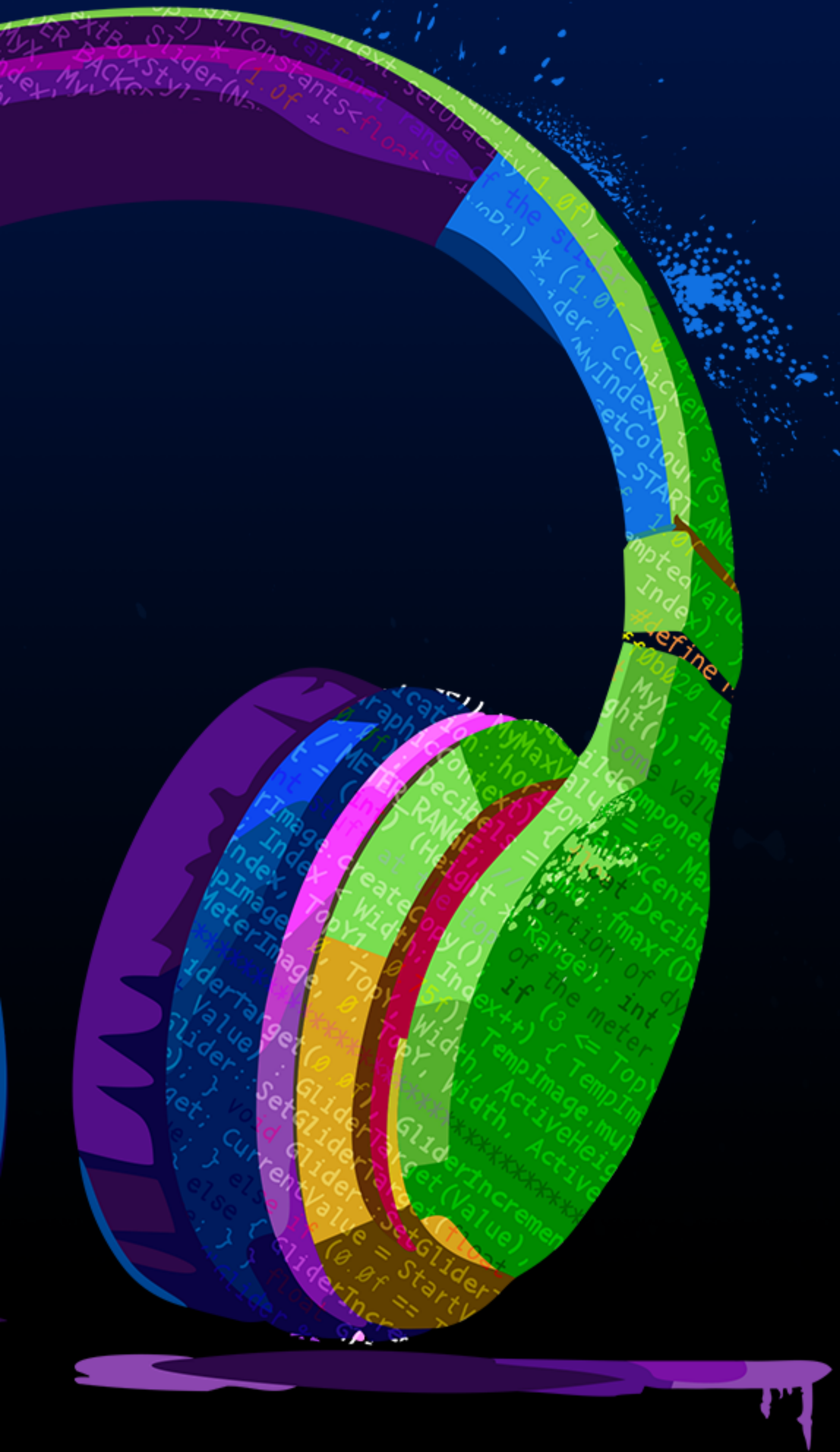


ADC24 Bristol

CAN AUDIO PROGRAMMING BE SAFE?

DAVE ROWLAND

PRISM SOUND

tracktion

POWERED BY
tracktion
ENGINE

1800 mm
2025 mm
EcoTherm
gapotape
easy-trim JUTA Breather Membranes
MASTER
BS5534

60

60

The Plan:

- Start by showing you a load of talks you should watch instead of this one
- Compare how safe different languages are
- See how C++ *might be* becoming safer…
- Tie it together at the end (hopefully)
- There will be (many) tangents

The Goal:

Start thinking about safety

C++ and Safety

Timur Doumler

@timur_audio

CppOnSea
29 June 2023

An artist's conception of a supernova explosion.
Credit: NASA's Goddard Space Flight Center / ESA / Hubble / L. Calcada

Timur Doumler

Who cares about safety?

Whether they're right,
that's a different question

Conclusion:

Today, C++ developers generally do not perceive undefined behaviour as a business-critical problem.

Conceptual

- JF Bastian
  Safety and Security: The Future of C++ - CppNow 2023

- Sean Parent
  All the Safeties: Safety in C++ - CppNow 2023

- Dave Abrams
  Value Semantics: Safety, Independence, Projection, & Future of Programming - CppCon 2022

- Timur Doumler
  C++ and Safety - C++ on Sea 2023

Practical

- Tristan Brindle
  Practical Tips for Safer C++ - C++ on Sea 2024

- Herb Sutter
  Any talk in the last 5ish years

- Louis Dionne
  Security in C++ - Hardening Techniques From the Trenches - C++Now 2024

- Gabor Horvath
  Lifetime Safety in C++: Past, Present and Future - CppCon 2023

Governments

- Nov. 10, 2022 - NSA Releases Guidance on How to Protect Against Software Memory Safety Issues[@nsa-guidance]
- Sep. 20, 2023 - The Urgent Need for Memory Safety in Software Products[@cisa-urgent]
- Dec. 6, 2023 - CISA Releases Joint Guide for Software Manufacturers: The Case for Memory Safe Roadmaps[@cisa-roadmaps]
- Feb. 26, 2024 - Future Software Should Be Memory Safe[@white-house]
- May 7, 2024 - National Cybersecurity Strategy Implementation Plan[@ncsi-plan]

It is possible that in the near future companies will have to adhere to some **programming safety regulation**

Safety and security

- **Safety:**
  - Focuses on system reliability and proper functioning
  - *Examples: preventing system crashes in medical devices, ensuring proper operation of industrial control systems*
- **Security:**
  - Aims to protect against intentional malicious actions
  - *Focuses on protecting data, systems, and networks from unauthorised access or attacks*
  - *Concerned with threats like data breaches, hacking, or malware*

Safety and security

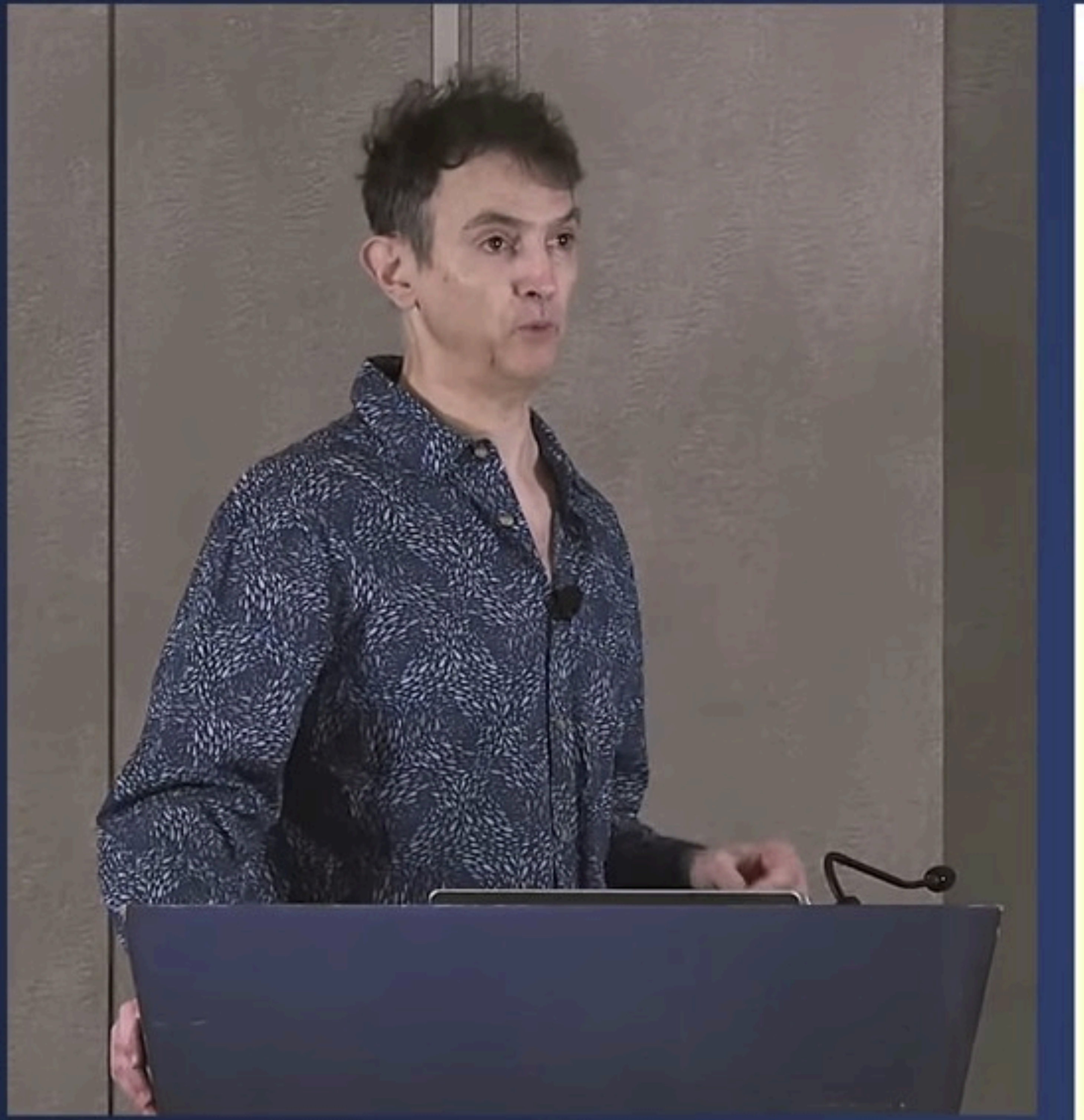
- **A *safe* program:**
  - A correctly functioning program that doesn’t crash
- **A *secure* program:**
  - A program that a malicious attacker can’t exploit

> *"No functional safety without security;*
> *no security without type, resource,*
> *& memory safety."*
>
> – JF Bastien

Types of Language Safety

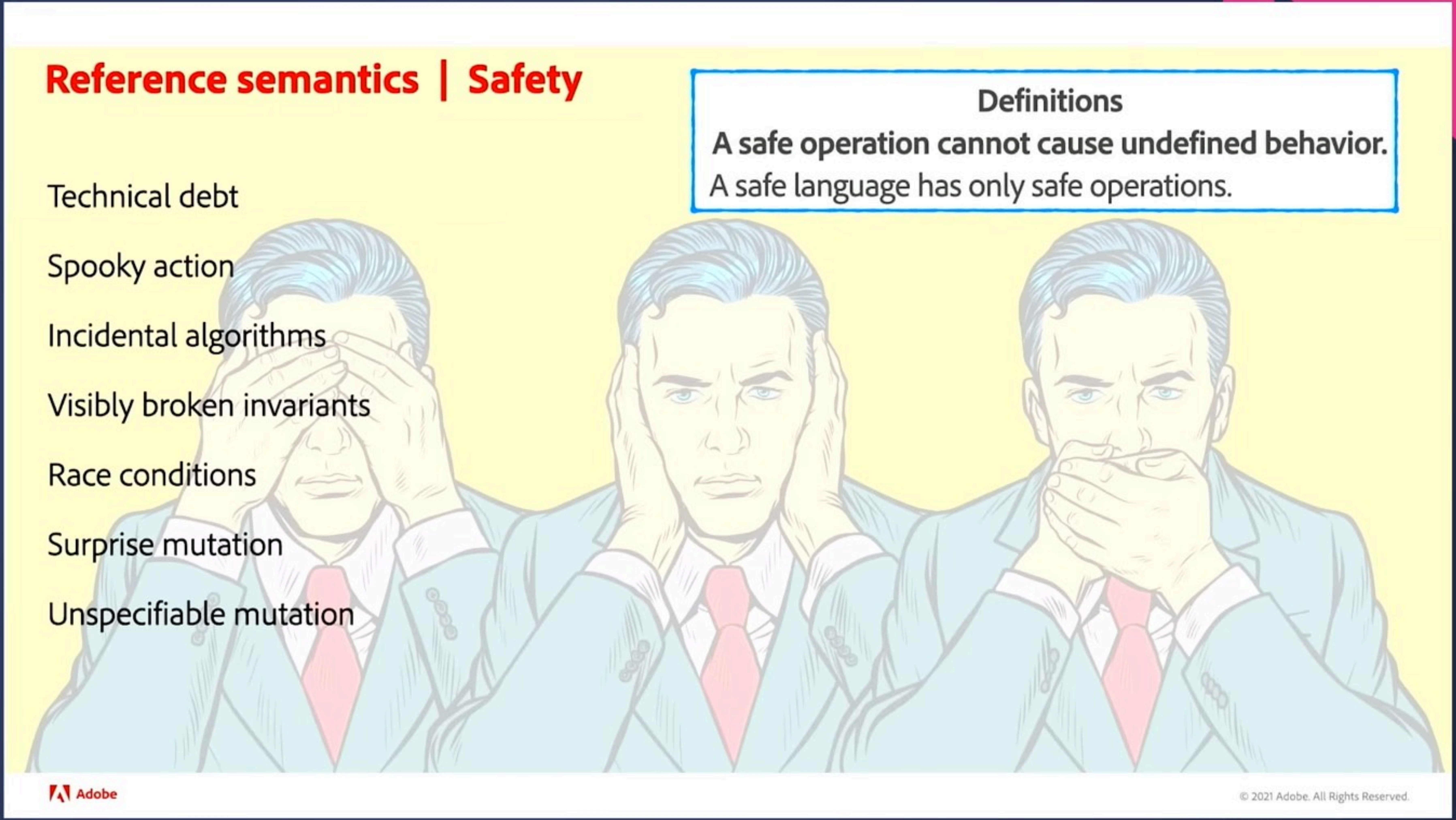
- Memory:
  - Type
  - Bounds
  - Lifetime
  - Initialisation
- Arithmetic
- Thread
- Definition safety

Cppcon 2022
The C++ Conference
September 12th-16th



Dave Abrahams

Values: Safety, Regularity, Independence, and the Future of Programming



Reference semantics | Safety

Definitions

A safe operation cannot cause undefined behavior.

A safe language has only safe operations.

Technical debt

Spooky action

Incidental algorithms

Visibly broken invariants

Race conditions

Surprise mutation

Unspecifiable mutation

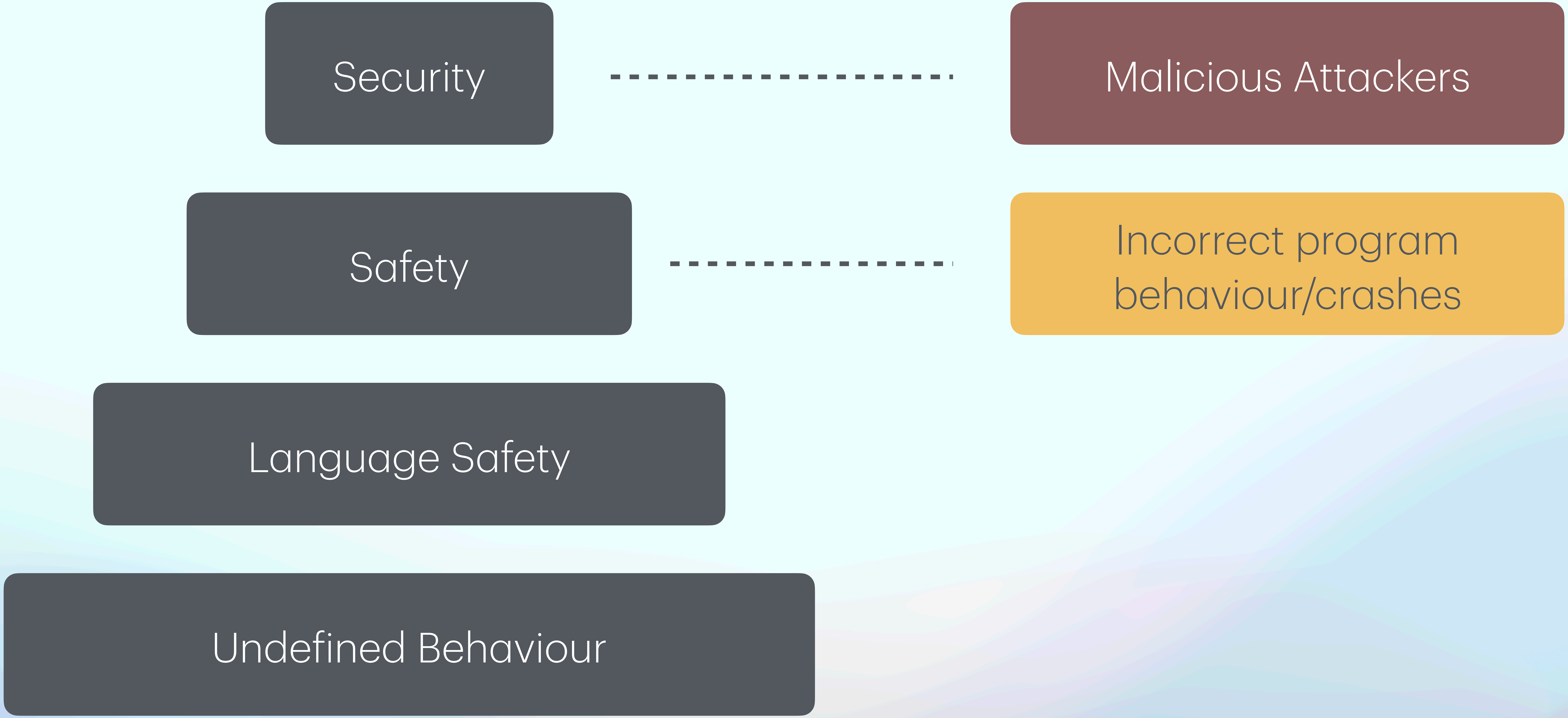
Adobe

Language Safety defined as UB (in C++)

(Paraphrasing Timur)

- ***A programming language is safe it it doesn’t let you express undefended behaviour***

- Why not make all UB ill-formed?
  - Performance
  - Backwards compatibility
  - Complexity
  - Expressivity
  - Portability



Security

Malicious Attackers

Safety

Incorrect program behaviour/crashes

Language Safety

Undefined Behaviour

Undefined Behaviour

Blank slate or existing C++

Cmajor/JS

Rust

Swift

Carbon

“C++"

safecpptool

Circle

cpp2

iso c++

Blank slate or existing C++

Cmajor/JS

Rust

Swift

“C++”

safecpptool

Circle

cpp2

iso c++

C++: safecpptool

- *“scpptool is a command line tool to help enforce a memory and data race safe subset of C++”*
- *“Designed to work with the SaferCPlusPlus library”*
- *“Necessarily comes at some (modest) expense of either flexibility or performance”*
- *“Impose only the minimum restrictions and departures from traditional C++ necessary to achieve practical performant memory safety”*
- Safer replacement std library
- Linter tool to check only “safe” code patterns/libraries are used

Blank slate or existing C++

Cmajor/JS

Rust

Swift

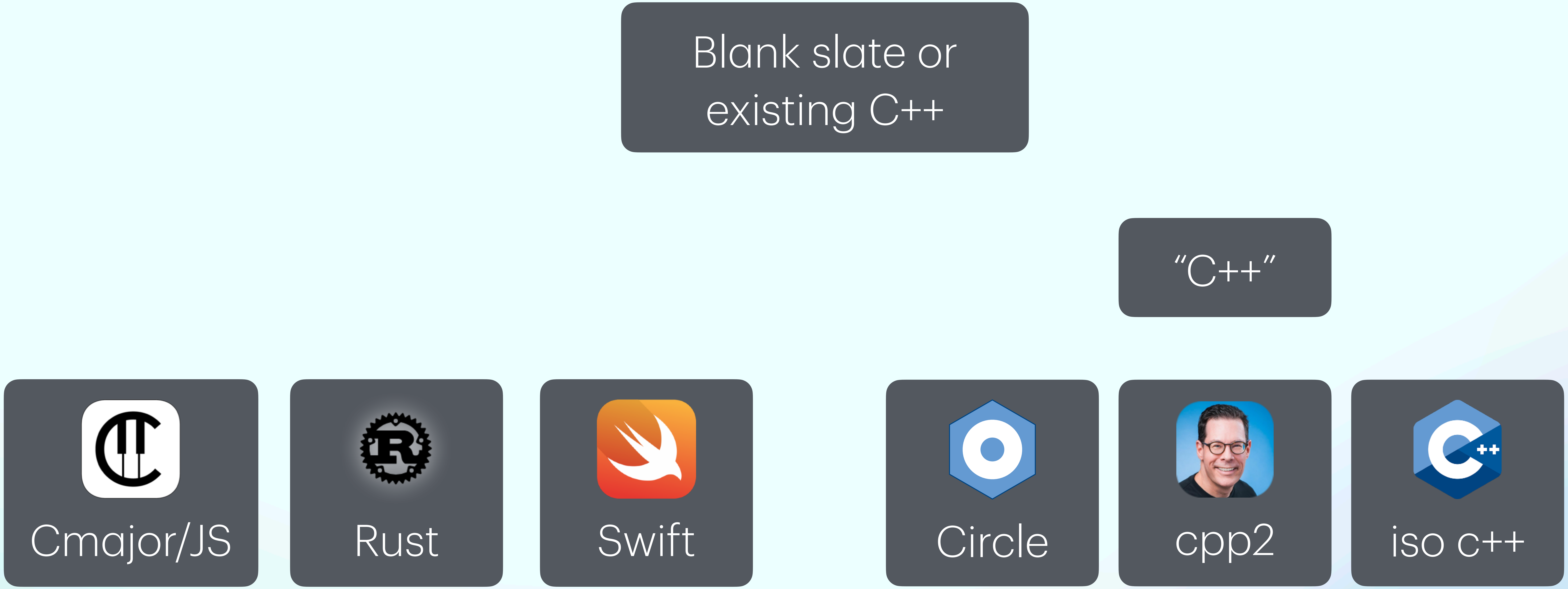
"C++"

safecpptool

Circle

cpp2

iso c++



Blank slate or existing C++

“C++”

Cmajor/JS

Rust

Swift

Circle

cpp2

iso c++

Key:

- **Type**
- **Bounds**
- **Lifetime**
- **Initialisation**
- **Arithmetic**
- **Thread**
- **Definition**

🟩 static/compile-time/enforced

🟦 dynamic/run-time/checked

🟨 separate-tool/external-application

🟥 non-existant

	Cmajor/JS	Swift	C++23	Circle	cpp2	iso c++
Type	Static/dynamic type system	Static type system	Static type system reinterpret_cast	Static type system	Static type system	Profile: Type*
Bounds	Enforced/checked	Checked	Asan	Checked	Checked	Profile: Ranges, Algorithms & Pointers
Lifetime	Static/ ref-counted	Value semantics & Ref-counted	Partially enforced/ Asan	Enforced borrow checker	Partially enforced/ checked	Profile: RAII
Initialisation	Default initialised	Enforced	MSan/Asan	Enforced	Enforced	Profile: Initialisation
Arithmetic	ID/defined	Trap/explicit behaviour	UBsan	Checked/ defined	Checked	Profile: Arithmetic
Thread	Single* threaded	Enforced actors & sendable	Tsan	Enforced sync/send & BC	Tsan	Tsan
Definition	Single file/modules	Modules	Modules	Modules	Modules	Modules

	Cmajor/JS	Swift	C++23	Circle	cpp2	iso c++
Type	Static/dynamic type system	Static type system	Static type system reinterpret_cast	Static type system	Static type system	Profile: Type*
Bounds	Enforced/checked	Checked	Asan	Checked	Checked	Profile: Ranges, Algorithms & Pointers
Lifetime	Static/ ref-counted	Value semantics & Ref-counted	Partially enforced/ Asan	Enforced borrow checker	Partially enforced/ checked	Profile: RAII
Initialisation	Default initialised	Enforced	MSan/Asan	Enforced	Enforced	Profile: Initialisation
Arithmetic	ID/defined	Trap/explicit behaviour	UBsan	Checked/ defined	Checked	Profile: Arithmetic
Thread	Single* threaded	Enforced actors & sendable	Tsan	Enforced sync/send & BC	Tsan	Tsan
Definition	Single file/modules	Modules	Modules	Modules	Modules	Modules

Cmajor

- **Pros:**
  - Basically statically safe/enforced
  - Arithmetic implementation defined (to maximise performance)
    - Divide by 0, signed integer overflow
  - Fast
- **Cons:**
  - DSL - need additional app/plugin logic
  - Need to add Cmajor runtime
    - Aided by plugin export tools

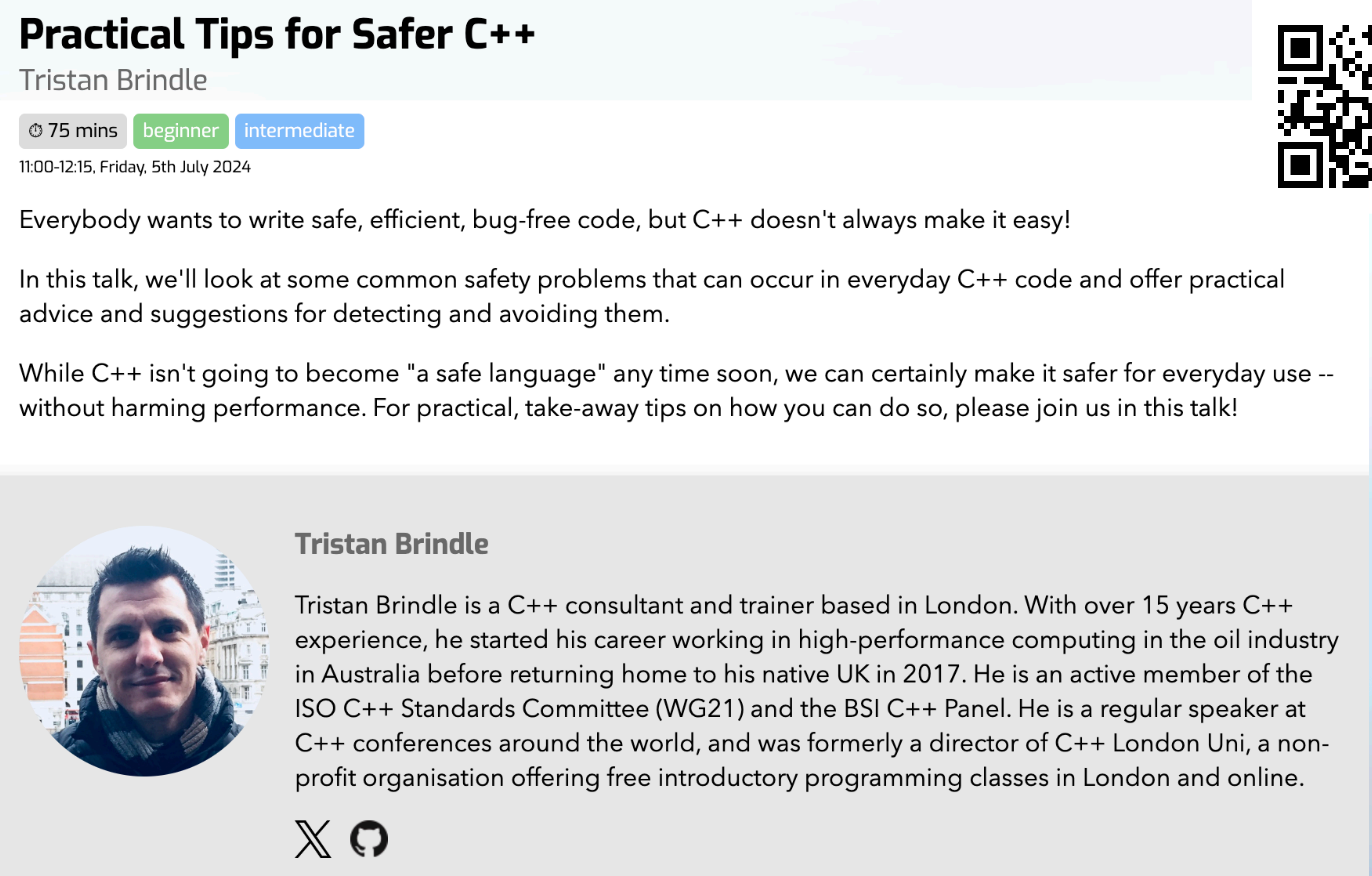
Type	Static
Bounds	Enforced
Lifetime	Static
Initialisation	Default initialised
Arithmetic	Implementation defined
Thread	Single* threaded
Definition	Single file

C++23

- **Pros:**
  - Lot's of tools to catch common errors
  - Compilers can help sometimes*
- **Cons:**
  - Tools ***not default***
  - Not available on all platforms
  - Mutually exclusive
  - False positives/negatives

Type	Static type system reinterpret_cast
Bounds	Asan
Lifetime	Asan
Initialisation	MSan/Asan
Arithmetic	UBsan
Thread	Tsan
Definition	Modules

Can C++23 do better?



Practical Tips for Safer C++

Tristan Brindle

⏱ 75 mins | beginner | intermediate

11:00-12:15, Friday, 5th July 2024

Everybody wants to write safe, efficient, bug-free code, but C++ doesn't always make it easy!

In this talk, we'll look at some common safety problems that can occur in everyday C++ code and offer practical advice and suggestions for detecting and avoiding them.

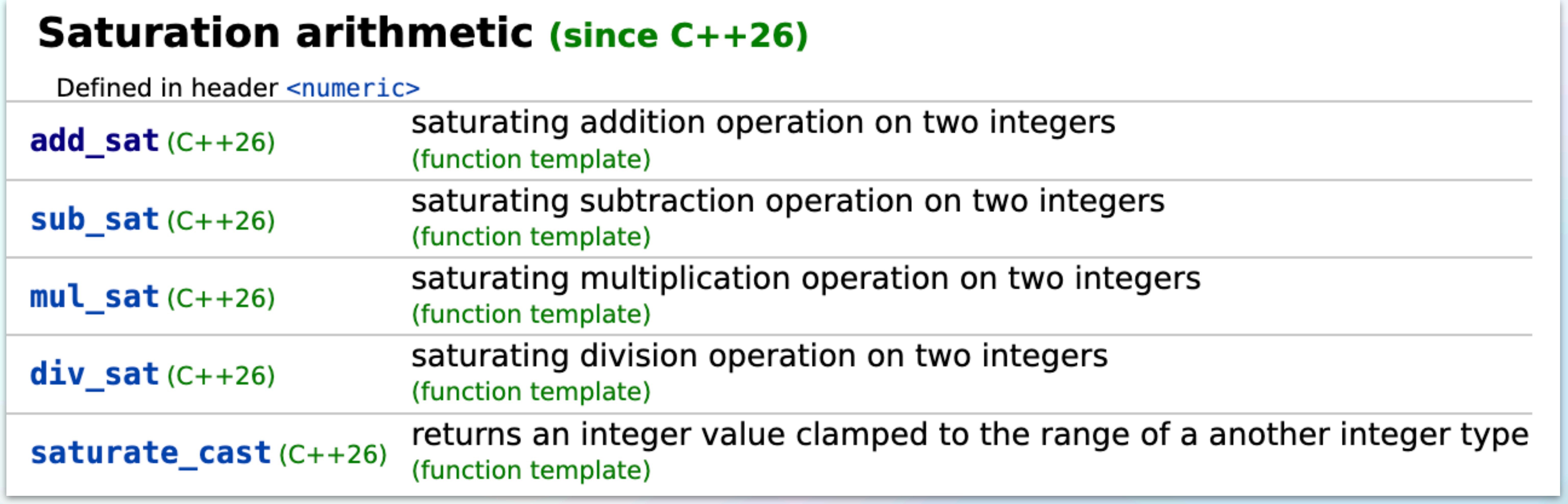
While C++ isn't going to become "a safe language" any time soon, we can certainly make it safer for everyday use -- without harming performance. For practical, take-away tips on how you can do so, please join us in this talk!

Tristan Brindle

Tristan Brindle is a C++ consultant and trainer based in London. With over 15 years C++ experience, he started his career working in high-performance computing in the oil industry in Australia before returning home to his native UK in 2017. He is an active member of the ISO C++ Standards Committee (WG21) and the BSI C++ Panel. He is a regular speaker at C++ conferences around the world, and was formerly a director of C++ London Uni, a non-profit organisation offering free introductory programming classes in London and online.

C++23/26*

- **Bounds:**
  - Use the ***flux*** library (index based ranges) or `std::ranges`
  - Use hardened std library (`_LIBCPP_HARDENING_MODE_DEBUG/FAST=1`)
- **Lifetime:**
  - Static analyser
- **Initialisation:**
  - Static analyser
- **Arithmetic:**
  - Saturating numeric operations (C++26)
  - Use ***`-ftrapv`*** to generate traps for signed integer overflow
- **Thread:** Tsan



Saturation arithmetic (since C++26)

Defined in header `<numeric>`

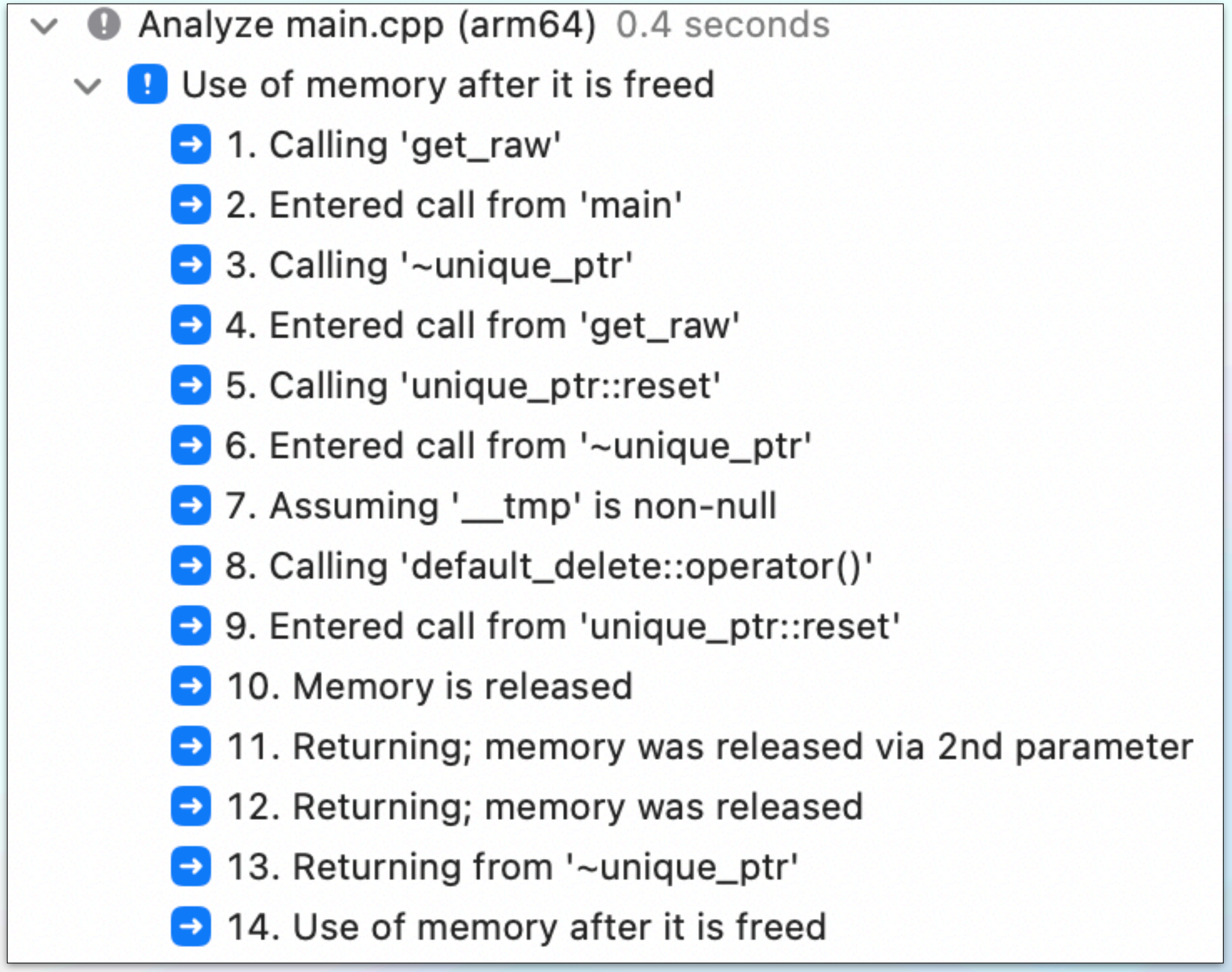
`add_sat` (C++26)	saturating addition operation on two integers (function template)
`sub_sat` (C++26)	saturating subtraction operation on two integers (function template)
`mul_sat` (C++26)	saturating multiplication operation on two integers (function template)
`div_sat` (C++26)	saturating division operation on two integers (function template)
`saturate_cast` (C++26)	returns an integer value clamped to the range of a another integer type (function template)

Bounds/Lifetime: Flux - Sequence Based Programming

- Cursor based design (similar to Rust iterators)
- Can't dangle
- Can't be invalidated
- Bounds checked

Lifetime: Static Analysis

```
int* get_raw() {
    auto ptr = std::make_unique<int> (42);
    return ptr.get();
}

int main() {
    auto raw = get_raw();
    std::cout << "Hello " << *raw;
}
```



Analyze main.cpp (arm64) 0.4 seconds

Use of memory after it is freed

1. Calling 'get_raw'
2. Entered call from 'main'
3. Calling '~unique_ptr'
4. Entered call from 'get_raw'
5. Calling 'unique_ptr::reset'
6. Entered call from '~unique_ptr'
7. Assuming '__tmp' is non-null
8. Calling 'default_delete::operator()'
9. Entered call from 'unique_ptr::reset'
10. Memory is released
11. Returning; memory was released via 2nd parameter
12. Returning; memory was released
13. Returning from '~unique_ptr'
14. Use of memory after it is freed

Tests

- Requires good test coverage
- Tests tend to cover intended use cases
  - Or reported (and already fixed) issues (regressions)

C++ Static Analysers

- **Clang**
- **CLion**
- **Sonar**
- **PVStudio**
- **Cppcheck**

- **clang-tidy**
  - **Core guidelines checks**
- **MSVC Code Analysis**
  - **CppCoreCheck**

C++ Core Guidelines Checks

- cppcoreguidelines-pro-bounds-array-to-pointer-decay
- cppcoreguidelines-pro-bounds-constant-array-index
- cppcoreguidelines-pro-bounds-pointer-arithmetic
- cppcoreguidelines-pro-type-const-cast
- cppcoreguidelines-pro-type-cstyle-cast
- cppcoreguidelines-pro-type-member-init
- cppcoreguidelines-pro-type-reinterpret-cast
- cppcoreguidelines-pro-type-static-cast-downcast
- cppcoreguidelines-pro-type-union-access
- cppcoreguidelines-pro-type-vararg

`cppcoreguidelines-pro-bounds-constant-array-index`

- This check flags all array subscript expressions on static arrays and `std::array`s that either do not have a constant integer expression index or are out of bounds (for `std::array`). For out-of-bounds checking of static arrays, see the `-Warray-bounds` Clang diagnostic.
    - **Bounds.2**: Only index into arrays using constant expressions: Pass pointers to single objects (only) and Keep pointer arithmetic simple.
- Optionally, this check can generate fixes using **`gsl::at`** for indexing.
    - If you provide clang-tidy with a path to `gsl/gsl.h` (non-standard)

```
clang-tidy -checks=‘cppcoreguidelines-pro-bounds-constant-array-index' file.cpp

int get_index();

int main()
{
    std::array arr = { 0, 1, 2, 3, 4, 5 };
    auto v = arr[get_index()];
}
```

```
<source>:13:14: warning: do not use array subscript when the index is not an integer
constant expression [cppcoreguidelines-pro-bounds-constant-array-index]
   13 |     auto v = arr[get_index()];
      |              ^
```

```cpp
int get_index();

int main()
{
    std::array arr = { 0, 1, 2, 3, 4, 5 };
    auto v = arr.at (get_index());        // Throws std::out_of_range
    auto v2 = gsl::at (arr, get_index()); // Contract violation (std::terminate)
}
```

C++ Core Guidelines

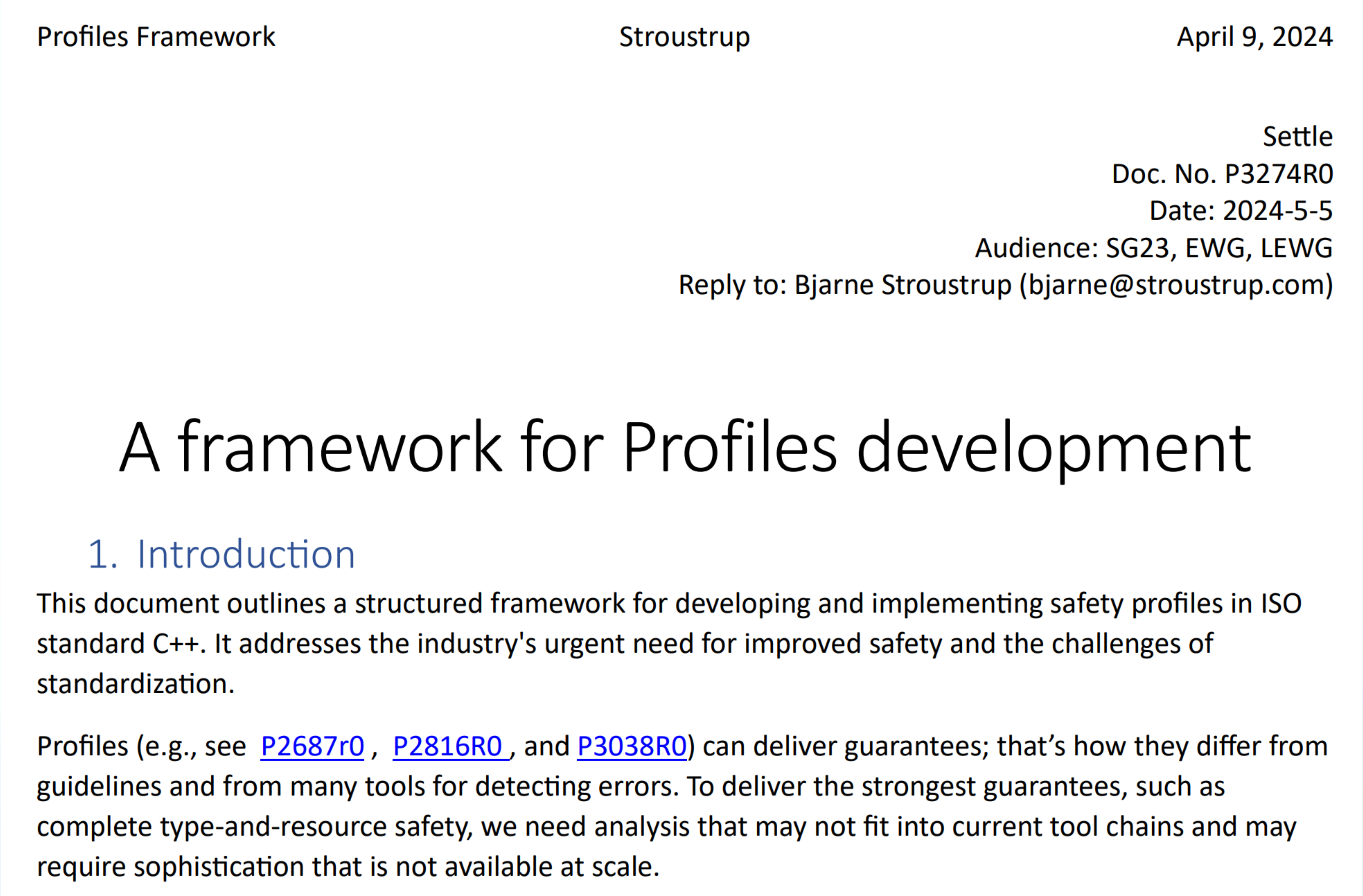
- **Pro.bounds: Bounds safety profile**
  - **Bounds.4**: Don’t use standard-library functions and types that are not bounds-checked: Use the standard library in a type-safe manner.
- **SL.con.3: Avoid bounds errors**
- ***Reason*** Read or write beyond an allocated range of elements typically leads to bad errors, wrong results, crashes, and security violations.

Has anyone read the C++ Core Guidelines?

All ~650 pages of them?

Warning: Speculation

C++29? P3274: Profiles



Profiles Framework Stroustrup April 9, 2024

Settle
Doc. No. P3274R0
Date: 2024-5-5
Audience: SG23, EWG, LEWG
Reply to: Bjarne Stroustrup (bjarne@stroustrup.com)

A framework for Profiles development

1. Introduction

This document outlines a structured framework for developing and implementing safety profiles in ISO standard C++. It addresses the industry's urgent need for improved safety and the challenges of standardization.

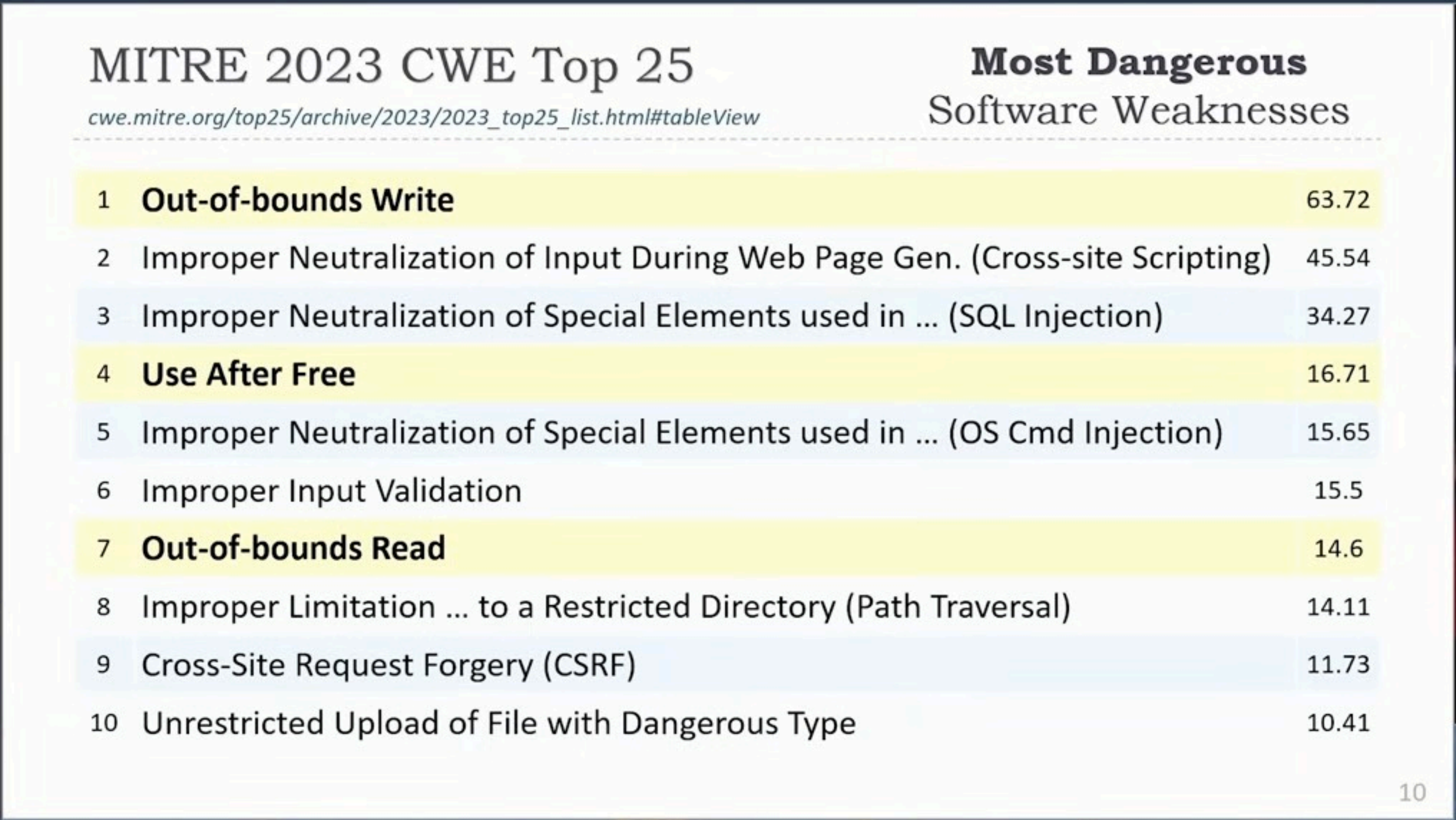
Profiles (e.g., see P2687r0 , P2816R0 , and P3038R0) can deliver guarantees; that’s how they differ from guidelines and from many tools for detecting errors. To deliver the strongest guarantees, such as complete type-and-resource safety, we need analysis that may not fit into current tool chains and may require sophistication that is not available at scale.

C++ Profiles

- Type
- Arithmetic
- Concurrency
- Ranges
- Pointers
- Algorithms
- Initialization
- Casting
- Invalidation
- RAII
- Union

```cpp
[[profiles::enable(ranges)]]

int get_index();

int main()
{
    std::array arr = { 0, 1, 2, 3, 4, 5 };
    auto v = arr[get_index()];
}
```

```
[[profiles::enable(ranges)]]

int get_index();

int main()
{
    std::array arr = { 0, 1, 2, 3, 4, 5 };
    [[profiles::suppress(ranges)]]
    auto v = arr[get_index()];
}
```



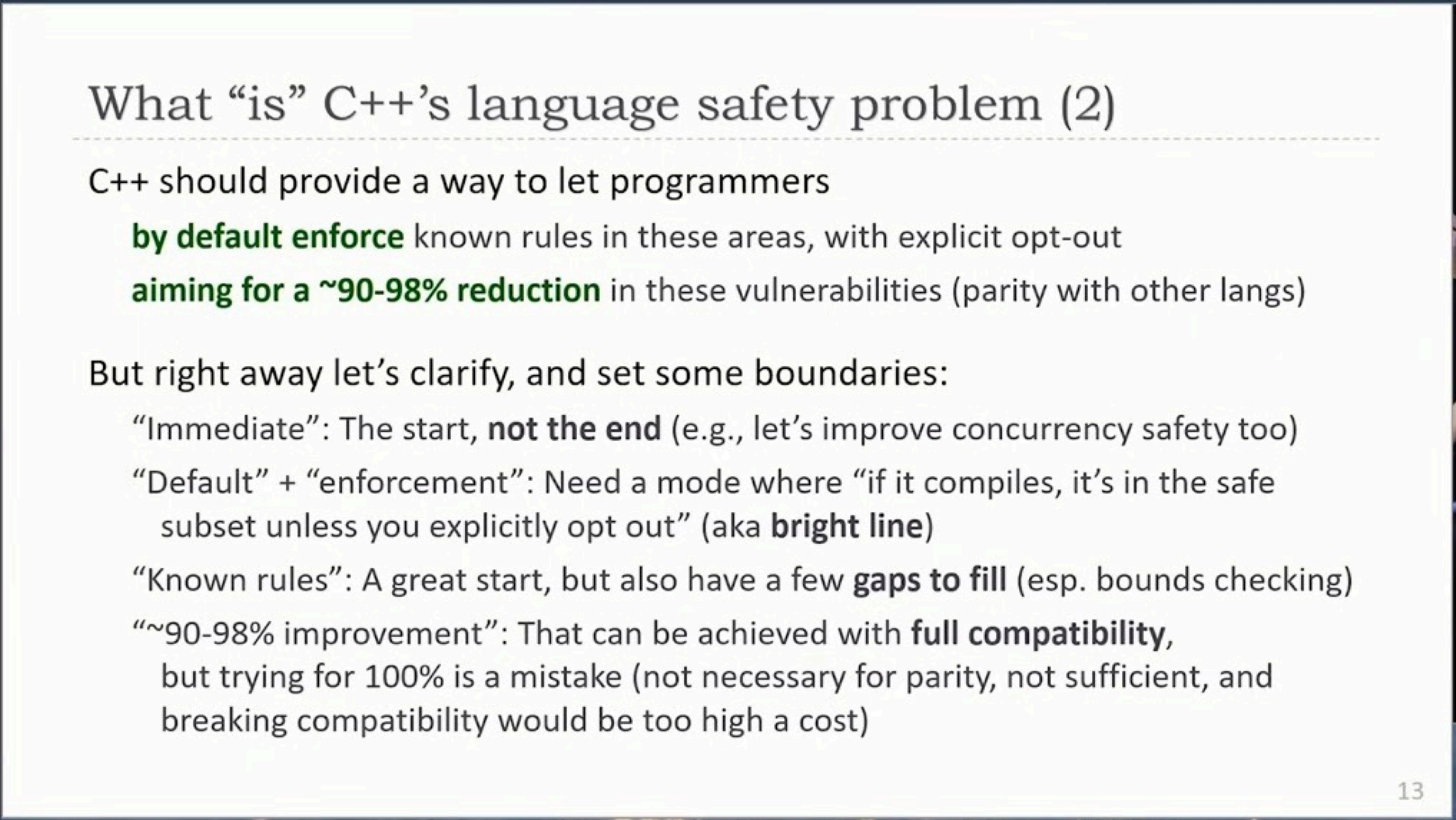
MITRE 2023 CWE Top 25

cwe.mitre.org/top25/archive/2023/2023_top25_list.html#tableView

Most Dangerous
Software Weaknesses

1	**Out-of-bounds Write**	63.72
2	Improper Neutralization of Input During Web Page Gen. (Cross-site Scripting)	45.54
3	Improper Neutralization of Special Elements used in … (SQL Injection)	34.27
4	**Use After Free**	16.71
5	Improper Neutralization of Special Elements used in … (OS Cmd Injection)	15.65
6	Improper Input Validation	15.5
7	**Out-of-bounds Read**	14.6
8	Improper Limitation … to a Restricted Directory (Path Traversal)	14.11
9	Cross-Site Request Forgery (CSRF)	11.73
10	Unrestricted Upload of File with Dangerous Type	10.41

Herb Sutter



What "is" C++'s language safety problem (2)

C++ should provide a way to let programmers

- **by default enforce** known rules in these areas, with explicit opt-out
- **aiming for a ~90-98% reduction** in these vulnerabilities (parity with other langs)

But right away let's clarify, and set some boundaries:

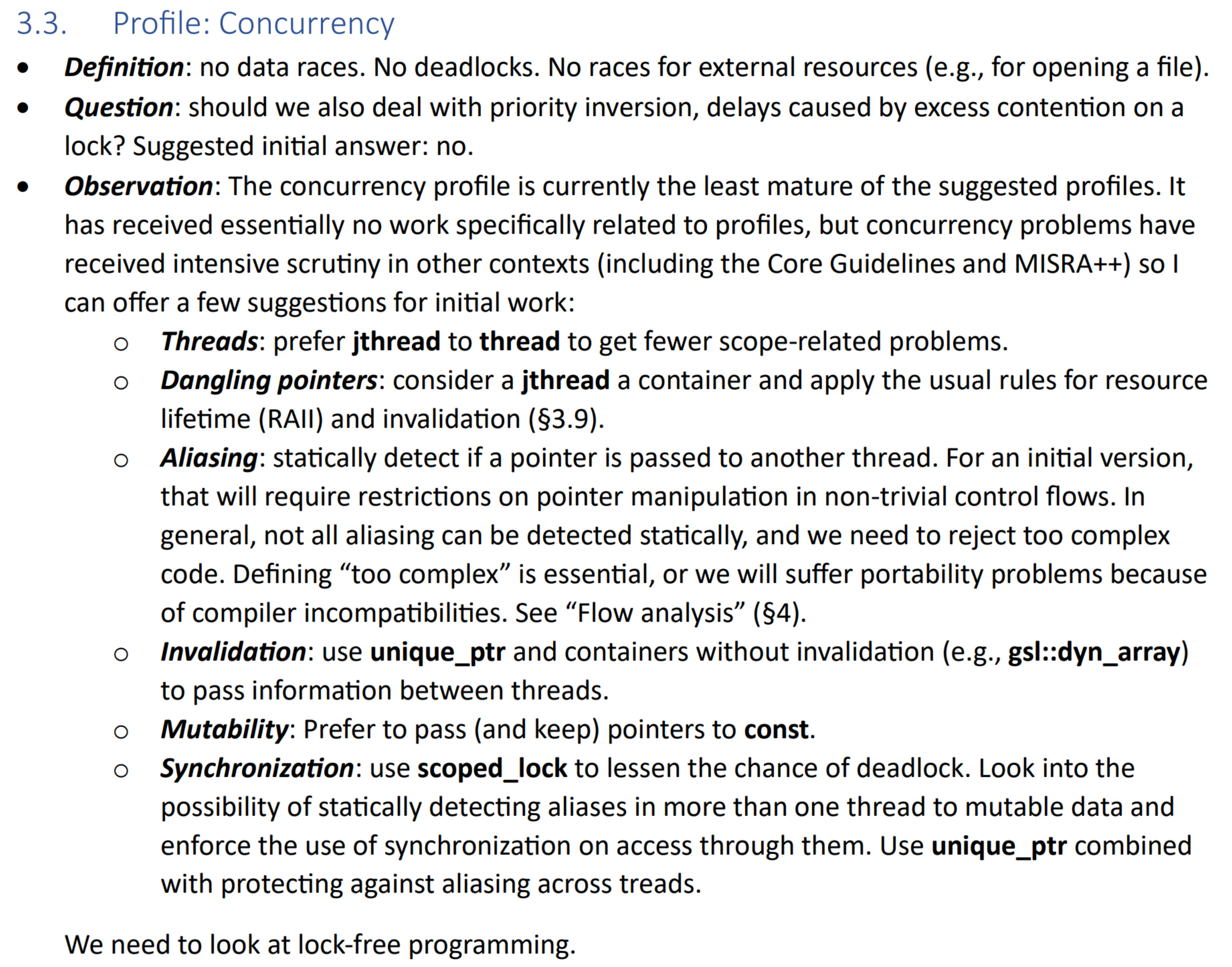
- "Immediate": The start, **not the end** (e.g., let's improve concurrency safety too)
- "Default" + "enforcement": Need a mode where "if it compiles, it's in the safe subset unless you explicitly opt out" (aka **bright line**)
- "Known rules": A great start, but also have a few **gaps to fill** (esp. bounds checking)
- "~90-98% improvement": That can be achieved with **full compatibility**, but trying for 100% is a mistake (not necessary for parity, not sufficient, and breaking compatibility would be too high a cost)

C++ Profiles

- **Pros**
  - Standard, no extra tools
  - Progressive adoption
- **Cons**
  - A mess
  - Difficulties with flow analysis
  - Mixture of compile and run-time violations
  - Initial offering like to be “lite” version

Threads?

Threads?



3.3. Profile: Concurrency

- ***Definition***: no data races. No deadlocks. No races for external resources (e.g., for opening a file).
- ***Question***: should we also deal with priority inversion, delays caused by excess contention on a lock? Suggested initial answer: no.
- ***Observation***: The concurrency profile is currently the least mature of the suggested profiles. It has received essentially no work specifically related to profiles, but concurrency problems have received intensive scrutiny in other contexts (including the Core Guidelines and MISRA++) so I can offer a few suggestions for initial work:
  - ○ ***Threads***: prefer **jthread** to **thread** to get fewer scope-related problems.
  - ○ ***Dangling pointers***: consider a **jthread** a container and apply the usual rules for resource lifetime (RAII) and invalidation (§3.9).
  - ○ ***Aliasing***: statically detect if a pointer is passed to another thread. For an initial version, that will require restrictions on pointer manipulation in non-trivial control flows. In general, not all aliasing can be detected statically, and we need to reject too complex code. Defining "too complex" is essential, or we will suffer portability problems because of compiler incompatibilities. See "Flow analysis" (§4).
  - ○ ***Invalidation***: use **unique_ptr** and containers without invalidation (e.g., **gsl::dyn_array**) to pass information between threads.
  - ○ ***Mutability***: Prefer to pass (and keep) pointers to **const**.
  - ○ ***Synchronization***: use **scoped_lock** to lessen the chance of deadlock. Look into the possibility of statically detecting aliases in more than one thread to mutable data and enforce the use of synchronization on access through them. Use **unique_ptr** combined with protecting against aliasing across treads.

We need to look at lock-free programming.

cpp2

- **Pros:**
  - Safer type, bounds, lifetime, initialisation and arithmetic
  - Incrementally opt-in
  - Transpiles to C++ (still have the C++ code)
  - Perfect interop*
- **Cons:**
  - Different language to learn/teach
  - Still in infancy
  - Lifetime safety not great
  - No thread safety

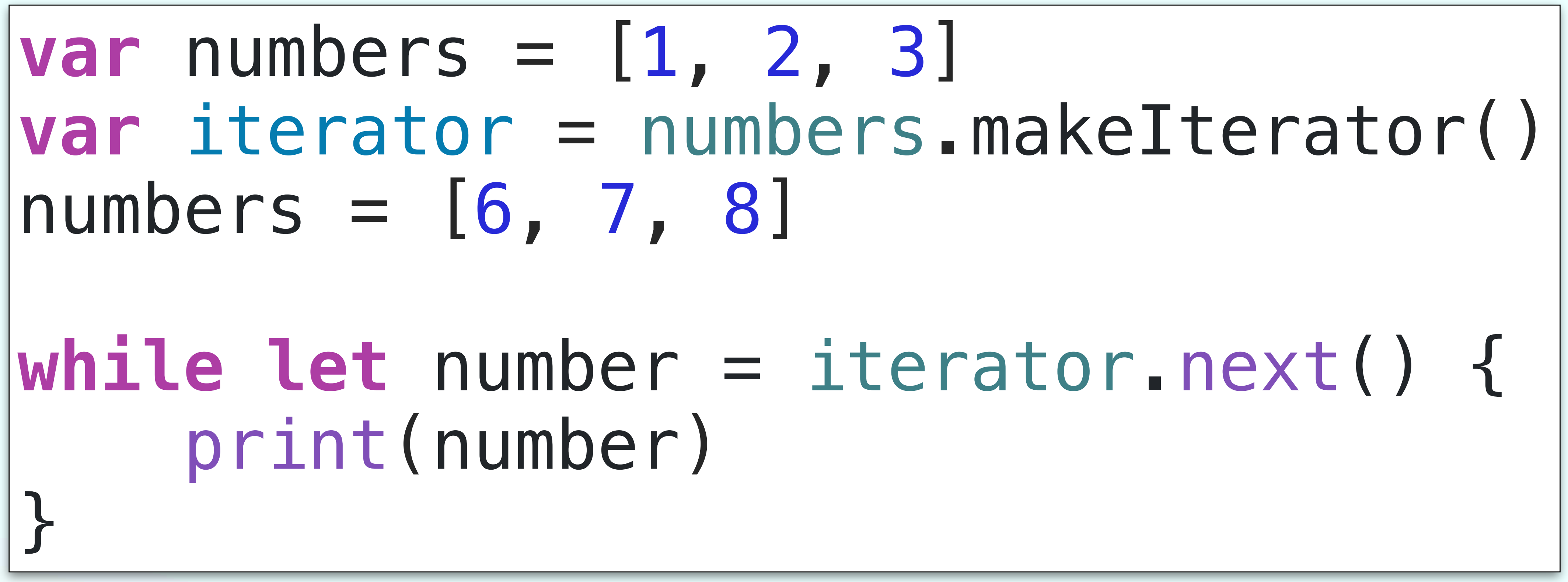
Type	Static type system
Bounds	Checked
Lifetime	Partially enforced/ checked
Initialisation	Enforced
Arithmetic	Checked
Thread	Tsan
Definition	Modules

Swift

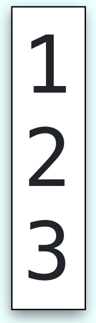
- **Pros:**
  - Memory & thread safe
  - Opt-out runtime checks (for performance)
- **Cons:**
  - Apple specific (at least ecosystem)
  - Immature C++ interop

Type	Static type system
Bounds	Checked
Lifetime	Value semantics & Ref-counted
Initialisation	Enforced
Arithmetic	Trap/explicit behaviour
Thread	Enforced actors & sendable
Definition	Modules

Swift

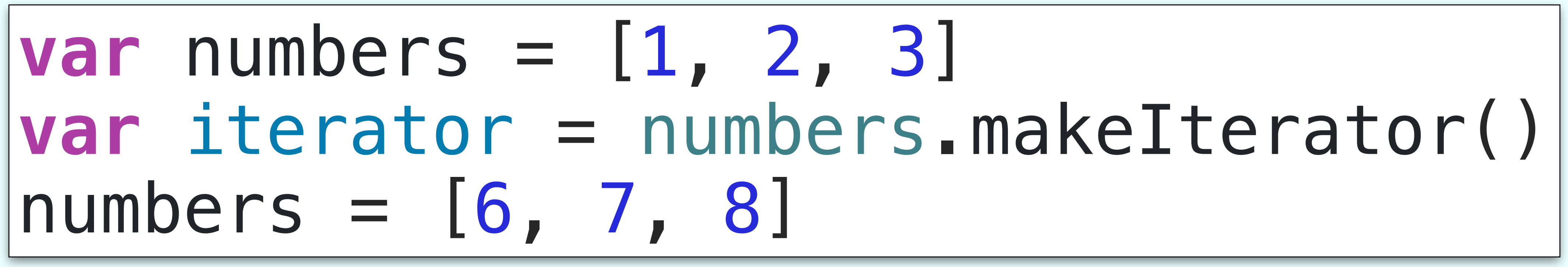


```swift
var numbers = [1, 2, 3]
var iterator = numbers.makeIterator()
numbers = [6, 7, 8]

while let number = iterator.next() {
    print(number)
}
```

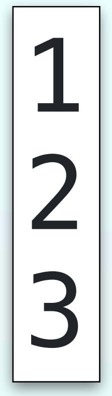


```
1
2
3
```

Swift

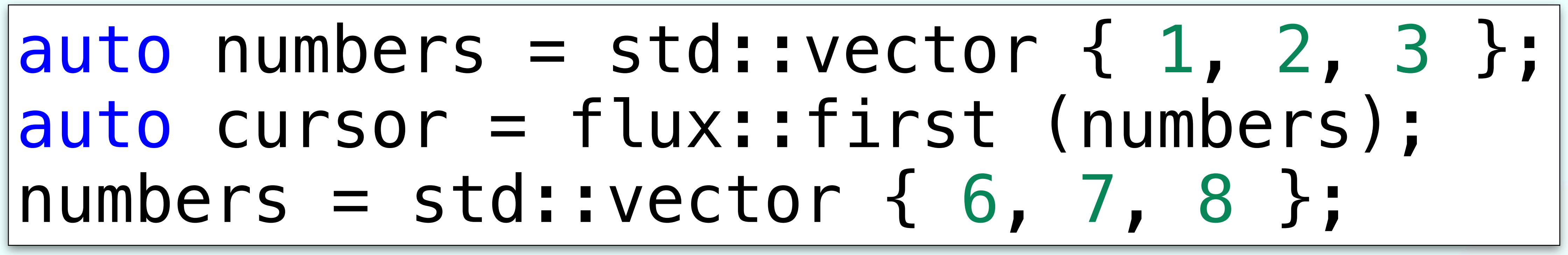


```swift
var numbers = [1, 2, 3]
var iterator = numbers.makeIterator()
numbers = [6, 7, 8]
```

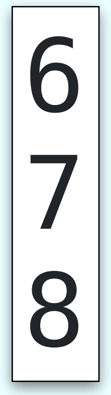


```
1
2
3
```

C++



```cpp
auto numbers = std::vector { 1, 2, 3 };
auto cursor = flux::first (numbers);
numbers = std::vector { 6, 7, 8 };
```



```
6
7
8
```

Which is correct?

Rust

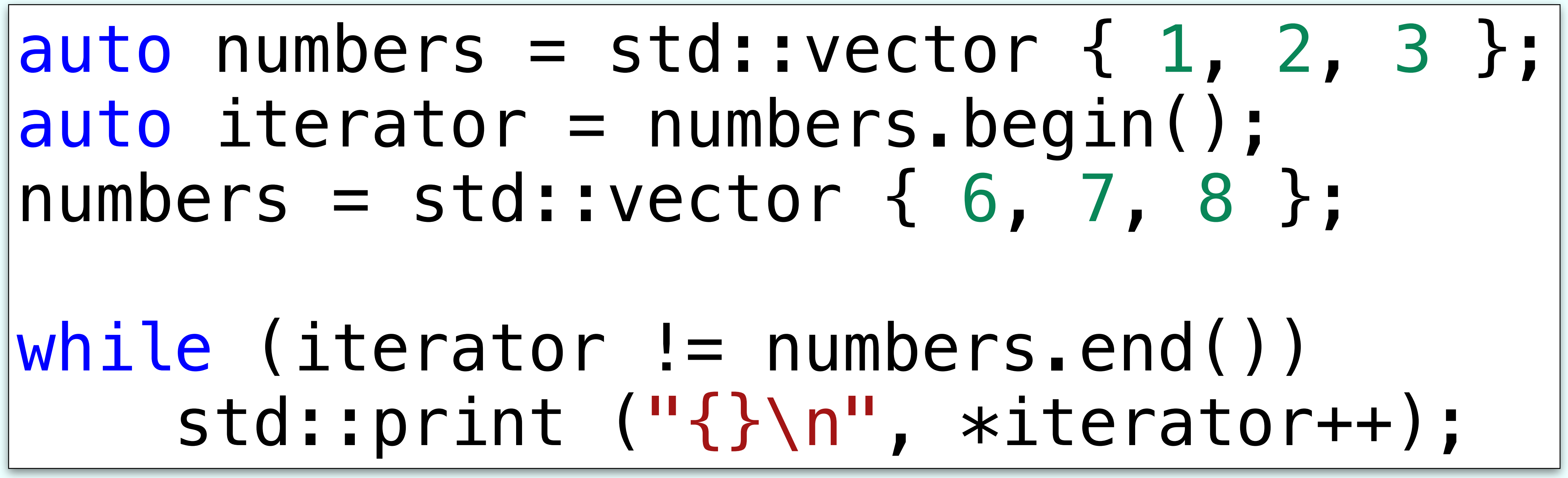
- **Pros:**
  - Memory & thread safe
  - Almost completely statically enforced
  - Sensible defaults (checked bounds and arithmetic*)
- **Cons:**
  - Completely new language
  - Lots of keywords and annotation
  - “Fighting the borrow checker”
  - C++ interop not great

Type	Static type system
Bounds	Checked
Lifetime	Enforced borrow checker
Initialisation	Enforced
Arithmetic	Checked/defined
Thread	Enforced sync/send & BC
Definition	Modules

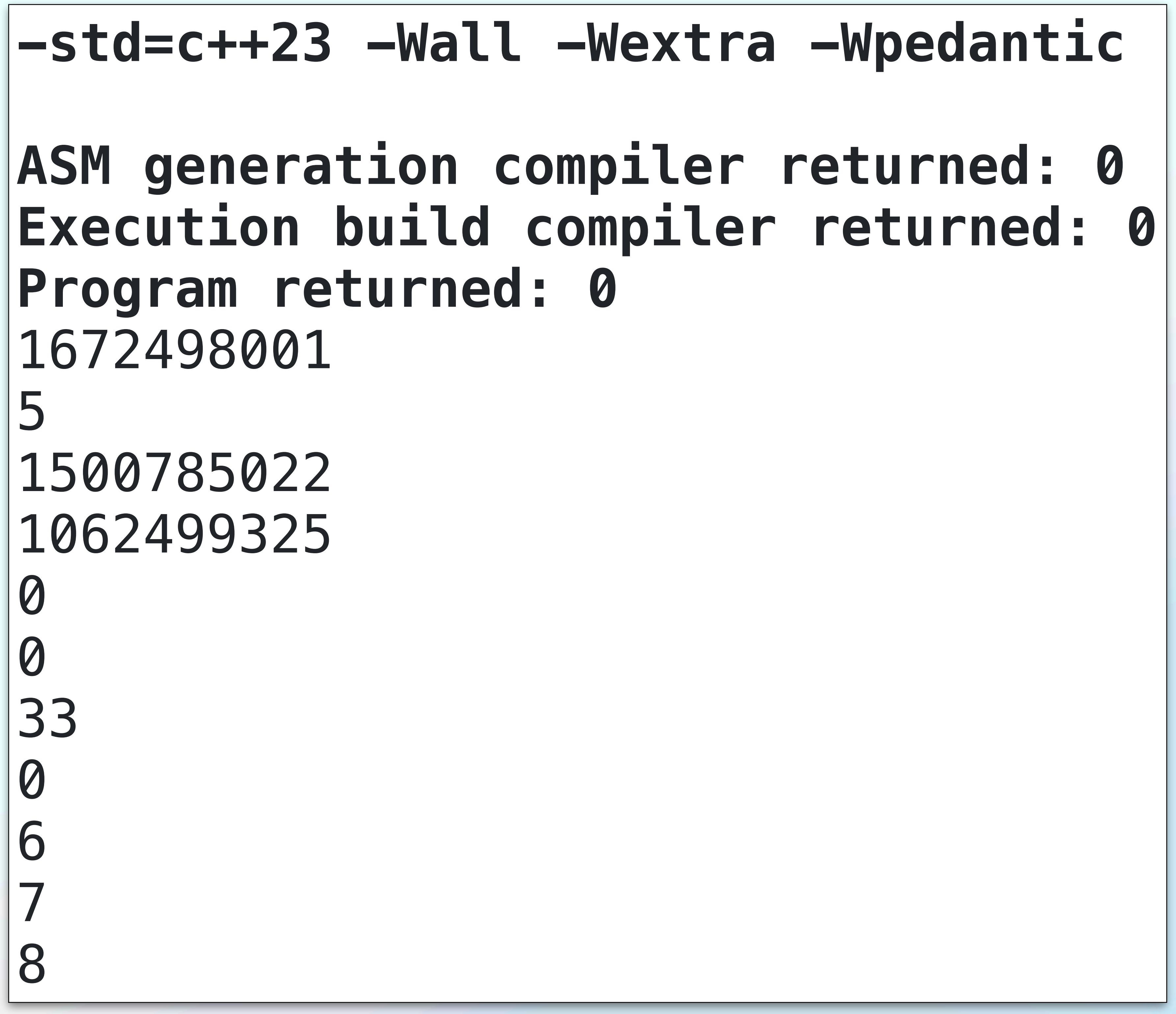
Law of Exclusivity

C++

Iterator invalidation

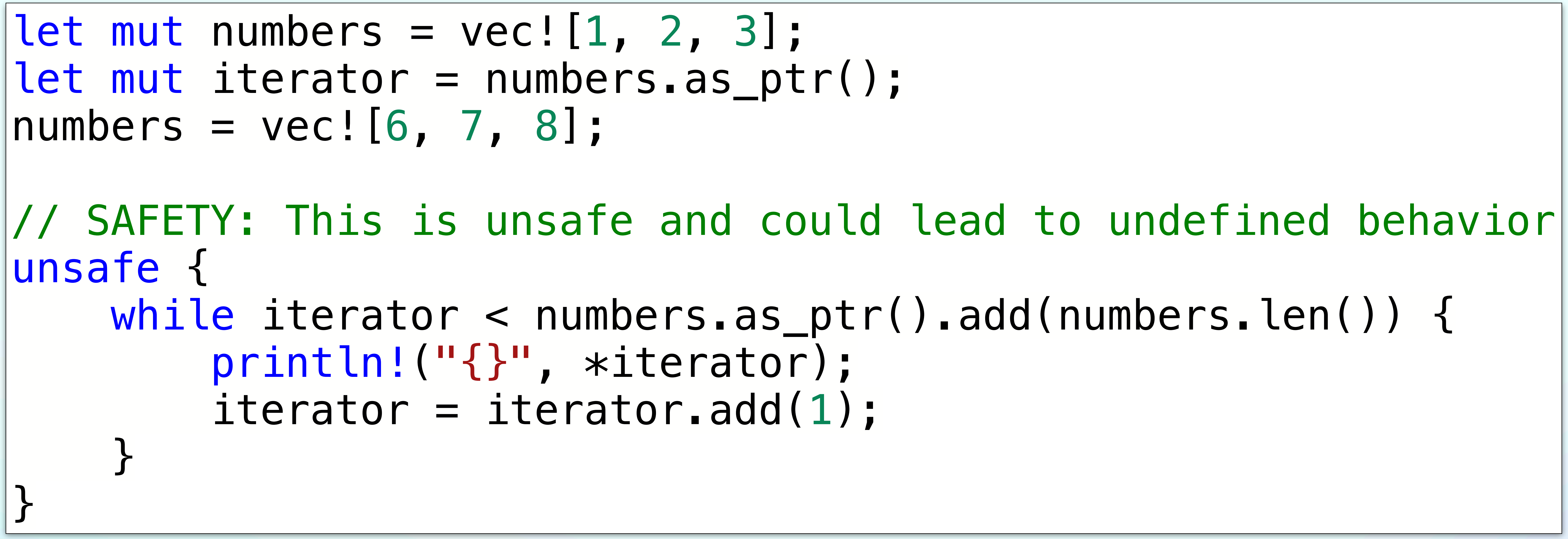


```cpp
auto numbers = std::vector { 1, 2, 3 };
auto iterator = numbers.begin();
numbers = std::vector { 6, 7, 8 };

while (iterator != numbers.end())
    std::print ("{}\n", *iterator++);
```

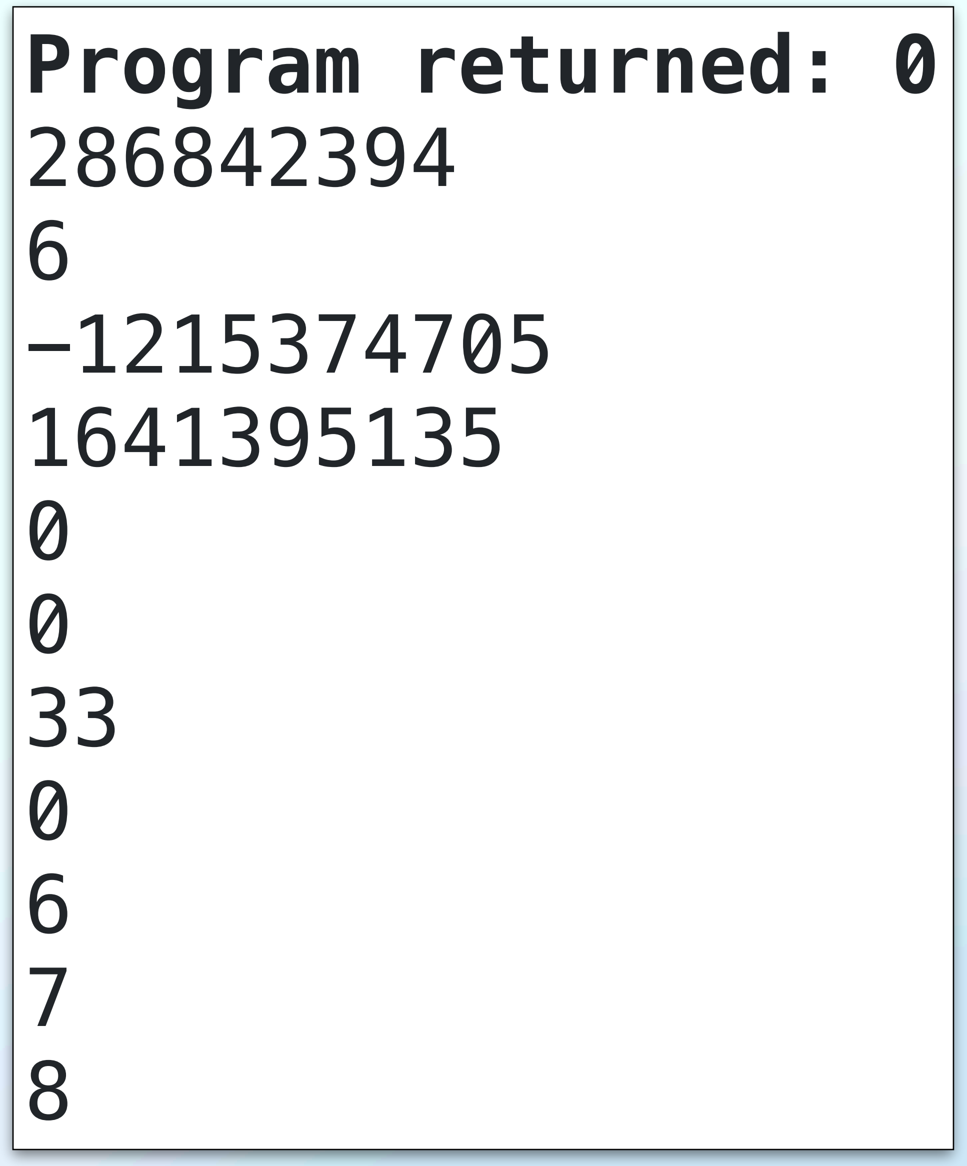


```
-std=c++23 -Wall -Wextra -Wpedantic

ASM generation compiler returned: 0
Execution build compiler returned: 0
Program returned: 0
1672498001
5
1500785022
1062499325
0
0
33
0
6
7
8
```

Rust (unsafe)



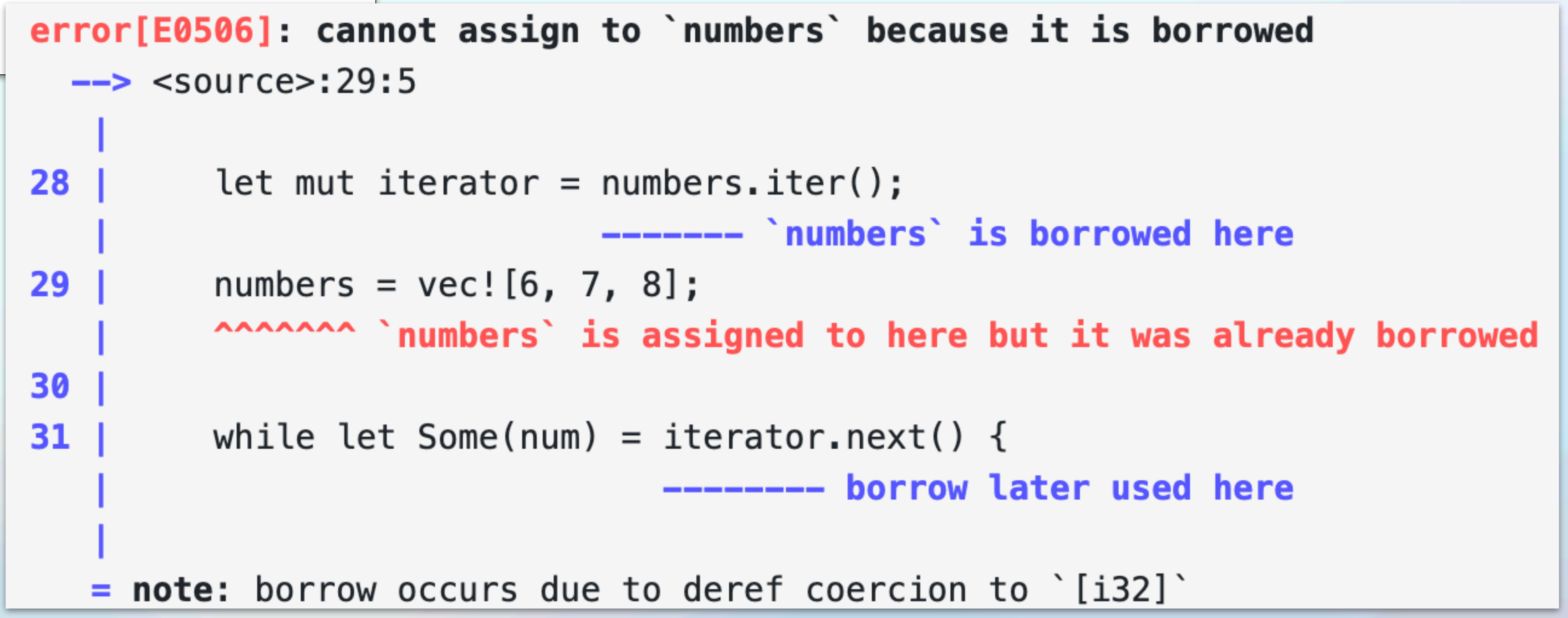
```rust
let mut numbers = vec![1, 2, 3];
let mut iterator = numbers.as_ptr();
numbers = vec![6, 7, 8];

// SAFETY: This is unsafe and could lead to undefined behavior
unsafe {
    while iterator < numbers.as_ptr().add(numbers.len()) {
        println!("{}", *iterator);
        iterator = iterator.add(1);
    }
}
```



```
Program returned: 0
286842394
6
-1215374705
1641395135
0
0
33
0
6
7
8
```

Rust (safe)

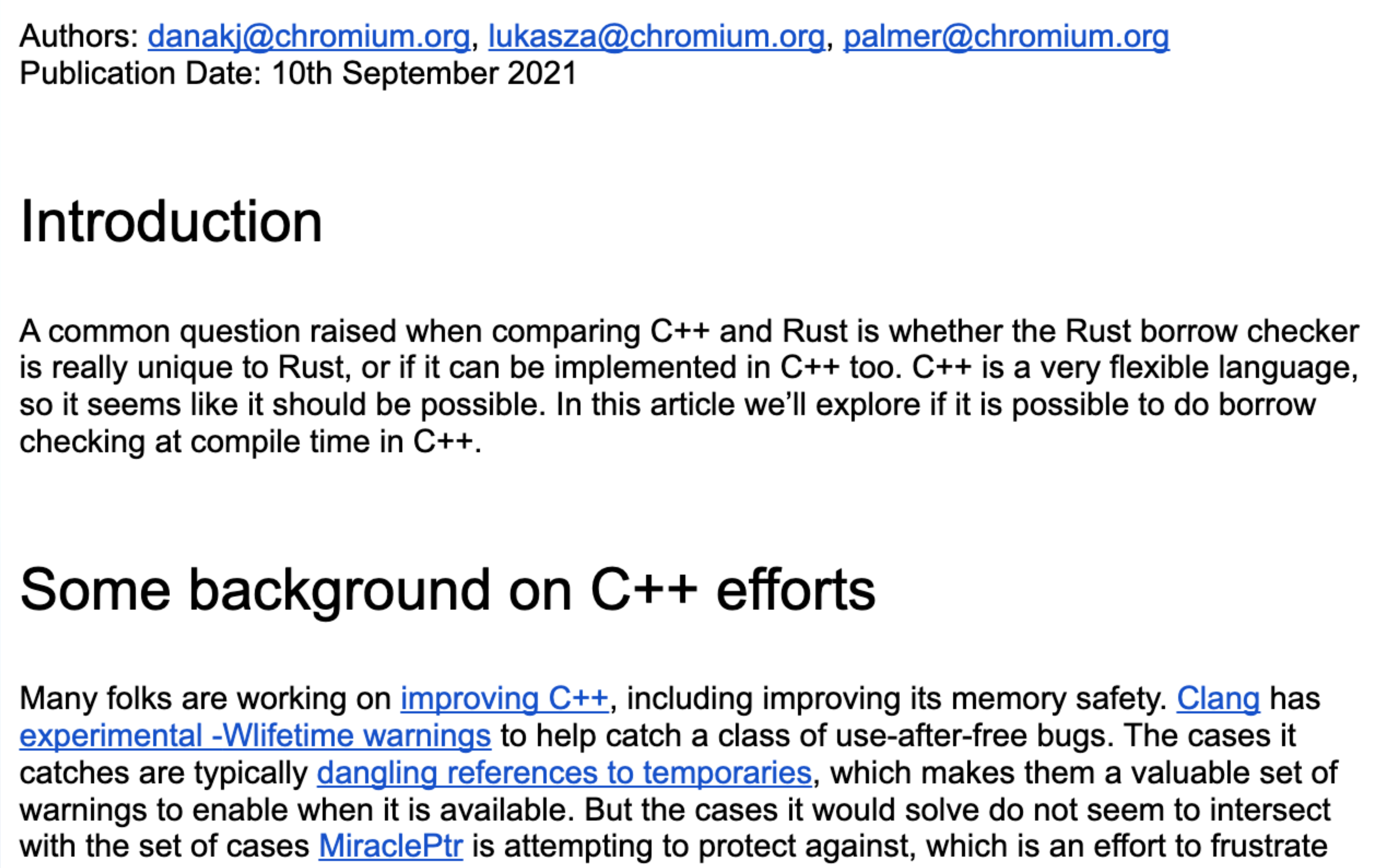
```
let mut numbers = vec![1, 2, 3];
let mut iterator = numbers.iter();
numbers = vec![6, 7, 8];

while let Some(num) = iterator.next() {
    println!("{}", num);
}
```



```
error[E0506]: cannot assign to `numbers` because it is borrowed
  --> <source>:29:5
   |
28 |     let mut iterator = numbers.iter();
   |                        ------- `numbers` is borrowed here
29 |     numbers = vec![6, 7, 8];
   |     ^^^^^^^ `numbers` is assigned to here but it was already borrowed
30 |
31 |     while let Some(num) = iterator.next() {
   |                           -------- borrow later used here
   |
   = note: borrow occurs due to deref coercion to `[i32]`
```

A C++ Borrow-Checker?

Borrowing Trouble: The Difficulties Of A C++ Borrow-Checker



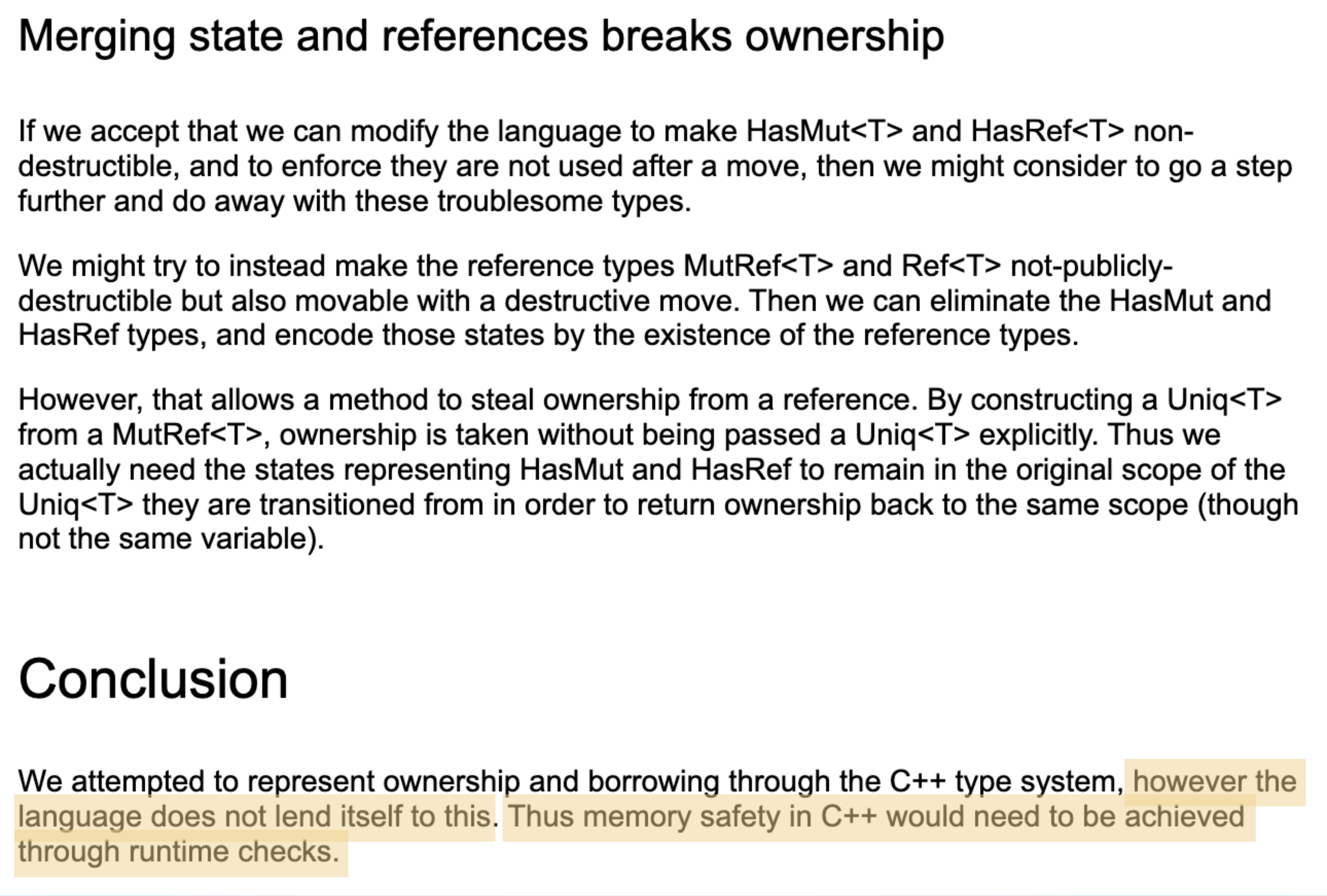
Authors: danakj@chromium.org, lukasza@chromium.org, palmer@chromium.org
Publication Date: 10th September 2021

Introduction

A common question raised when comparing C++ and Rust is whether the Rust borrow checker is really unique to Rust, or if it can be implemented in C++ too. C++ is a very flexible language, so it seems like it should be possible. In this article we'll explore if it is possible to do borrow checking at compile time in C++.

Some background on C++ efforts

Many folks are working on improving C++, including improving its memory safety. Clang has experimental -Wlifetime warnings to help catch a class of use-after-free bugs. The cases it catches are typically dangling references to temporaries, which makes them a valuable set of warnings to enable when it is available. But the cases it would solve do not seem to intersect with the set of cases MiraclePtr is attempting to protect against, which is an effort to frustrate



Merging state and references breaks ownership

If we accept that we can modify the language to make HasMut<T> and HasRef<T> non-destructible, and to enforce they are not used after a move, then we might consider to go a step further and do away with these troublesome types.

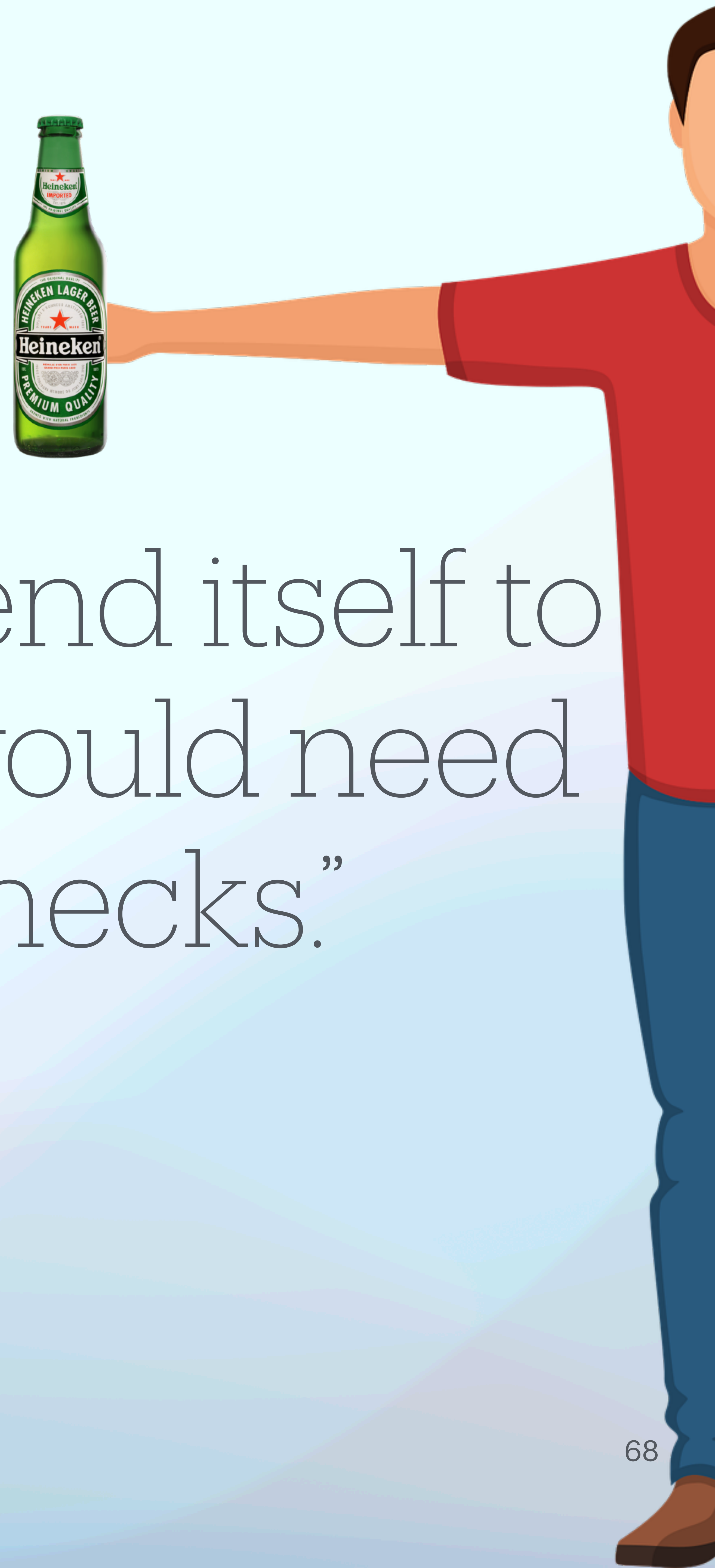
We might try to instead make the reference types MutRef<T> and Ref<T> not-publicly-destructible but also movable with a destructive move. Then we can eliminate the HasMut and HasRef types, and encode those states by the existence of the reference types.

However, that allows a method to steal ownership from a reference. By constructing a Uniq<T> from a MutRef<T>, ownership is taken without being passed a Uniq<T> explicitly. Thus we actually need the states representing HasMut and HasRef to remain in the original scope of the Uniq<T> they are transitioned from in order to return ownership back to the same scope (though not the same variable).

Conclusion

We attempted to represent ownership and borrowing through the C++ type system, however the language does not lend itself to this. Thus memory safety in C++ would need to be achieved through runtime checks.

“However, the language does not lend itself to this. Thus memory safety in C++ would need to be achieved through runtime checks.”



Circle

```
auto numbers = std2::vector<int> { 1, 2, 3 };
auto iterator = numbers.iter();
numbers = std2::vector<int> { 4, 5, 6 };

for (auto number : iterator)
    std2::println (number);
```

```
safety: during safety checking of int main() safe
  borrow checking: example.cpp:10:24
      for (auto number : iterator)
                         ^
  use of iterator depends on expired loan
  drop of numbers between its shared borrow and its use
  invalidating operation at example.cpp:8:13
      numbers = std2::vector<int> { 4, 5, 6 };
              ^
  loan created at example.cpp:7:21
      auto iterator = numbers.iter();
```

```
template<class T+>
class vector
{
public:
  using value_type = T;
  using size_type = std::size_t;

  //…

  [[unsafe::drop_only(T)]]
  ~vector() safe {
    // TODO: std::destroy_n() doesn't seem to
    // like `int^` as a value_type
    // eventually we should fix this

    unsafe {
      auto const* end = self.data() + self.size();
      auto* pos = self^.data();

      while (pos < end) {
        auto t = __rel_read(pos);
        drp t;
        ++pos;
      }

      ::operator delete(p_);
    }
  }
```

```
template<class T+>
class
[[unsafe::send(T~is_send), unsafe::sync(T~is_send)]]
mutex
{
  using mutex_type = unsafe_cell<std::mutex>;

  unsafe_cell<T> data_;
  box<mutex_type> mtx_;


public:
  class lock_guard/(a)
  {
    friend class mutex;

    mutex const^/a m_;

    lock_guard(mutex const^/a m) noexcept safe
      : m_(m)
    {
    }
```

Rust

```
pub(super) struct PthreadMutexAttr<'a>(pub &'a mut MaybeUninit<libc::pthread_mutexattr_t>);

impl Drop for PthreadMutexAttr<'_> {
    fn drop(&mut self) {
        unsafe {
            let result = libc::pthread_mutexattr_destroy(self.0.as_mut_ptr());
            debug_assert_eq!(result, 0);
        }
    }
}
```

Circle is safe C++

Circle is (statically*) safe C++

sean@red: ~/projects/circle4/talk
Search Terminal Help
~/projects/circle4/talk/safe1.cxx (talk) - Sublime Text
File Edit Selection Find View Goto Tools Project Preferences Help
safe1.cxx
safety.png
File Edit View Go Help
Sean Baxter
924 × 607 pixels 84.6 kB 100%
2 / 2
Line 1, Column 1
lifetime (22949)
Spaces: 2
C++

```
ction does not return a value [-Wreturn-type]

enerated.
ng CXX executable c
 target circle
rojects/circle4/Rel
rojects/circle4/Rel
ord for sean:
[100%] Built target circle
Install the project...
-- Install configuration: "Rele
-- Installing: /usr/bin/circle
-- Set runtime path of "/usr/bi
sean@red:~/projects/circle4/Rel
[sudo] password for sean:
Scanning dependencies of target
[  0%] Building CXX object CMak
.o
/home/sean/projects/circle4/src
meration values 'language_kind_
ied' not handled in switch [-Ws
          switch(frontend.o
                    ^

1 warning generated.
[  1%] Linking CXX executable c
[100%] Built target circle
Install the project...
-- Install configuration: "Rele
-- Installing: /usr/bin/circle
-- Set runtime path of "/usr/bi
sean@red:~/projects/circle4/Rel
Scanning dependencies of target
[  0%] Building CXX object CMak
riance.cxx.o
[  1%] Linking CXX executable c
[100%] Built target circle
Install the project...
-- Install configuration: "Rele
-- Installing: /usr/bin/circle
-- Set runtime path of "/usr/bi
sean@red:~/projects/circle4/Rel
sean@red:~/projects/circle4/talk$ subl .
sean@red:~/projects/circle4/talk$ xdg-open billion.png
sean@red:~/projects/circle4/talk$
```

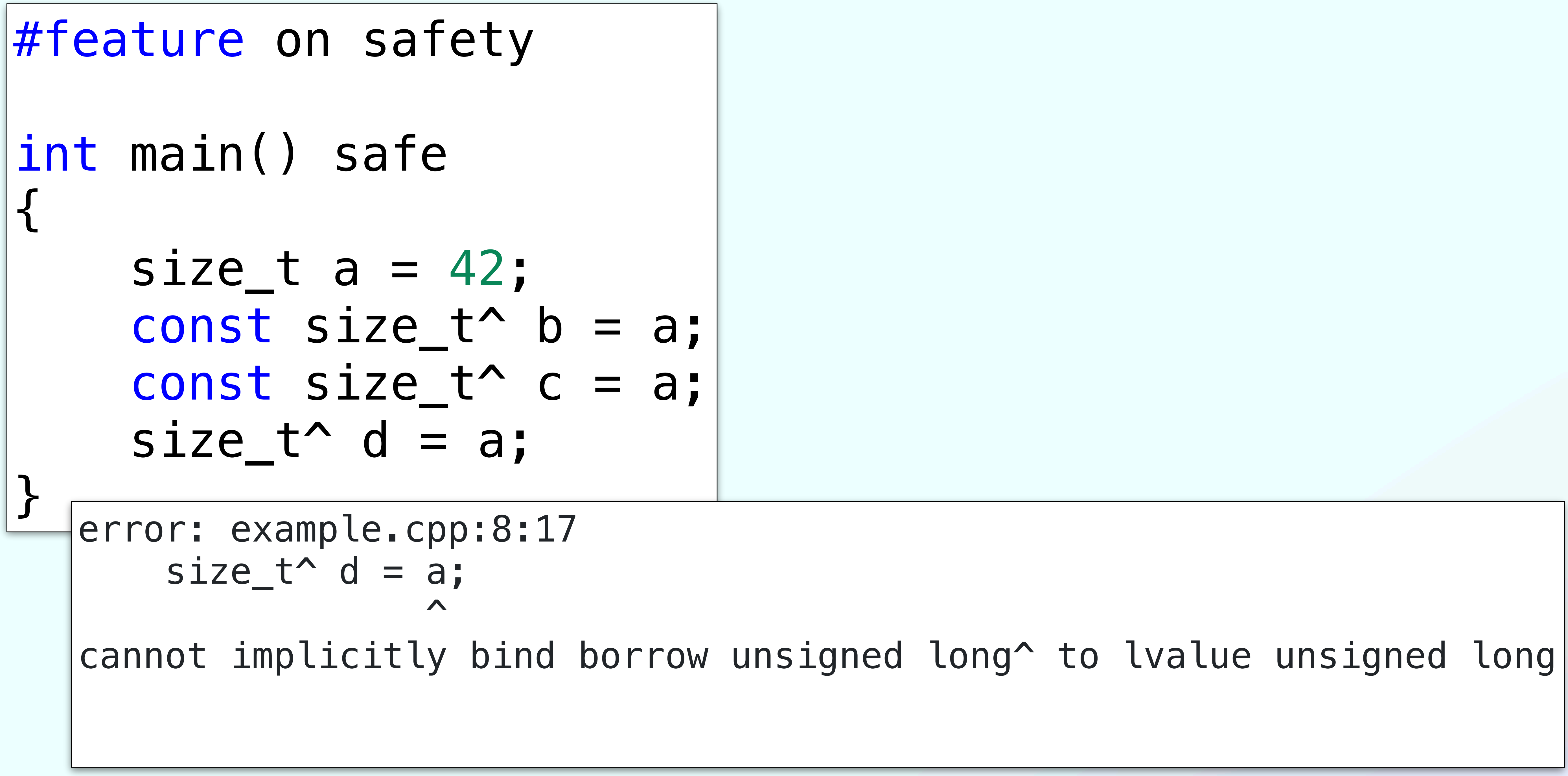
Kinds of memory safety and their solutions

- Lifetime safety - static
  - Borrow checking.
  - A local solution to a non-local problem.
- Type safety (nullptr variety) - static
  - Relocation object model.
- Type safety (union variety) - static
  - Choice types and pattern matching.
- Thread/data race safety - static
  - Send/sync traits.
- Out-of-bounds subscript, divide-by-zero, etc - runtime
  - Panic!

Other unsafe stuff is banned in `safe` contexts.

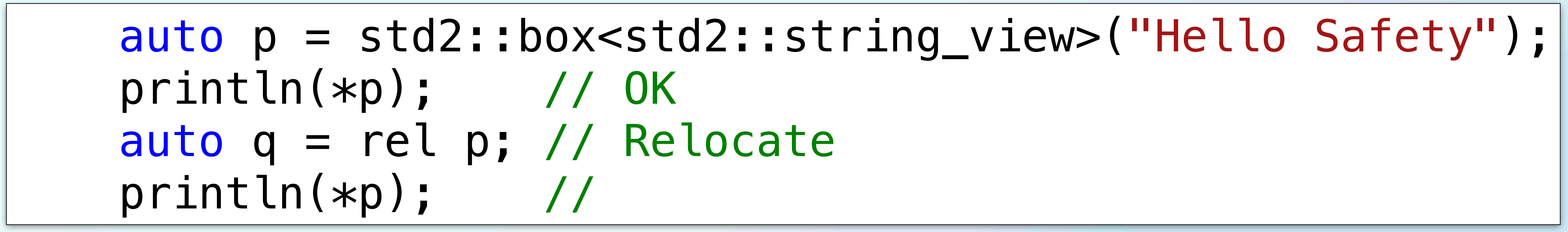
Steps Involved

- Add borrows (similar to references)
- Add relocation (destructive move)
- Add choice type (language variant)
- Add pattern matching
- Add escape hatch (unsafe)
- Add new safe standard library
- Add protocols (type traits)
- Implement sync/send

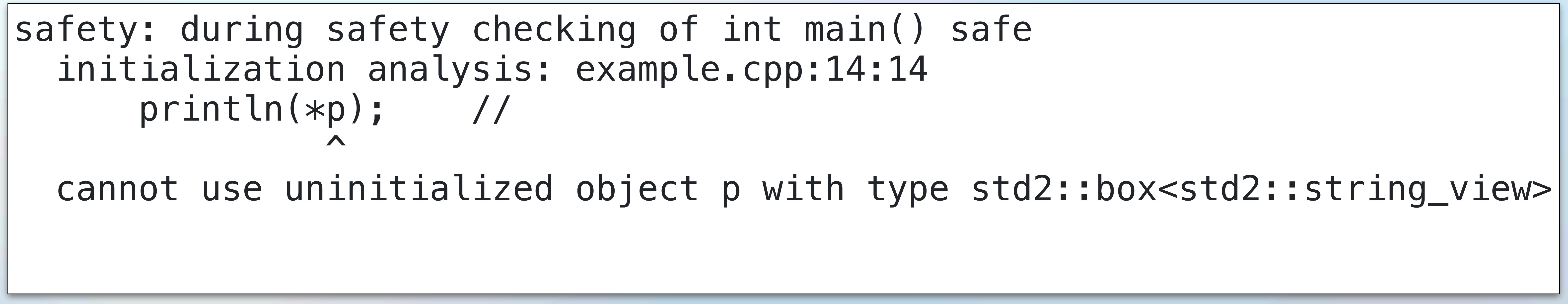


```
#feature on safety

int main() safe
{
    size_t a = 42;
    const size_t^ b = a;
    const size_t^ c = a;
    size_t^ d = a;
}
```

```
error: example.cpp:8:17
    size_t^ d = a;
                ^
cannot implicitly bind borrow unsigned long^ to lvalue unsigned long
```



```
    auto p = std2::box<std2::string_view>("Hello Safety");
    println(*p);    // OK
    auto q = rel p; // Relocate
    println(*p);    //
```



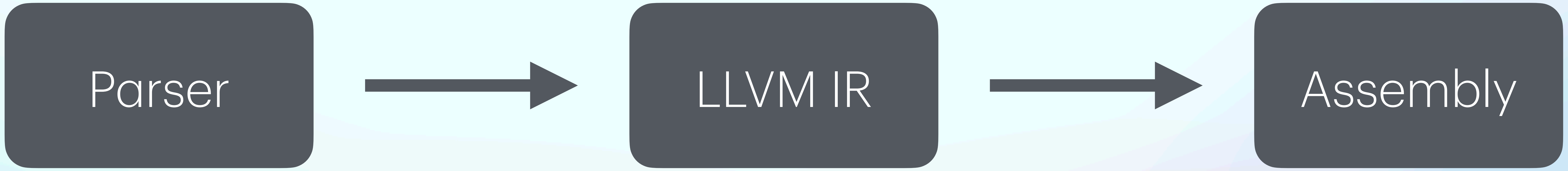
```
safety: during safety checking of int main() safe
  initialization analysis: example.cpp:14:14
      println(*p);    //
             ^
  cannot use uninitialized object p with type std2::box<std2::string_view>
```

Easy?

wg21.link/P3390

Implementation

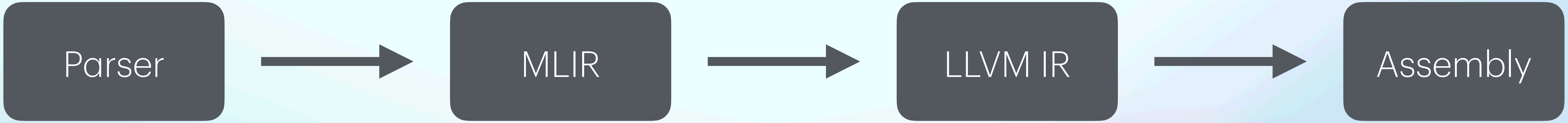
- Old Circle:



Implementation

- Safe Circle:
  “Mid-level-IR” borrow checker/lifetime analysis

- Unlikely feasible in Clang/GCC/MSVC



Circle

- **Pros:**
  - Same level of safety as Rust
  - Almost completely statically enforced
  - Sensible defaults (checked bounds and arithmetic)
  - Incrementally opt-in
  - Perfect C++ interop
- **Cons:**
  - Closed source, individual built compiler (business risk)
  - *Incrementally opt-in*

Type	Static type system
Bounds	Checked
Lifetime	Enforced borrow checker
Initialisation	Enforced
Arithmetic	Checked/defined
Thread	Enforced sync/send & BC
Definition	Modules

What do Circle, Swift and Rust have in common?

Thread safety

Sync & Send

Low-level

Actors

High-level

Sync & Send

- Protocols (like type traits) that are checked
- A ***sync*** object can be safely ***shared*** between threads
- A ***send*** object can be safely ***transferred*** between threads

Sync & Send in Swift

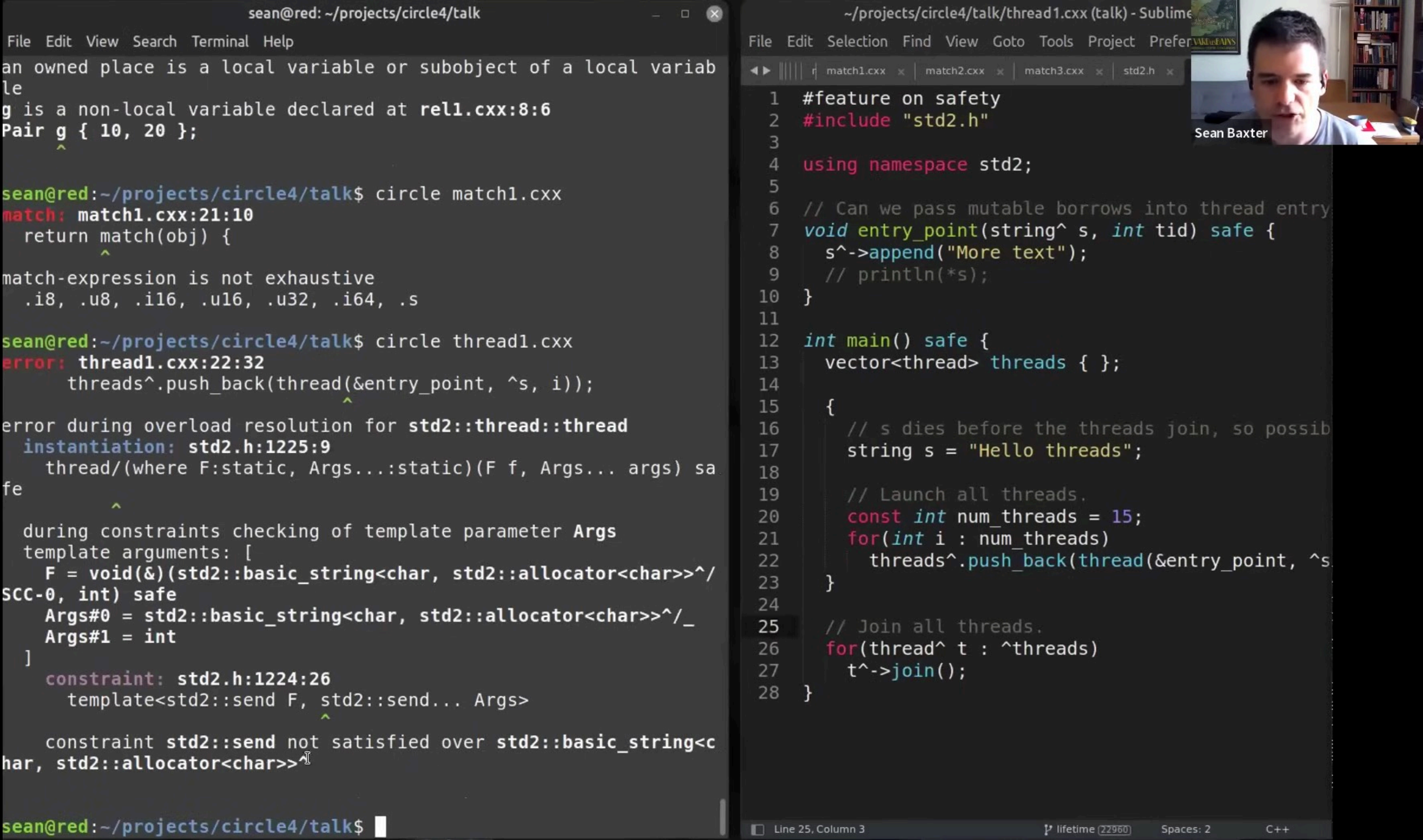
The Sendable Protocol

- Notion of “isolation boundaries” between potential thread execution contexts
- Objects can only pass isolation boundaries if they conform to the `@Sendable` protocol
  - Sendable can be inferred in some cases
- Syncable objects are a special case of Sendable objects
  - E.g. a `LockingResource`
  - *No “syncable” keyword*

```swift
open class Thread : NSObject {

    public convenience init(block: @escaping @Sendable () -> Void)

```

Sync & Send in Circle

- `send` is a “marker interface” (in Rust a “marker trait”)
  - Similar to a C++ “type trait”
- Inferred if:
  - A copy can be made (value semantics)
  - A borrow can shared (`const T^`)
  - **NOT** mutable borrow (T^)

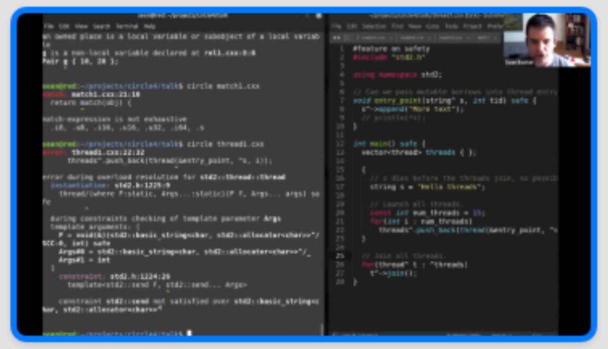

sean@red: ~/projects/circle4/talk
File Edit View Search Terminal Help
~/projects/circle4/talk/thread1.cxx (talk) - Sublime
File Edit Selection Find View Goto Tools Project Prefer
match1.cxx
match2.cxx
match3.cxx
std2.h
Sean Baxter
Line 25, Column 3
lifetime (22960)
Spaces: 2
C++

```
an owned place is a local variable or subobject of a local variab
le
g is a non-local variable declared at rel1.cxx:8:6
Pair g { 10, 20 };
   ^

sean@red:~/projects/circle4/talk$ circle match1.cxx
match: match1.cxx:21:10
  return match(obj) {
         ^

match-expression is not exhaustive
  .i8, .u8, .i16, .u16, .u32, .i64, .s

sean@red:~/projects/circle4/talk$ circle thread1.cxx
error: thread1.cxx:22:32
    threads^.push_back(thread(&entry_point, ^s, i));
                               ^

error during overload resolution for std2::thread::thread
  instantiation: std2.h:1225:9
    thread/(where F:static, Args...:static)(F f, Args... args) sa
fe
       ^

  during constraints checking of template parameter Args
  template arguments: [
    F = void(&)(std2::basic_string<char, std2::allocator<char>>^/
SCC-0, int) safe
    Args#0 = std2::basic_string<char, std2::allocator<char>>^/_
    Args#1 = int
  ]
    constraint: std2.h:1224:26
      template<std2::send F, std2::send... Args>
                                  ^

    constraint std2::send not satisfied over std2::basic_string<c
har, std2::allocator<char>>^


sean@red:~/projects/circle4/talk$
```

```
1  #feature on safety
2  #include "std2.h"
3
4  using namespace std2;
5
6  // Can we pass mutable borrows into thread entry
7  void entry_point(string^ s, int tid) safe {
8    s^->append("More text");
9    // println(*s);
10 }
11
12 int main() safe {
13   vector<thread> threads { };
14
15   {
16     // s dies before the threads join, so possib
17     string s = "Hello threads";
18
19     // Launch all threads.
20     const int num_threads = 15;
21     for(int i : num_threads)
22       threads^.push_back(thread(&entry_point, ^s
23   }
24
25   // Join all threads.
26   for(thread^ t : ^threads)
27     t^->join();
28 }
```

Sync and send in C++?

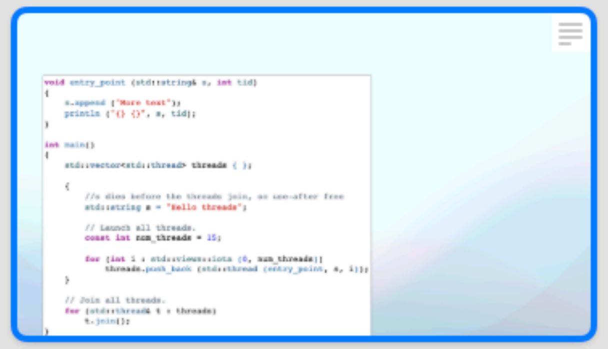
scl - Safe Concurrency Library



88

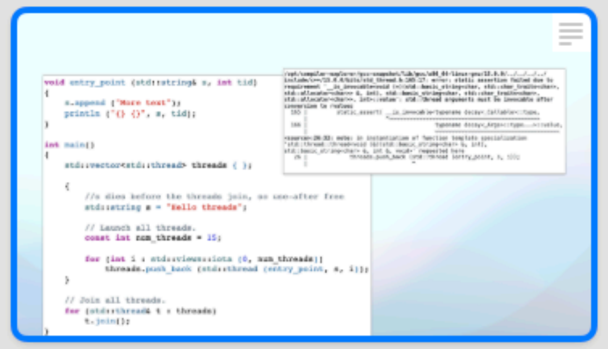
Sync and send in C++?

89

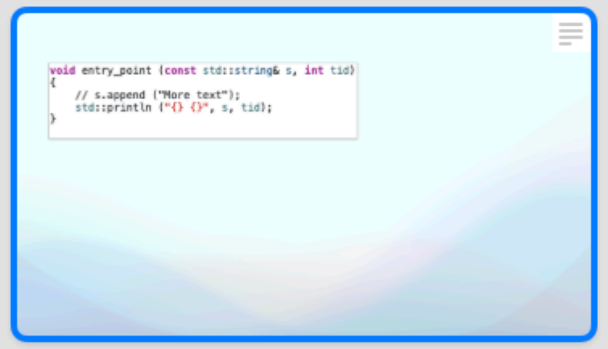


90

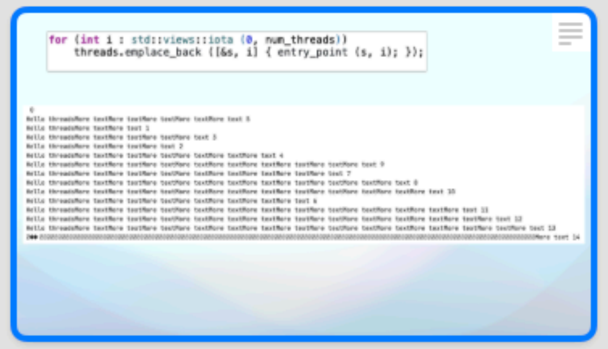
91



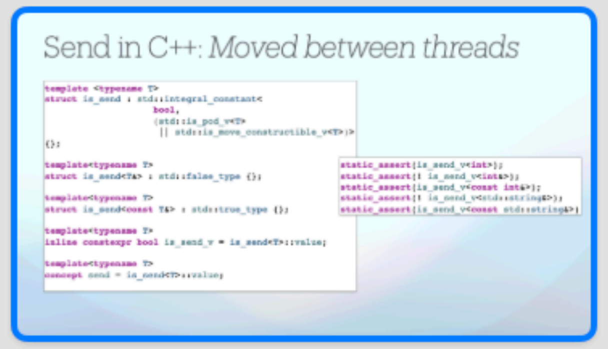
92



93



94



Send in C++: *Moved between threads*

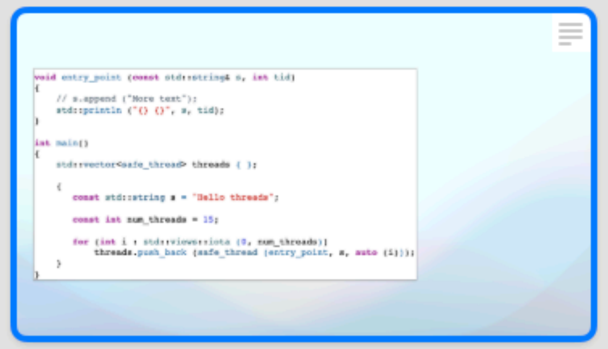
95

96

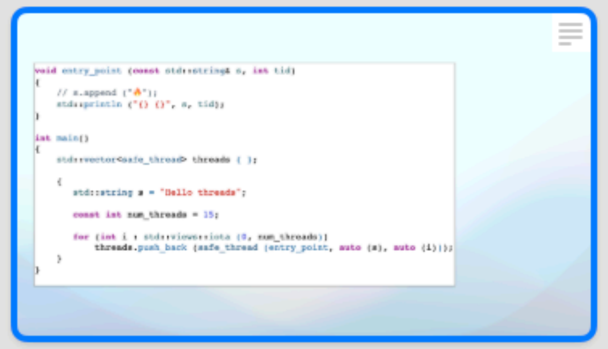
97

98

99



100



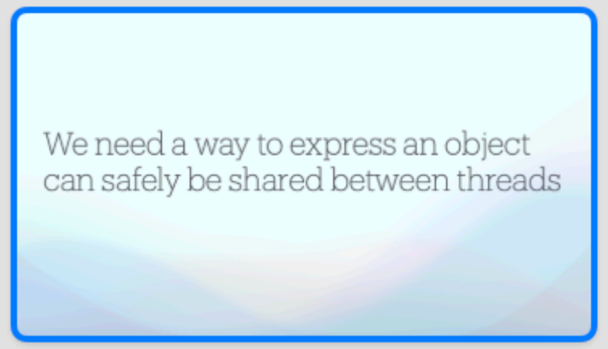
101

`std::shared_ptr` to the rescue!

102

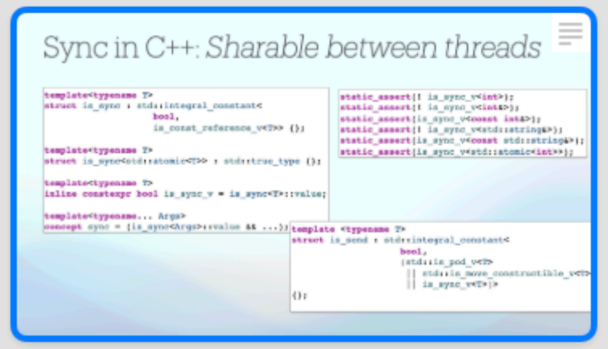
103

104



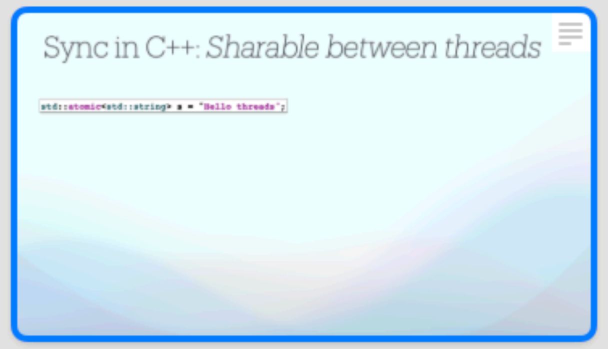
We need a way to express an object can safely be shared between threads

105



Sync in C++: *Sharable between threads*

106



Sync in C++: *Sharable between threads*

107

108

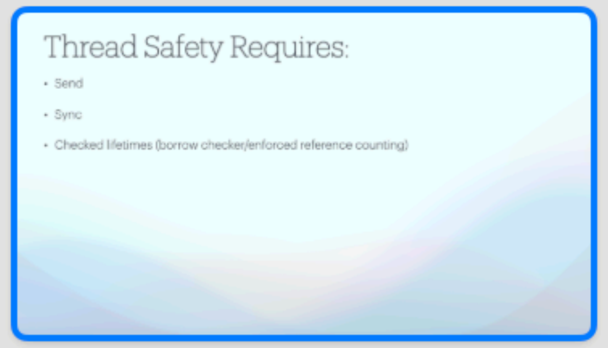
109

110

111

Problems

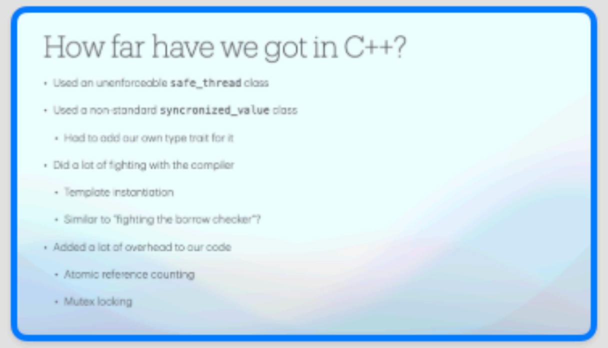
112



Thread Safety Requires:

- Send
- Sync
- Checked lifetimes (borrow checker/enforced reference counting)

113



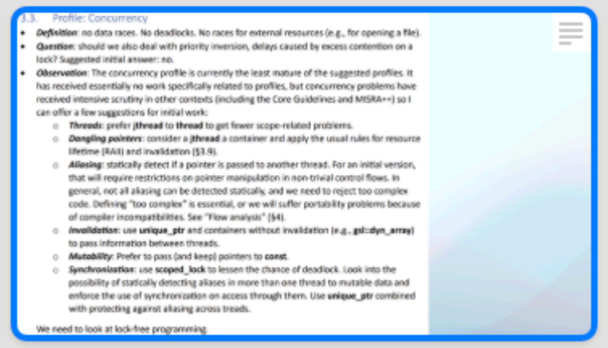
How far have we got in C++?

- Used an unenforceable `safe_thread` class
- Used a non-standard `syncronized_value` class
  - Had to add our own type trait for it
- Did a lot of fighting with the compiler
  - Template instantiation
  - Similar to "fighting the borrow checker"?
- Added a lot of overhead to our code
  - Atomic reference counting
  - Mutex locking

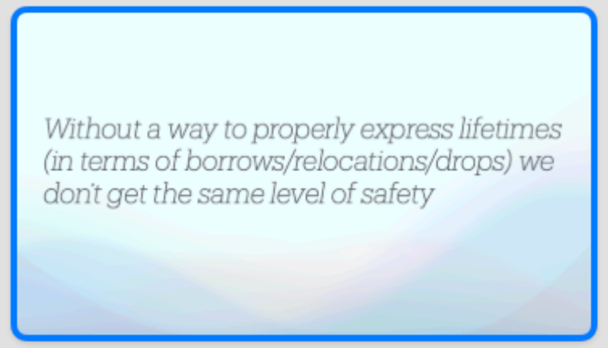
114

- Not bullet proof
  - C++ "aliasing"
- Not beginner friendly
- Not default

115

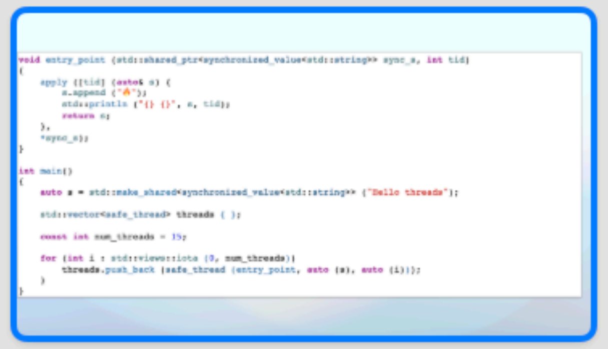


116

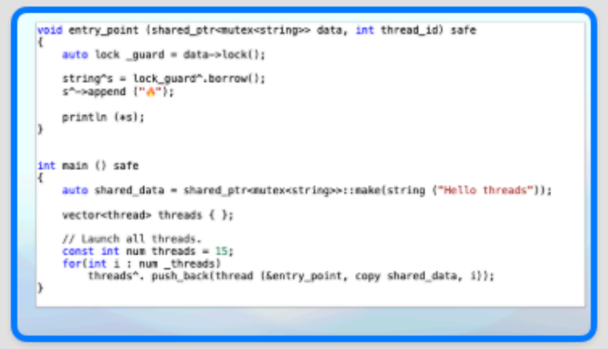


Without a way to properly express lifetimes (in terms of borrows/relocations/drops) we don't get the same level of safety

117



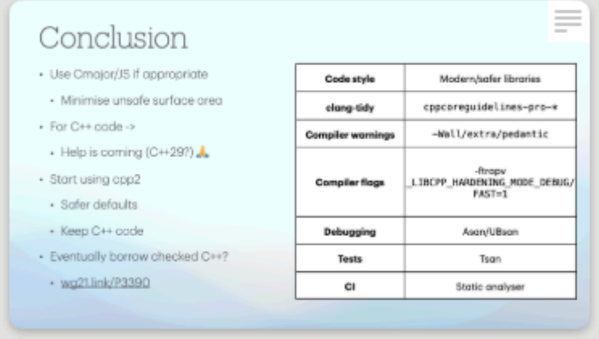
118



119

120

121

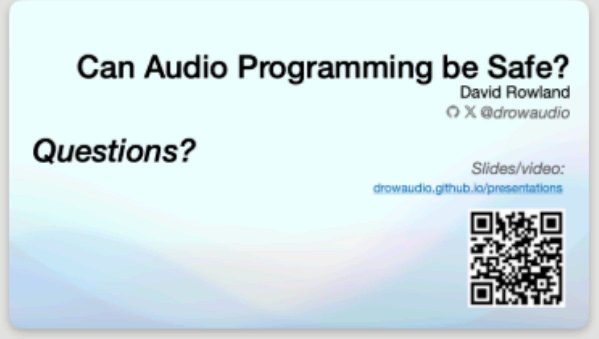


Conclusion

- Use Cmajor/JS if appropriate
  - Minimise unsafe surface area
- For C++ code ->
  - Help is coming (C++29?) 🙏
- Start using cpp2
  - Safer defaults
  - Keep C++ code
- Eventually borrow checked C++?
  - wg21.link/P3390

Code style	Modern/safer libraries
clang-tidy	cppcoreguidelines-pro-*
Compiler warnings	-Wall/extra/pedantic
Compiler flags	-ftrapv _LIBCPP_HARDENING_MODE_DEBUG/FAST=1
Debugging	Asan/UBsan
Tests	Tsan
CI	Static analyser

122



Can Audio Programming be Safe?
David Rowland
@drowaudio

Questions?

Slides/video:
drowaudio.github.io/presentations

123

```
class safe_thread
{
public:
    template<typename F, send... Args>
    safe_thread (F&& f, Args&&... args)
        : thread (std::forward<F> (f), std::forward<Args> (args)...)
    {
        static_assert((is_function_pointer_v<std::decay_t<std::decay_t<F>>>
                       && ! std::is_member_function_pointer_v<std::decay_t<F>>)
                      || is_send_v<std::decay_t<F>>);
    }

    safe_thread (safe_thread&& other)
        : thread (std::move (other.thread))
    {
    }

private:
    std::jthread thread;
};
```

Send in C++: *Moved between threads*

```
template<typename F, send... Args>
safe_thread (F&& f, Args&&... args)
    : thread (std::forward<F> (f), std::forward<Args> (args)...)
{
}
```

```
template<typename T>
struct is_send : std::integral_constant<
        bool,
        (! (std::is_lvalue_reference_v<T>
            || std::is_pointer_v<std::remove_extent_t<T>>
            || is_lambda_v<T>))
         &&
           (std::is_move_constructible_v<T>
            || (is_function_pointer_v<std::decay_t<T>>
                && ! std::is_member_function_pointer_v<T>))>
{};

template<typename T>
concept send = is_send<T>::value;
```

```
static_assert(is_send_v<const int>);
static_assert(is_send_v<int>);
static_assert(is_send_v<int&&>);
static_assert(is_send_v<int>);

static_assert(! is_send_v<int&>);
static_assert(! is_send_v<int*&>);
static_assert(! is_send_v<const int&>);
static_assert(! is_send_v<const int*&>);
static_assert(! is_send_v<std::string&>);
static_assert(! is_send_v<const std::string&>);
static_assert(! is_send_v<std::string*&>);
static_assert(! is_send_v<const std::string*&>);
```

Send in C++: *Moved between threads*

- ***No***
  - lvalue references
  - Object pointers
  - Lambdas
  - *May be referenced outside this thread boundary*
- ***Only***
  - rvalues
  - Non-member function pointers
  - *Can be sure no data is shared*

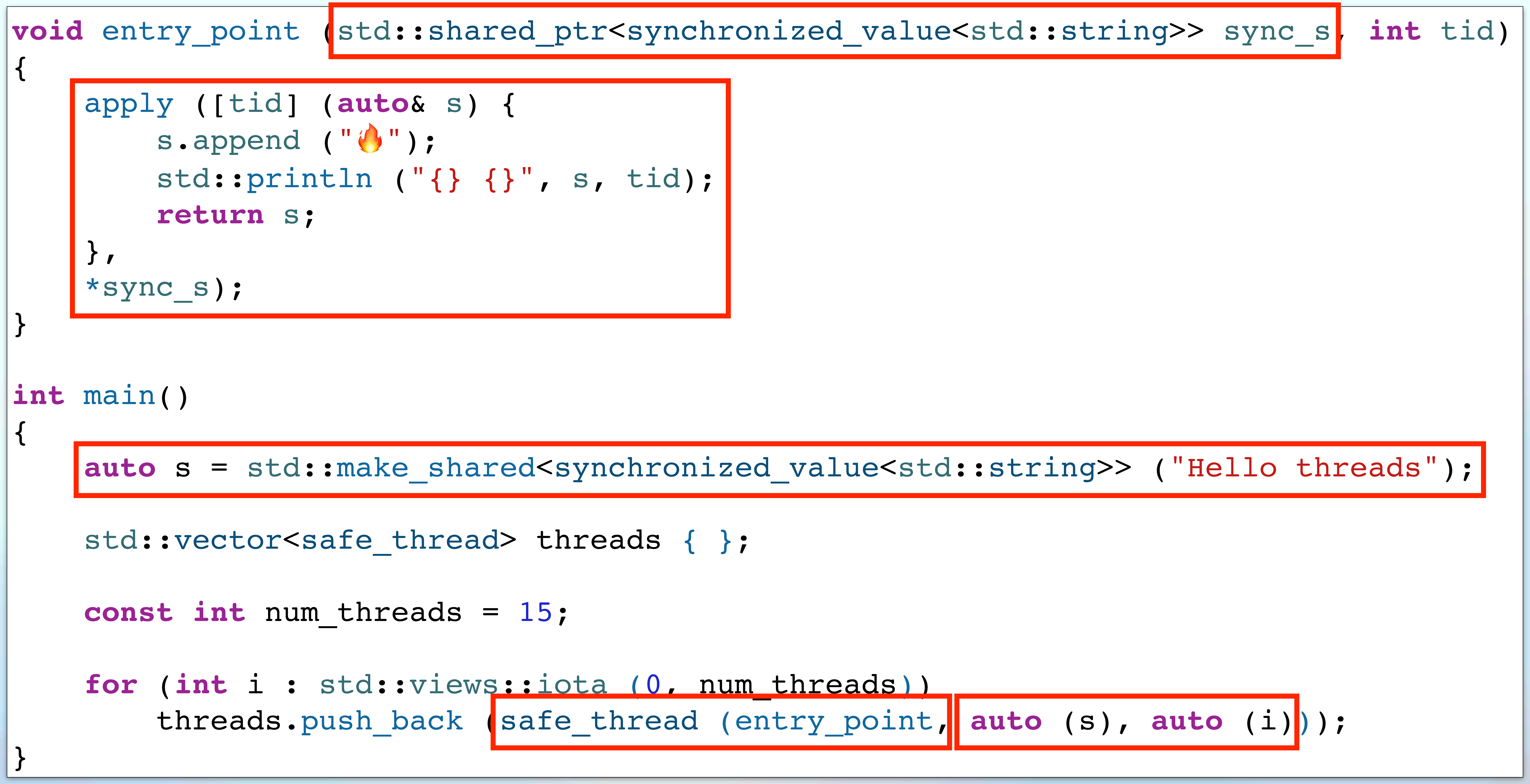
Sync in C++: *Sharable between threads*

```cpp
template<typename T>
struct is_sync : std::false_type {};

template<typename T>
struct is_sync<std::atomic<T>> : std::true_type {};

template<typename T>
inline constexpr bool is_sync_v = is_sync<T>::value;

template<typename... Args>
concept sync = (is_sync<Args>::value && ...);
```

```cpp
static_assert(! is_sync_v<int>);
static_assert(! is_sync_v<int&>);
static_assert(! is_sync_v<const int&>);
static_assert(! is_sync_v<std::string&>);
static_assert(! is_sync_v<const std::string&>);
static_assert(is_sync_v<std::atomic<int>>);
```

```
template <typename T>
struct is_send : std::integral_constant<
                    bool,
                    (! (std::is_lvalue_reference_v<T>
                        || std::is_pointer_v<std::remove_extent_t<T>>
                        || is_lambda_v<T>))
                    &&
                    (std::is_move_constructible_v<T>
                     || (is_function_pointer_v<std::decay_t<T>>
                         && ! std::is_member_function_pointer_v<T>)
                     || is_sync_v<T>)>
{};
```

```
template<sync T>
struct is_send<std::shared_ptr<T>> : std::true_type
{};
```



```cpp
void entry_point (std::shared_ptr<synchronized_value<std::string>> sync_s, int tid)
{
    apply ([tid] (auto& s) {
        s.append ("🔥");
        std::println ("{} {}", s, tid);
        return s;
    },
    *sync_s);
}

int main()
{
    auto s = std::make_shared<synchronized_value<std::string>> ("Hello threads");

    std::vector<safe_thread> threads { };

    const int num_threads = 15;

    for (int i : std::views::iota (0, num_threads))
        threads.push_back (safe_thread (entry_point, auto (s), auto (i)));
}
```

Problems

```
template<typename T>
struct Node
{
    Node* next;
    Node* prev;
};
```

```
void entry_point (std::shared_ptr<synchronized_value<~~std::string~~>> sync_s, int tid)
{
    apply ([tid] (auto& s) {
        //...
        return s;
    },
    *sync_s);
}

int main()
{
        //...
        auto s = std::make_shared<synchronized_value<~~std::string~~>> ("Hello threads");
        //...
}
```

Problems

```
void setGlobalString (std::string*);

void entry_point (std::shared_ptr<synchronized_value<std::string>> sync_s, int tid)
{
    apply ([tid] (auto& s) {
        setGlobalString (&s);
        //...
        return s;
    },
    *sync_s);
}

int main()
{
        //...
        auto s = std::make_shared<synchronized_value<std::string>> ("Hello threads");
        //...
}
```

Problems

```
threads.push_back (safe_thread (entry_point, auto (s), auto (i)));
```

```
threads.push_back (safe_thread ([this]
                                {
                                    memberFunction();
                                });
```

How far have we got in C++?

Safer, but not safe™

How far have we got in C++?

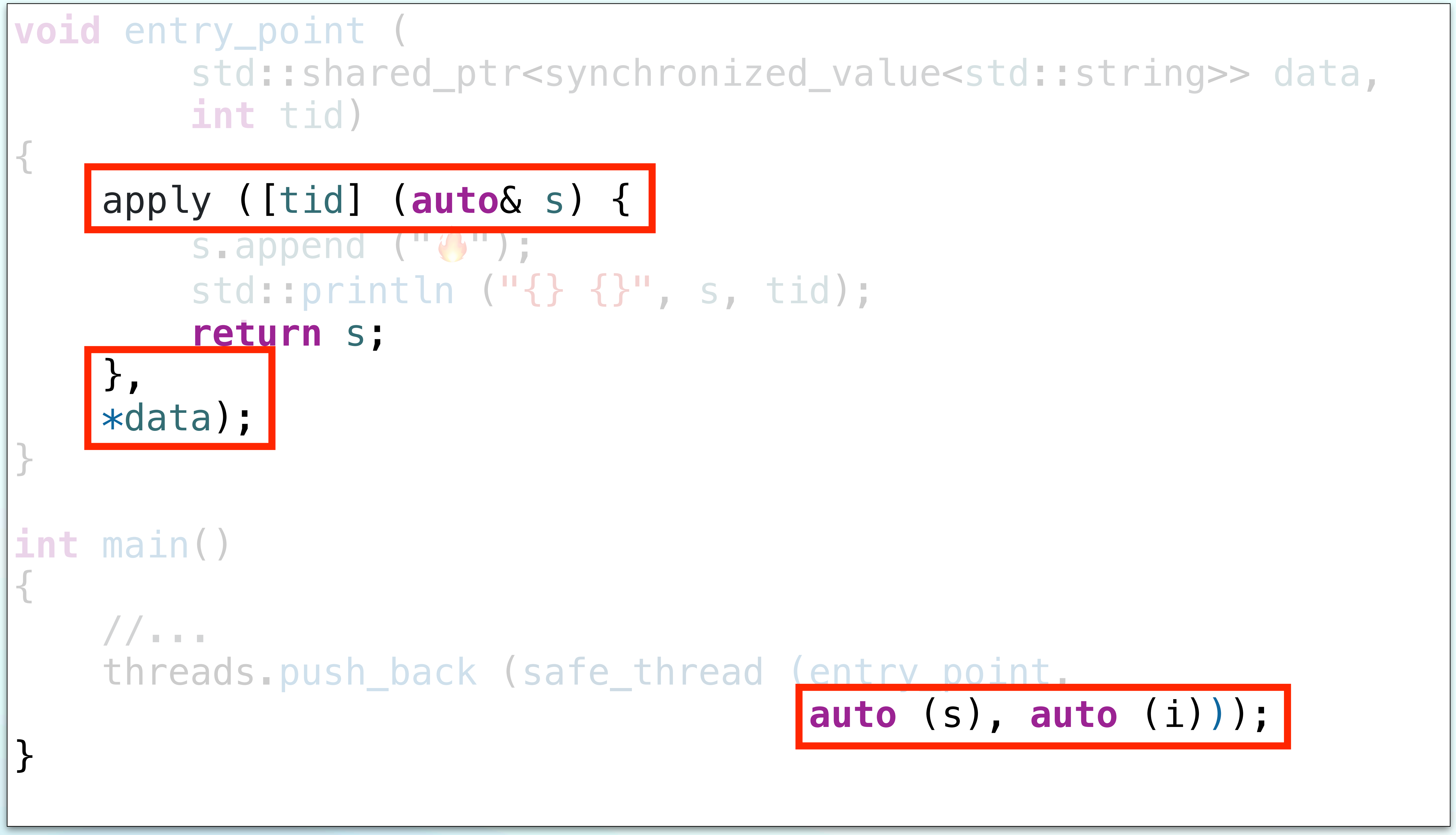
- Used an unenforceable `safe_thread` class
- Used a non-standard `syncronized_value` class
  - Had to add our own type trait for it
- Did a lot of fighting with the compiler
  - Template instantiation
  - Similar to “fighting the borrow checker”?
- Added a lot of overhead to our code
  - Atomic reference counting
  - Mutex locking

How far have we got in C++?

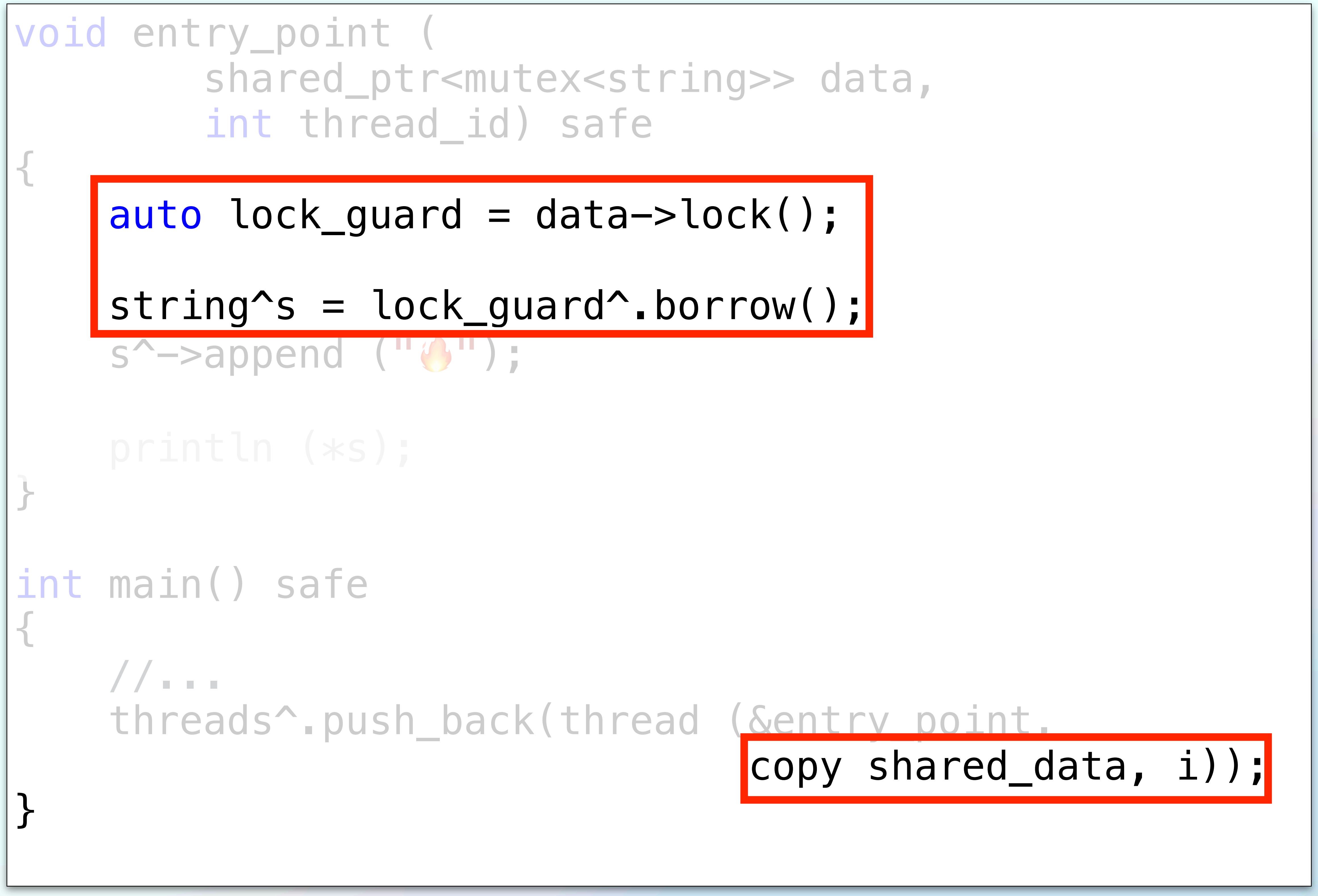
- Not *bullet proof*
- Not *beginner friendly*
- Not *default*

```
void entry_point (std::shared_ptr<synchronized_value<std::string>> sync_s, int tid)
{
    apply ([tid] (auto& s) {
        s.append ("🔥");
        std::println ("{} {}", s, tid);
        return s;
    },
    *sync_s);
}

int main()
{
    auto s = std::make_shared<synchronized_value<std::string>> ("Hello threads");

    std::vector<safe_thread> threads { };

    const int num_threads = 15;

    for (int i : std::views::iota (0, num_threads))
        threads.push_back (safe_thread (entry_point, auto (s), auto (i)));
}
```

```
void entry_point (shared_ptr<mutex<string>> data, int thread_id) safe
{
    auto lock _guard = data->lock();

    string^s = lock_guard^.borrow();
    s^->append ("🔥");

    println (*s);
}

int main () safe
{
    auto shared_data = shared_ptr<mutex<string>>::make(string ("Hello threads"));

    vector<thread> threads { };

    const int num threads = 15;

    for(int i : num _threads)
        threads^. push_back(thread (&entry_point, copy shared_data, i));
}
```



```
void entry_point (
        std::shared_ptr<synchronized_value<std::string>> data,
        int tid)
{
    apply ([tid] (auto& s) {
        s.append ("🔥");
        std::println ("{} {}", s, tid);
        return s;
    },
    *data);
}

int main()
{
    //...
    threads.push_back (safe_thread (entry_point,
                                    auto (s), auto (i)));
}
```



```
void entry_point (
        shared_ptr<mutex<string>> data,
        int thread_id) safe
{
    auto lock_guard = data->lock();

    string^s = lock_guard^.borrow();
    s^->append ("🔥");

    println (*s);
}

int main() safe
{
    //...
    threads^.push_back(thread (&entry_point,
                                            copy shared_data, i));
}
```

C++ Reflection to the Rescue?

- **Recursive Sync/Send Type Trait Checking**
  - Check members of types are all sendable
  - Check members of lambdas are all sendable

```
template<typename T>
struct Node
{
    Node* next;
    Node* prev;
};

std::shared_ptr<syncronized_value<Node>();
```

❌

```
auto node = std::make_shared<Node>();
safe_threads.emplace_back ([this, node]
                           {
                               memberFunction();
                           });
```

❌

Wrapping with Reflection

- **Value wrappers around shared objects**
  - `juce::Value`/`ValueTree`?
  - Copy-on-write objects
- **Thread-safe wrappers**
  - `synchronized_value`
  - `std::mutex/shared_mutex/spin_lock`
  - `crill::seqlock_object`
- **Async classes**

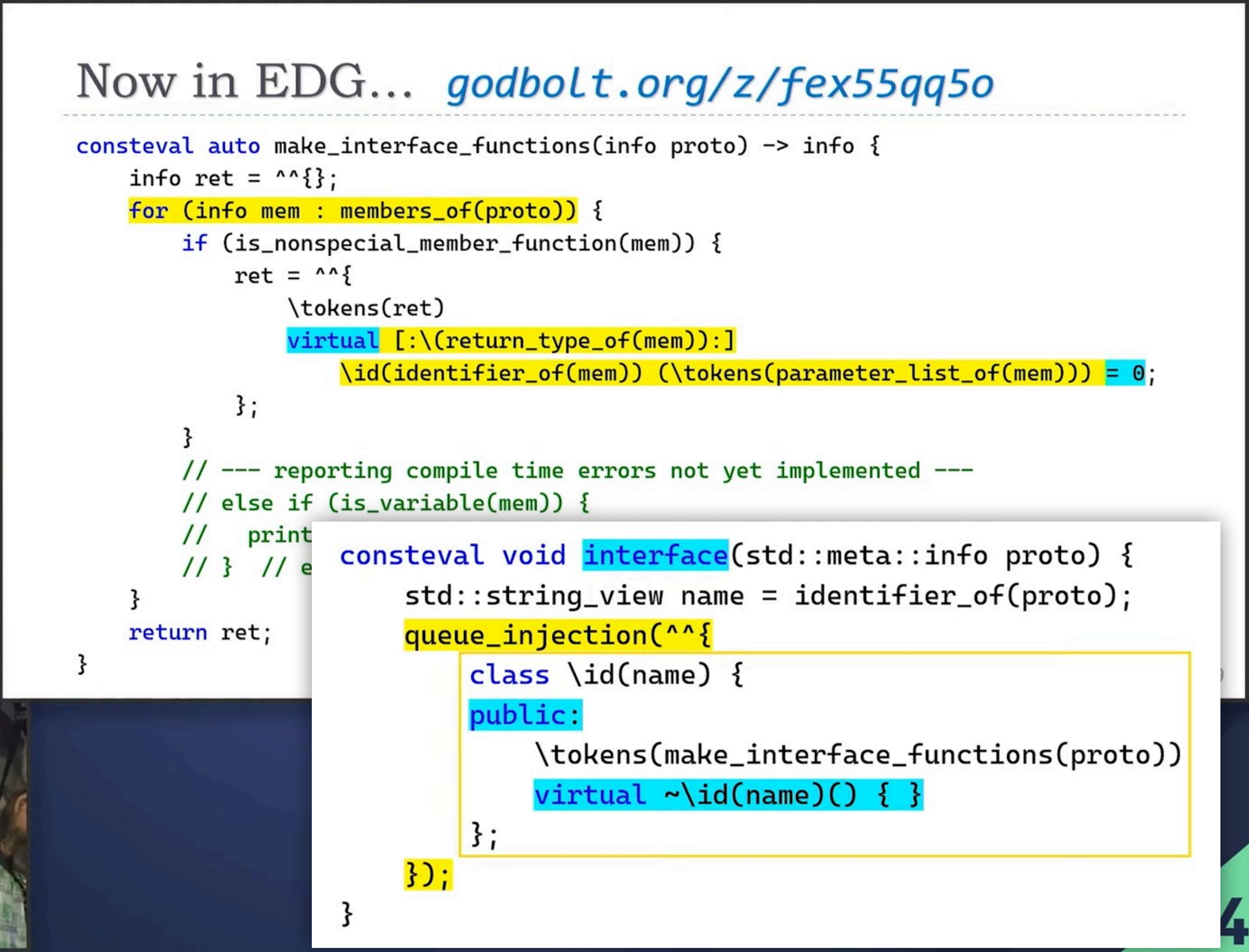
- P2996 - Reflection for C++26
  Accepted ✅
- P3294 - Code Injection with Token Sequences
  Hopeful for C++26 🔜
- P0707 - Metaclasses
  Proposed ⚠️

Implicit synchronized_value

metaclass proposed syntax

```
class person
{
public:
    person() = default;

    std::string get_first_name() const
    {
        return first_name;
    }

    void set_first_name (std::string_view new_first)
    {
        first_name = new_first;
    }

    // Repeat for last_name

private:
    std::string first_name, last_name;
};
```

```
class person
{
public:
    person() = default;

    std::string get_first_name() const
    {
       return apply ([] (auto& p) {
                         return p.get_first_name();
                     },
                     person_internal);
   }

    void set_first_name (std::string_view new_first)
    {
        apply ([&] (auto& p) {
                   p.set_first_name (new_first);
               },
               person_internal);
    }

    // Repeat for last_name

private:
    struct person_internal;
    mutable synchronized_value<person_internal> person_;
};
```



Now in EDG... *godbolt.org/z/fex55qq5o*

```
consteval auto make_interface_functions(info proto) -> info {
    info ret = ^^{};
    for (info mem : members_of(proto)) {
        if (is_nonspecial_member_function(mem)) {
            ret = ^^{
                \tokens(ret)
                virtual [:\(return_type_of(mem)):]
                    \id(identifier_of(mem)) (\tokens(parameter_list_of(mem))) = 0;
            };
        }
        // --- reporting compile time errors not yet implemented ---
        // else if (is_variable(mem)) {
        //   print
        // }  // e
    }
    return ret;
}
```

```
consteval void interface(std::meta::info proto) {
    std::string_view name = identifier_of(proto);
    queue_injection(^^{
        class \id(name) {
        public:
            \tokens(make_interface_functions(proto))
            virtual ~\id(name)() { }
        };
    });
}
```

Implicit `mutex` locking

```
class person(mutex)
{
public:
    person() = default;

    std::string get_first_name() const
    {
        return first_name;
    }

    void set_first_name (std::string_view new_first)
    {
        first_name = new_first;
    }

    // Repeat for last_name

private:
    std::string first_name, last_name;
};
```

```
class person
{
public:
    person() = default;

    std::string get_first_name() const
    {
        std::scoped_lock _ (mutex);
        return person_internal.get_first_name();
    }

    void set_first_name (std::string_view new_first)
    {
        std::scoped_lock _ (mutex);
        person_internal.set_first_name (new_first);
   }

    // Repeat for last_name

private:
    struct person_internal;
    std::mutex mutex;
    mutable person_internal person_internal;
};

template<>
struct is_sync<person> : std::true_type {};
```

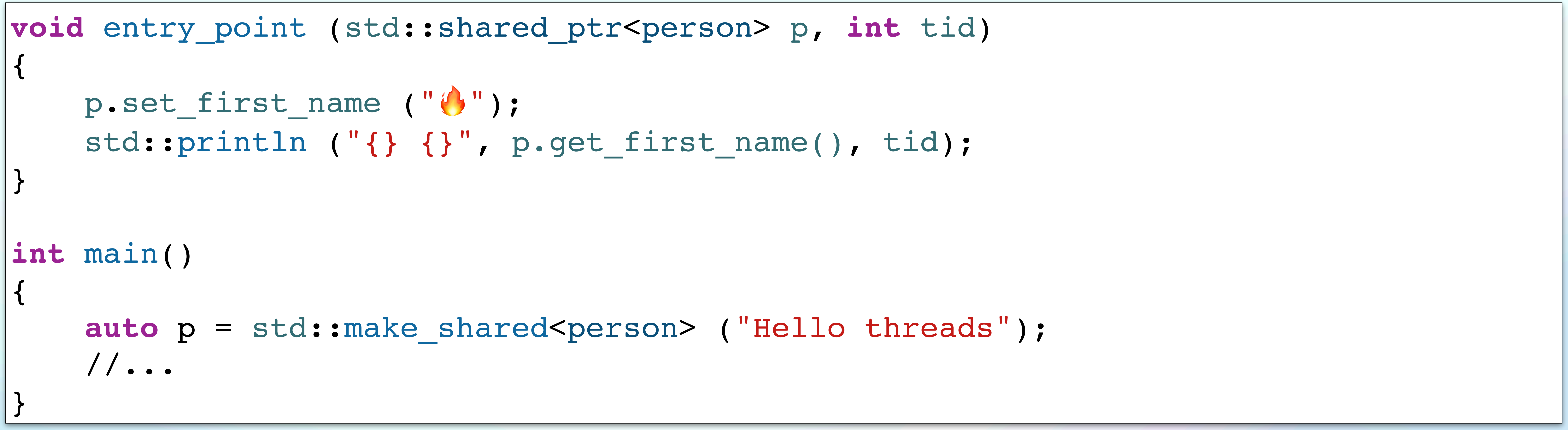
Implicit shared_mutex locking

```
class person(shared_mutex)
{
public:
    person() = default;

    std::string get_first_name() const
    {
        return first_name;
    }

    void set_first_name (std::string_view new_first)
    {
        first_name = new_first;
    }

    // Repeat for last_name

private:
    std::string first_name, last_name;
};
```

```
class person
{
public:
    person() = default;

    std::string get_first_name() const
    {
        std::shared_lock _ (mutex);
        return person_internal.get_first_name();
    }

    void set_first_name (std::string_view new_first)
    {
        std::unique_lock _ (mutex);
        person_internal.set_first_name (new_first);
   }

    // Repeat for last_name

private:
    struct person_internal;
    std::shared_mutex mutex;
    mutable person_internal person_internal;
};

template<>
struct is_sync<person> : std::true_type {};
```

```cpp
void entry_point (std::shared_ptr<synchronized_value<std::string>> sync_s, int tid)
{
    apply ([tid] (auto& s) {
        s.append ("🔥");
        std::println ("{} {}", s, tid);
        return s;
    },
    *sync_s);
}

int main()
{
    auto p = std::make_shared<synchronized_value<std::string>> ("Hello threads");
    //...
}
```

```
void entry_point (std::shared_ptr<person> p, int tid)
{
    apply ([tid] (auto& s) {
        s.append ("🔥");
        std::println ("{} {}", s, tid);
        return s;
    },
    *sync_s);
}

int main()
{
    auto p = std::make_shared<person> ("Hello threads");
    //...
}
```



```cpp
void entry_point (std::shared_ptr<person> p, int tid)
{
    p.set_first_name ("🔥");
    std::println ("{} {}", p.get_first_name(), tid);
}

int main()
{
    auto p = std::make_shared<person> ("Hello threads");
    //...
}
```

```cpp
void entry_point (person p, int tid)
{
    p.set_first_name ("🔥");
    std::println ("{} {}", p.get_first_name(), tid);
}

int main()
{
    auto p = person ("Hello threads");
    //...
}
```

Sync & Send

Low-level

Actors

High-level

Actors

High-level

Actors

High-level

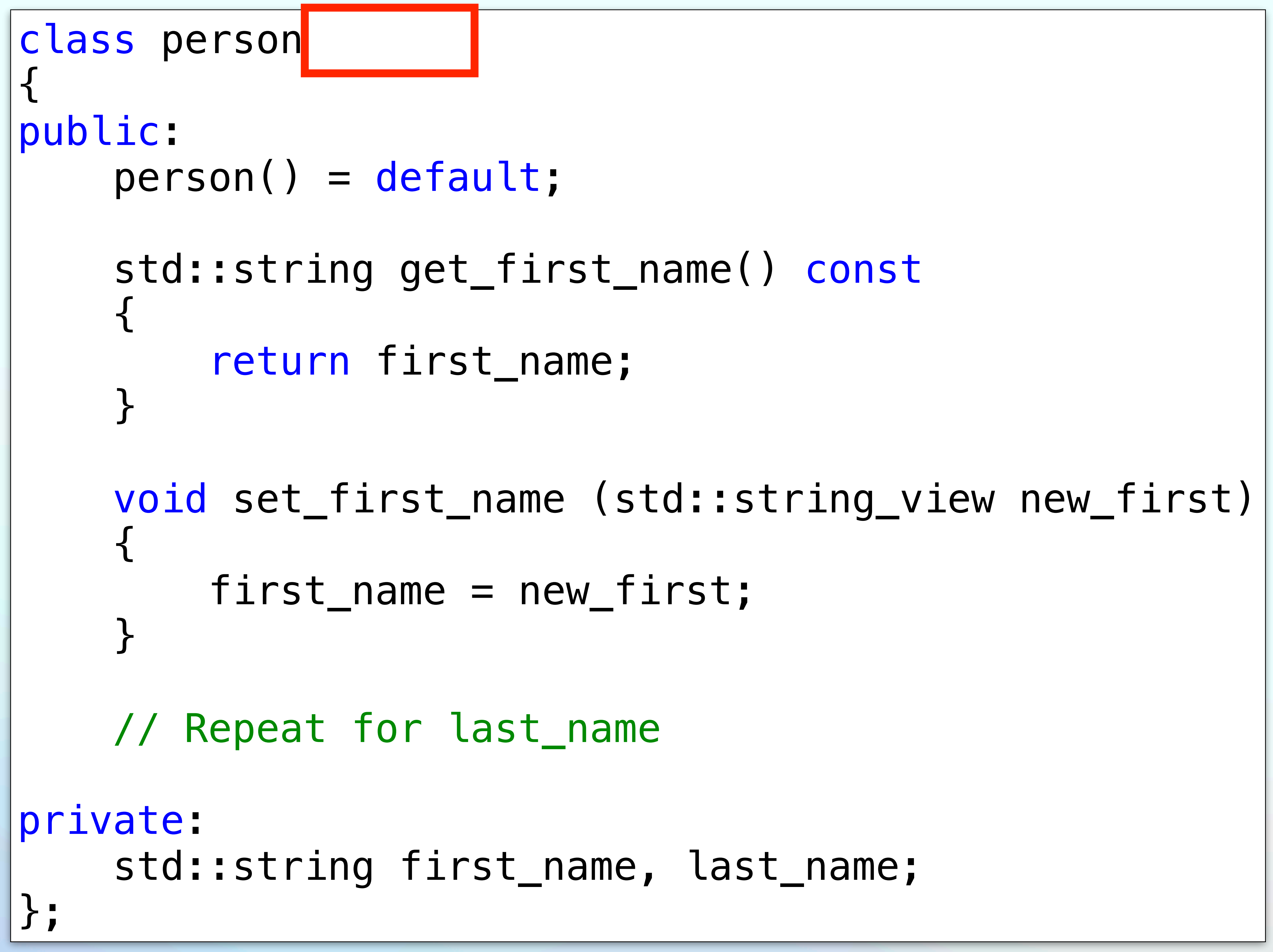
Swift Actors

```
actor Person
{
    private var first_name: String = "";

    func set_first_name (new_first: String)
    {
        first_name = new_first;
    }

    func get_first_name() -> String
    {
        return first_name
    }
}
```

```
var p = Person();

await p.set_first_name (new_first: "Dave")
print (await p.get_first_name())
```

Actors

metaclass proposed syntax



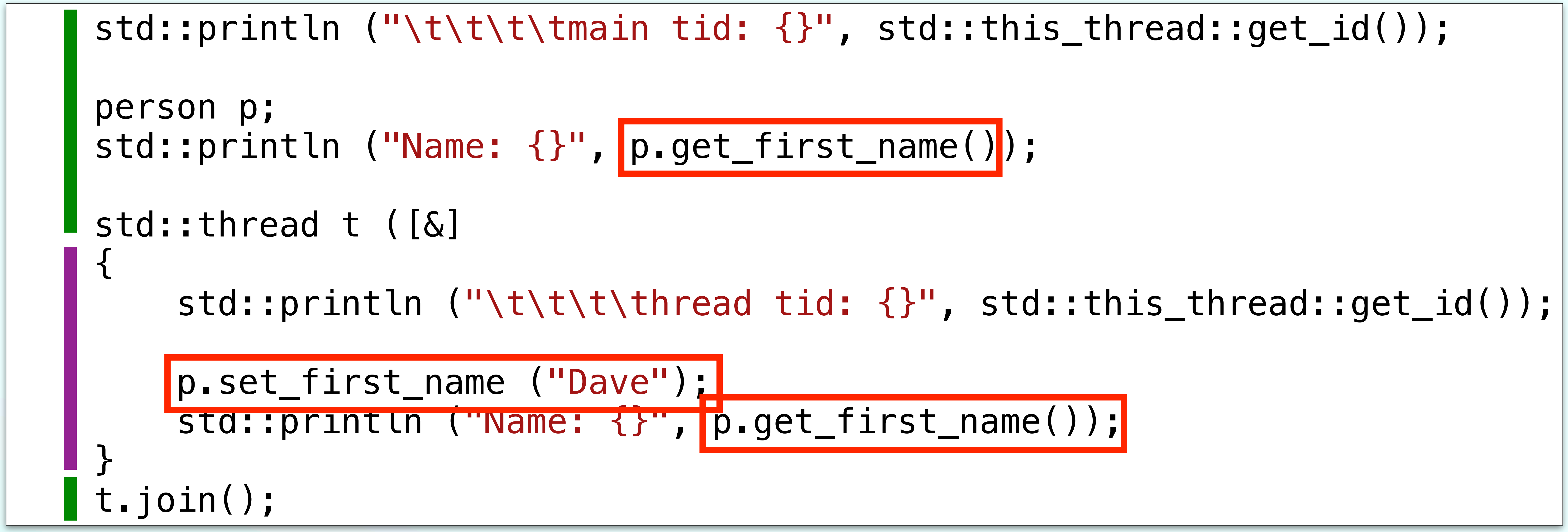
```
class person
{
public:
    person() = default;

    std::string get_first_name() const
    {
        return first_name;
    }

    void set_first_name (std::string_view new_first)
    {
        first_name = new_first;
    }

    // Repeat for last_name

private:
    std::string first_name, last_name;
};
```

Actors

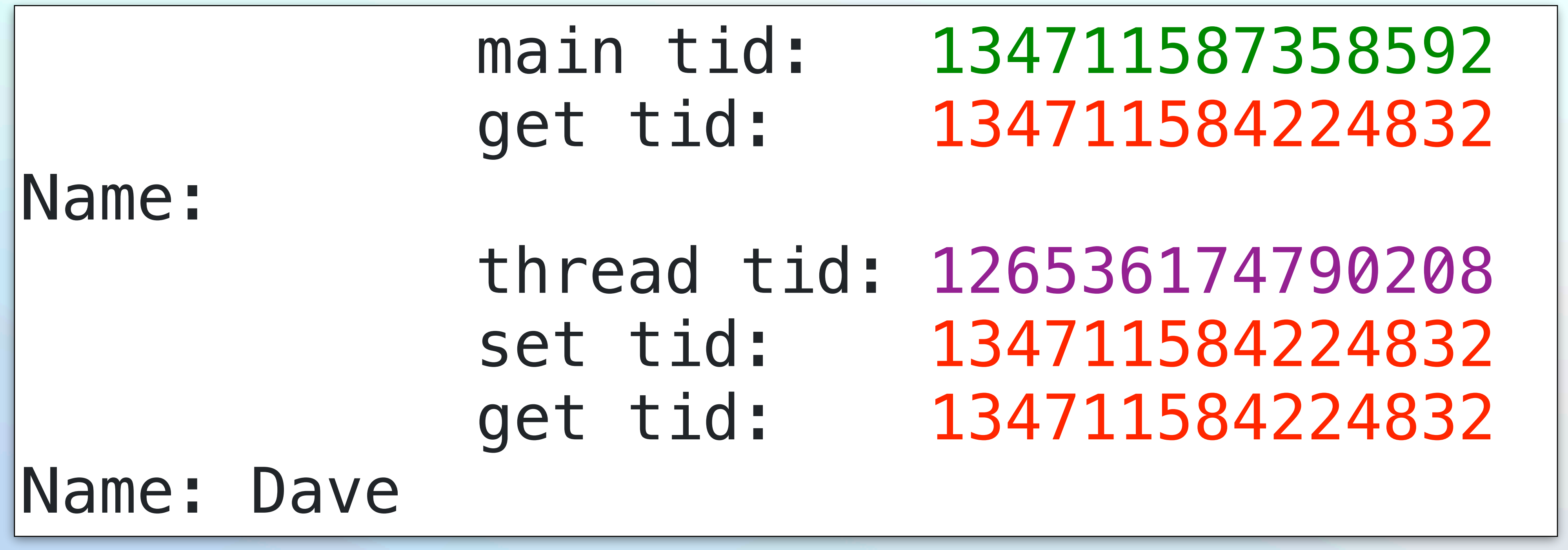
```
auto get_scheduler()
{
    static exec::static_thread_pool pool(1);
    return pool.get_scheduler();
}
```

```
class person
{
public:
    std::string get_first_name() const







    void set_first_name (std::string new_first)






private:
    mutable person_internal person;
};
```

Actors



```
std::println ("\t\t\t\tmain tid: {}", std::this_thread::get_id());

person p;
std::println ("Name: {}", p.get_first_name());

std::thread t ([&]
{
    std::println ("\t\t\t\tthread tid: {}", std::this_thread::get_id());

    p.set_first_name ("Dave");
    std::println ("Name: {}", p.get_first_name());
}
t.join();
```



```
             main tid:   134711587358592
             get tid:    134711584224832
Name:
             thread tid: 126536174790208
             set tid:    134711584224832
             get tid:    134711584224832
Name: Dave
```

Actors

```
std::string get_first_name() const
{
    auto sender = stdexec::then (stdexec::schedule (get_scheduler()),
                                 [this] { return person.get_first_name(); });
    auto [ret] = stdexec::sync_wait (sender).value();
    return ret;
}
```

Actors as co-routines

```
exec::task<std::string> get_first_name() const
{
    auto sender = stdexec::then (stdexec::schedule (get_scheduler()),
                                 [this] { return person.get_first_name(); });
    co_return co_await sender;
}
```

```
std::string first_name = co_await person.get_first_name();
```

Actors as co-routines

```
exec::task<std::string> get_first_name() const
{
    co_return co_await stdexec::then (stdexec::schedule (get_scheduler()),
                                      [this] { return person.get_first_name(); });
}
```

```
std::string first_name = co_await person.get_first_name();
```

Actors as co-routines

```
exec::task<std::string> get_first_name() const
{

                                      [this] { return person.get_first_name(); });
}
```

```
exec::task<void> set_first_name (std::string new_first)
{

                                      [this, =]
                                      { return person.set_first_name (new_first); });
}
```

```
actor Person
{
    private var first_name: String = "";

    func set_first_name (n: String) {
       first_name = n;
    }

    func get_first_name() -> String {
        return first_name
    }
}
```

```
struct person(actor)
{
    std::string get_first_name() const {
        return first_name;
    }

    void set_first_name (std::string_view n) {
        first_name = n;
    }

private:
    std::string first_name;
};
```

```
var p = Person();

await p.set_first_name (new_first: "Dave")
print (await p.get_first_name())
```

```
person p;

co_await p.set_first_name ("Dave");
std::print (co_await p.get_first_name());
```

	Cmajor/JS	Swift	C++23	Circle	cpp2	iso c++
Type	Static/dynamic type system	Static type system	Static type system reinterpret_cast	Static type system	Static type system	Profile: Type*
Bounds	Enforced/checked	Checked	Asan	Checked	Checked	Profile: Ranges, Algorithms & Pointers
Lifetime	Static/ ref-counted	Value semantics & Ref-counted	Partially enforced/ Asan	Enforced borrow checker	Partially enforced/ checked	Profile: RAII
Initialisation	Default initialised	Enforced	MSan/Asan	Enforced	Enforced	Profile: Initialisation
Arithmetic	ID/defined	Trap/explicit behaviour	UBsan	Checked/ defined	Checked	Profile: Arithmetic
Thread	Single* threaded	Enforced actors & sendable	Tsan	Enforced sync/send & BC	Tsan	Tsan

	Swift	C++23	Circle	cpp2	iso c++
Type	Static type system	Static type system reinterpret_cast	Static type system	Static type system	Profile: Type*
Bounds	Checked	Asan	Checked	Checked	Profile: Ranges, Algorithms & Pointers
Lifetime	Value semantics & Ref-counted	Partially enforced/ Asan	Enforced borrow checker	Partially enforced/ checked	Profile: RAII
Initialisation	Enforced	MSan/Asan	Enforced	Enforced	Profile: Initialisation
Arithmetic	Trap/explicit behaviour	UBsan	Checked/ defined	Checked	Profile: Arithmetic
Thread	Enforced actors & sendable	Tsan	Enforced sync/send & BC	Tsan	Tsan

	C++23	Circle	cpp2	iso c++
Type	Static type system reinterpret_cast	Static type system	Static type system	Profile: Type*
Bounds	Asan	Checked	Checked	Profile: Ranges, Algorithms & Pointers
Lifetime	Partially enforced/ Asan	Enforced borrow checker	Partially enforced/ checked	Profile: RAII
Initialisation	MSan/Asan	Enforced	Enforced	Profile: Initialisation
Arithmetic	UBsan	Checked/ defined	Checked	Profile: Arithmetic
Thread	Tsan	Enforced sync/send & BC	Tsan	Tsan

	C++23	cpp2	iso c++	Circle
Type	Static type system reinterpret_cast	Static type system	Profile: Type*	Static type system
Bounds	Asan	Checked	Profile: Ranges, Algorithms & Pointers	Checked
Lifetime	Partially enforced/ Asan	Partially enforced/ checked	Profile: RAII	Enforced borrow checker
Initialisation	MSan/Asan	Enforced	Profile: Initialisation	Enforced
Arithmetic	UBsan	Checked	Profile: Arithmetic	Checked/ defined
Thread	Tsan	Tsan	Tsan	Enforced sync/send & BC

	C++23	cpp2	iso c++	Circle
Type	Static type system reinterpret_cast	Static type system	Profile: Type*	Static type system
Bounds	Asan	Checked	Profile: Ranges, Algorithms & Pointers	Checked
Lifetime	Partially enforced/Asan	Partially enforced/ checked	Profile: RAII	Enforced borrow checker
Initialisation	MSan/Asan	Enforced	Profile: Initialisation	Enforced
Arithmetic	UBsan	Checked	Profile: Arithmetic	Checked/defined
Thread	Tsan	Tsan	Tsan	Enforced sync/send & BC

	C++23	cpp2	iso c++	Circle
Type	Static type system reinterpret_cast	Static type system	Profile: Type*	Static type system
Bounds	Asan	Checked	Profile: Ranges, Algorithms & Pointers	Checked
Lifetime	Partially enforced/Asan	Partially enforced/ checked	Profile: RAII	Enforced borrow checker
Initialisation	MSan/Asan	Enforced	Profile: Initialisation	Enforced
Arithmetic	UBsan	Checked	Profile: Arithmetic	Checked/defined
Thread	Tsan	Tsan	Enforced* sync/send & meta	Enforced sync/send & BC

Conclusion

- Use Cmajor/JS if appropriate
  - Minimise unsafe surface area
- For C++ code ->
  - Help is coming (C++26/29?) 🙏
- Start using cpp2
  - Safer defaults
  - Keep C++ code
- Eventually borrow checked C++?
  - wg21.link/P3390

Code style	Modern/safer libraries
clang-tidy	`cppcoreguidelines-pro-*`
Compiler warnings	`-Wall/extra/pedantic`
Compiler flags	-ftrapv `_LIBCPP_HARDENING_MODE_DEBUG/FAST=1`
Debugging	Asan/UBsan
Tests	Tsan
CI	Static analyser

Can Audio Programming be Safe?

David Rowland

𝕏 *@drowaudio*

Questions?

Slides/video:

drowaudio.github.io/presentations