_3:
        xor     eax, eax
        ret
```

C++ source #1

```cpp

#include <cstddef>
#include <stdint.h>
#include <vector>

#ifdef USE_FLUX

#include "https://raw.githubusercontent.com/tcbrindle/flux/main/single_i

int sum_sq_even_flux(std::vector<int> const& vec)
{
    return flux::ref(vec)
            .filter(flux::pred::even)
            .map([](int i) { return i * i; })
            .sum();
}

#else

#include <algorithm>
#include <numeric>
#include <ranges>

int sum_sq_even_ranges(std::vector<int> const& vec)
{
    auto view = vec
        | std::views::filter([](int i) { return i % 2 == 0; })
        | std::views::transform([](int i) { return i * i; });
    return std::accumulate(view.begin(), view.end(), 0);
}

#endif

```

x86-64 clang 16.0.0 (Editor #1)

x86-64 clang 16.0.0 -std=c++20 -O3 -DNDEBUG -m

```
        vpxor   xmm3, xmm3, xmm3
        vpxor   xmm4, xmm4, xmm4
LBB0_5:                                 # =>This Inner Loop Header: Depth=1
        vmovdqu ymm5, ymmword ptr [r8 + 4*rax]
        vmovdqu ymm6, ymmword ptr [r8 + 4*rax + 32]
        vmovdqu ymm7, ymmword ptr [r8 + 4*rax + 64]
        vmovdqu ymm8, ymmword ptr [r8 + 4*rax + 96]
        vptestnmd       k1, ymm5, ymm1
        vptestnmd       k2, ymm6, ymm1
        vptestnmd       k3, ymm7, ymm1
        vptestnmd       k4, ymm8, ymm1
        vpmulld ymm5, ymm5, ymm5
        vpmulld ymm6, ymm6, ymm6
        vpmulld ymm7, ymm7, ymm7
        vpmulld ymm8, ymm8, ymm8
        vpaddd  ymm0 {k1}, ymm0, ymm5
        vpaddd  ymm2 {k2}, ymm2, ymm6
        vpaddd  ymm3 {k3}, ymm3, ymm7
        vpaddd  ymm4 {k4}, ymm4, ymm8
        add     rax, 32
        cmp     r9, rax
        jne     .LBB0_5
        vpaddd  ymm0, ymm2, ymm0
        vpaddd  ymm1, ymm4, ymm3
        vpaddd  ymm0, ymm1, ymm0
        vextracti128    xmm1, ymm0, 1
        vpaddd  xmm0, xmm0, xmm1
        vpshufd xmm1, xmm0, 238                 # xmm1 = xmm0[2,3,2,3]
        vpaddd  xmm0, xmm0, xmm1
        vpshufd xmm1, xmm0, 85                  # xmm1 = xmm0[1,1,1,1]
        vpaddd  xmm0, xmm0, xmm1
        vmovd   eax, xmm0
        cmp     rdi, r9
        je      .LBB0_8
LBB0_7:                                 # =>This Inner Loop Header: Depth=1
        mov     edi, dword ptr [rsi]
        mov     r8d, edi
        imul    r8d, edi
        test    dil, 1
        cmovne  r8d, edx
        add     eax, r8d
        add     rsi, 4
        cmp     rsi, rcx
        jne     .LBB0_7
```

x86-64 clang 16.0.0 (Editor #1)

x86-64 clang 16.0.0 -std=c++20 -O3 -DNDEBUG -R

```
sum_sq_even_ranges(std::vector<int, std::allocator<int> > const&): # @sum_
        mov     rcx, qword ptr [rdi]
        mov     rdx, qword ptr [rdi + 8]
        cmp     rcx, rdx
        je      .LBB0_4
.LBB0_1:                                # =>This Inner Loop Header: Depth=
        test    byte ptr [rcx], 1
        je      .LBB0_4
        add     rcx, 4
        cmp     rcx, rdx
        jne     .LBB0_1
        jmp     .LBB0_3
.LBB0_4:
        cmp     rcx, rdx
        je      .LBB0_3
        mov     esi, dword ptr [rcx]
        xor     eax, eax
.LBB0_6:                                # =>This Loop Header: Depth=1
        mov     edi, eax
        imul    esi, esi
        mov     eax, esi
        add     eax, edi
.LBB0_7:                                #   Parent Loop BB0_6 Depth=1
        add     rcx, 4
        cmp     rcx, rdx
        je      .LBB0_10
        mov     esi, dword ptr [rcx]
        test    sil, 1
        jne     .LBB0_7
        cmp     rcx, rdx
        jne     .LBB0_6
.LBB0_10:
        ret
.LBB0_3:
        xor     eax, eax
        ret
```

AVOIDING BOUNDS CHECKING

- Implementations of `for_each_while()` typically don't need internal bounds-checking
- In most other cases, the compiler an optimise away bounds checks
- But what about corner cases?

AVOIDING BOUNDS CHECKING

- If bounds checking does lead to unacceptable overhead, Flux provides the `read_at_unchecked()` function which performs element access without additional checks
- This can be used ***selectively***, only where truly needed
  - Always measure!
- Think of it as the equivalent of `unsafe` blocks in other languages

CONCLUSION

- Flux is a modern C++20 library for *collection-orientated programming*
- It can be used everywhere C++20 ranges can be used, and offers
  - Much improved safety by default
  - An easy-to-use API for both producers and consumers
  - Compatibility with existing STL code
  - In many cases, better performance than equivalent ranges code

THANK YOU!

- Flux: github.com/tcbrindle/flux
- Docs: tristanbrindle.com/flux
- Twitter: @tristanbrindle

EXPLICIT REFERENCING IN FLUX

- C++20 range adaptors (“views”) take an implicit reference when passed a non-`view`, lvalue range
- This can lead to surprising behaviour and “dangling views” in some cases

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

auto evens = filter_evens({1, 2, 3, 4, 5}); // uh oh
```

EXPLICIT REFERENCING IN FLUX

- In Flux, potentially “long-lived” references must be explicitly passed to sequence adaptors
- Makes dangling much more obvious in code

```
auto filter_evens(std::vector<int> const& vec)
{
    return flux::filter(vec, [](int i) { return i % 2 == 0; }); // Compile error!
}
```

EXPLICIT REFERENCING IN FLUX

- In Flux, potentially “long-lived” references must be explicitly passed to sequence adaptors
- Makes dangling much more obvious in code

```
auto filter_evens(std::vector<int> const& vec)
{
    // Now compiles, but more obviously problematic
    return flux::filter(flux::ref(vec), [](int i) { return i % 2 == 0; });
}
```