# C++ONLINE

DAVE ROWLAND

TALK:

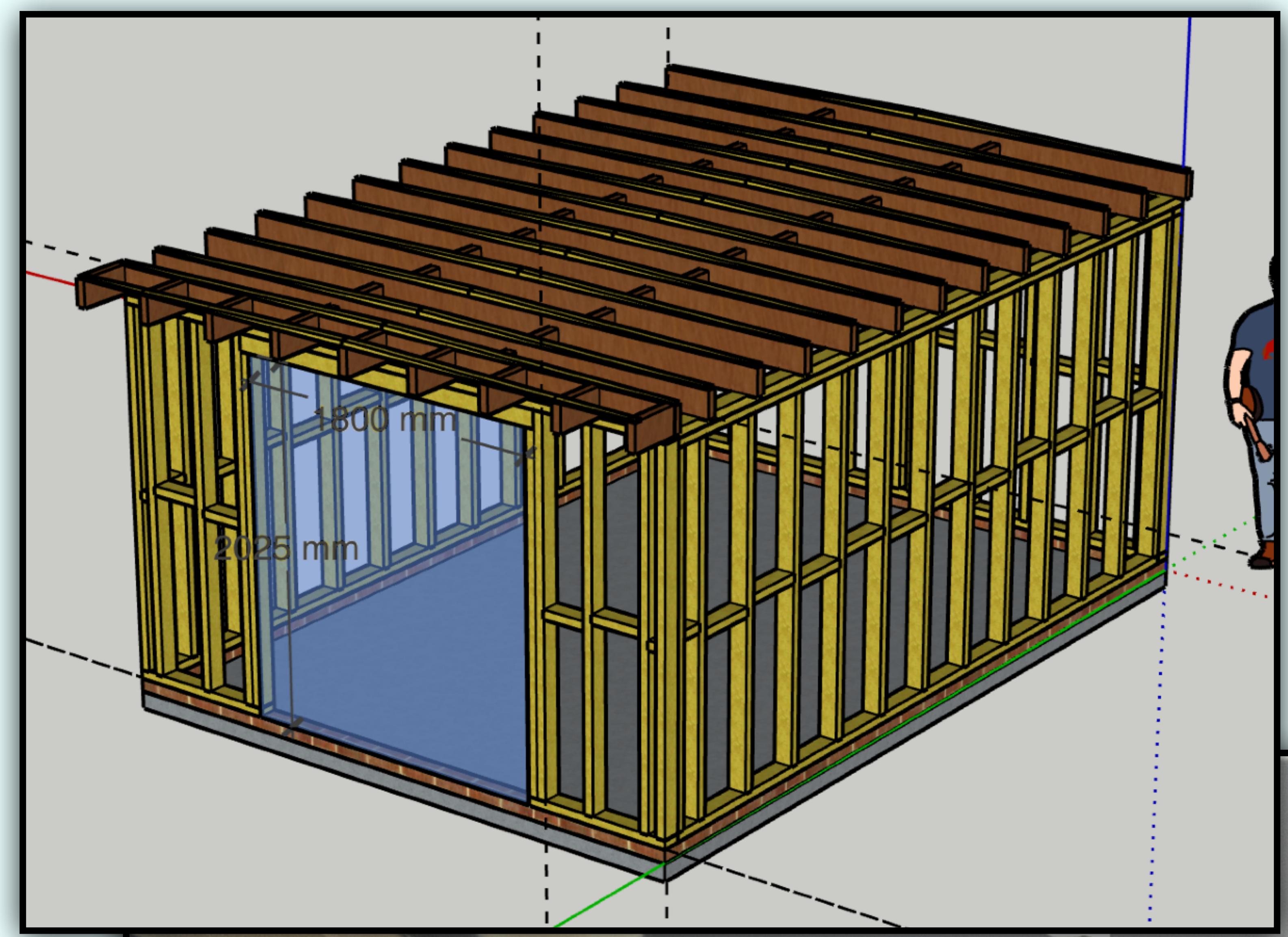
# WHAT CAN C++ LEARN ABOUT THREAD SAFETY FROM OTHER LANGUAGES?

2025

PRISM SOUND

tracktion


POWERED BY tracktion ENGINE


2025 mm


EcoTherm


JUTA
Breather Membranes
MASTER
BS5534



60

# 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*

# 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

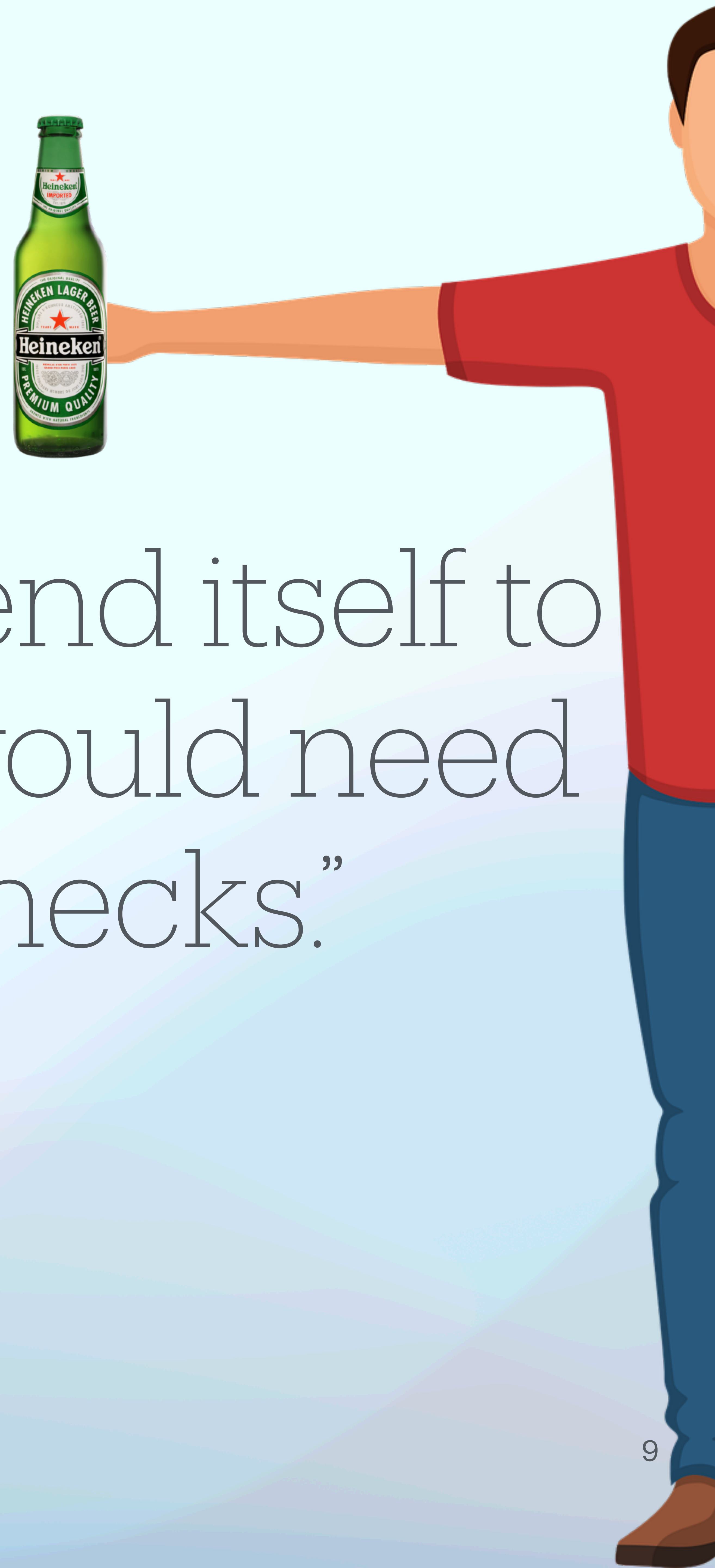
If we accept that we can modify the language to make HasMut<T> and HasRef<T> non-destructible, and to enforce they are not used after a move, then we might consider to go a step further and do away with these troublesome types.

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."

# 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.

# Thread safety:

# *Will it C++?*

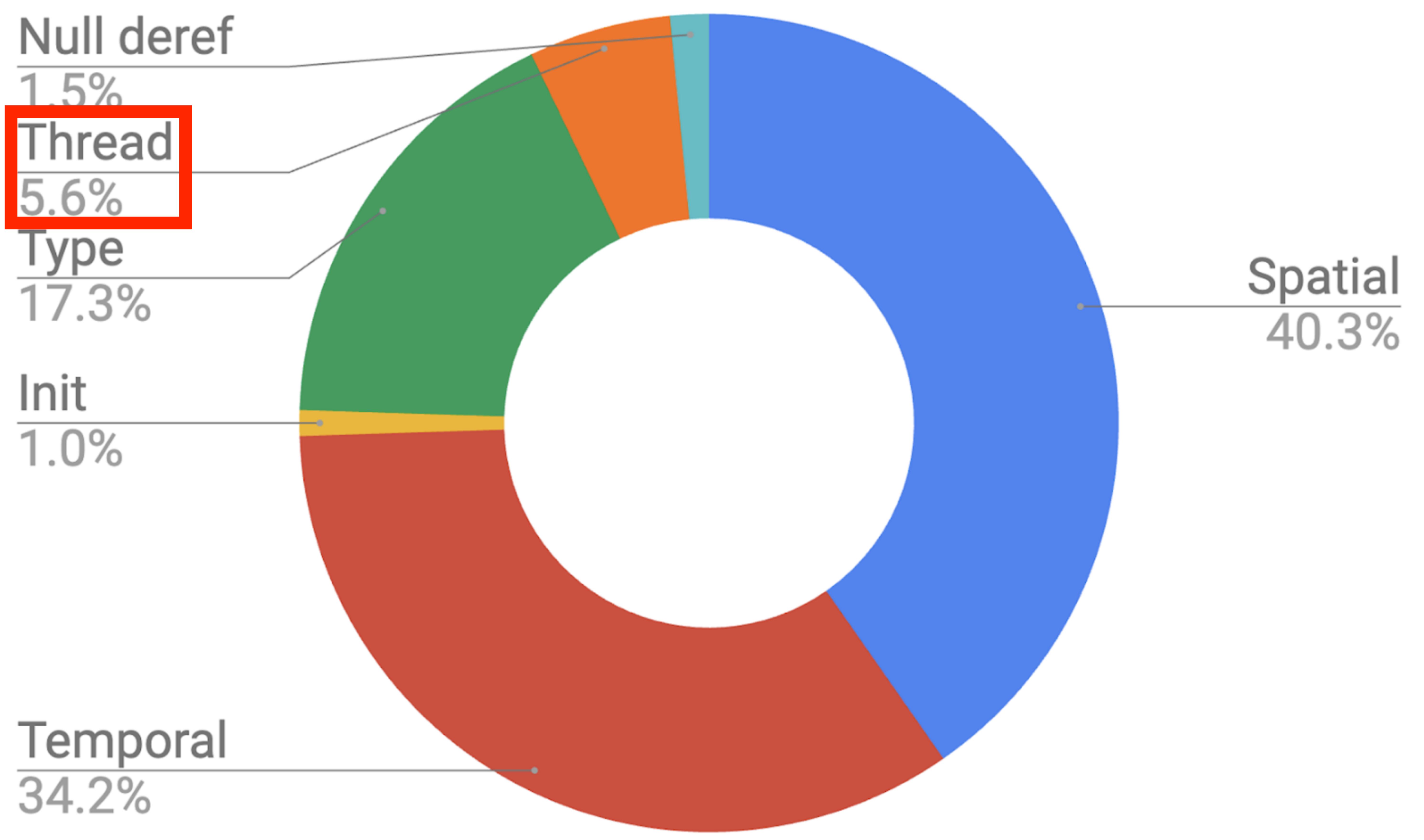
# What is Thread Safety?

# What is Thread Safety?

- A program is thread safe if it is free from data races
  - *And dead-locks*
- A data race is when two threads access the same memory location when at least one of them is a write
- A thread safe programming language makes it impossible to express data races

# Why Thread Safety?

# Why Thread Safety?




*Breakdown of memory safety CVEs exploited in the wild by vulnerability class.*[1]

Source: Retrofitting spatial safety to hundreds of millions of lines of C++
https://security.googleblog.com/2024/11/retrofitting-spatial-safety-to-hundreds.html

# Why Thread Safety?

- **5.6%** seems small but will grow
  - As "low hanging fruit" memory safety improves
  - As machines gain more cores
  - As multi-threading becomes easier and more ubiquitous e.g. `std::execution`
  - We will see lifetime/temporal safety is inextricably to thread safety

# Why Thread Safety?

- Bugs are difficult to spot and difficult to debug
- Problems typically arise long after a data race occurs
- Even if only **5.6%** of bugs are thread related, the time spent fixing them is likely much higher

# Why Thread Safety?

- **5.6%** is in Google's research
- In some industries that are inherently real-time (like audio) this is likely to be much higher

# Landscape of Approaches

Rust

ERLANG

Haskell

Circle

Carbon

hylo

Swift

Elixer

**Immutability**
**(Pure value semantics)**

**Mutable value semantics**
**(Local reasoning)**

**Message passing**
**(No shared state)**

# Focus On

Rust

Circle

Swift

## Sync & Send

Low-level

## Actors

High-level

# Sync & Send

- Protocols/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 Rust

- `send` is a "marker trait"
  - Similar to a C++ "type trait"
- Inferred if:
  - A copy can be made (value semantics)
  - A borrow can shared (`T&`)
  - **NOT** mutable borrow (`mut T&`)

# Sync & Send in Circle

- `send` is a "marker interface"
  - 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

```
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$
```

~/projects/circle4/talk/thread1.cxx (talk) - Sublime

File Edit Selection Find View Goto Tools Project Prefer

match1.cxx × match2.cxx × match3.cxx × std2.h ×

```
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 }
```

Line 25, Column 3    lifetime (22960)    Spaces: 2    C++

# 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:
113
How far have we got in C++?
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
122
Can Audio Programming be Safe?
David Rowland
Questions?
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)...)
    {
        // N.B. We can't constrain F to the concept due to recursion of is_move_constructable
        // So we have to statically assert it
        static_assert (send<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)...)
{
    static_assert (send<F>);
}
```

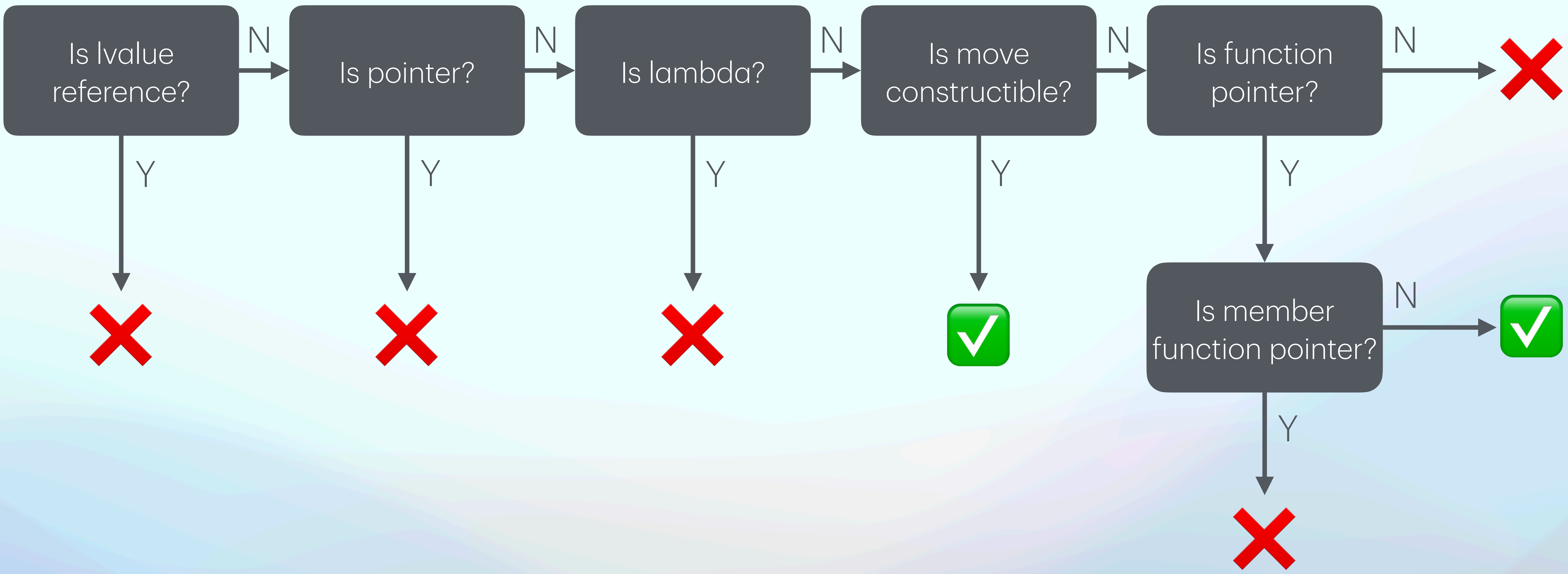
```
template<typename T>
struct is_send : std::bool_constant<
        (! (std::is_lvalue_reference_v<T>
            || std::is_pointer_v<std::remove_extent_t<T>>
            || is_lambda_v<T>))
         &&
           (std::is_move_constructible_v<T>
            || (is_function_pointer_v<std::decay_t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

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

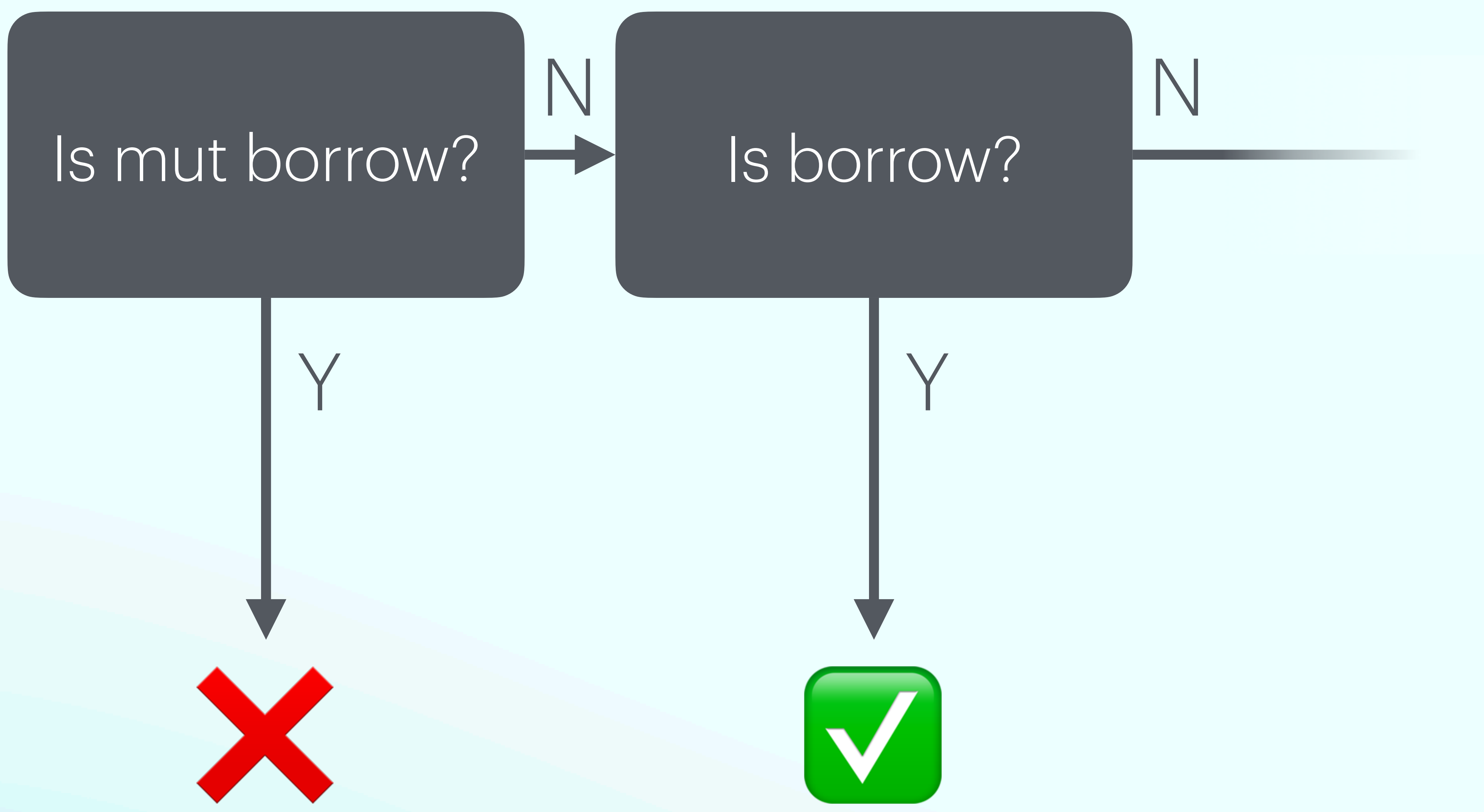
```
static_assert(is_send_v<const int>);
static_assert(is_send_v<int>);
static_assert(is_send_v<int&&>);
static_assert(is_send_v<int>);

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*&>);
```

# Is T send?


Is lvalue reference?
N
Y
Is pointer?
N
Y
Is lambda?
N
Y
Is move constructible?
N
Y
Is function pointer?
N
Y
Is member function pointer?
N
Y

# Is T send?



# 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>>);
```

# What types are sync?

- **`std::`**
  - `std::atomic`
    - *Trivial types only*
- **`synchronized_value (P0290)`**
  - Wraps a type with a `std::mutex`
  - Automatically locks during access
  - *Works with any type*

# synchronized_value

```
template<typename Type>
class synchronized_value
{
public:
    synchronized_value(const synchronized_value&) = delete;
    synchronized_value &operator=(const synchronized_value&) = delete;

    template<typename... Args>
    synchronized_value(Args&&... args)
        : val (std::forward<Args> (args)...)
    {}

    template<typename Fn, typename Up, typename... Types>
    friend std::invoke_result_t<Fn, Up&, Types&...> apply (Fn&&, synchronized_value<Up>&,
                                                           synchronized_value<Types>&...);

private:
    std::mutex mutex;
    Type val;
};
```

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

```
template <typename T>
struct is_send : std::bool_constant<
                    (! (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
{};
```

- Good

```
std::shared_ptr<std::atomic<int>> ✅
std::shared_ptr<synchronized_value<std::string>> ✅
```

- Bad

```
std::shared_ptr<int> ❌
std::shared_ptr<std::string> ❌
```

```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: Nested Pointers

```
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: this Pointers

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

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

# Problems: Global Pointers

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

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

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

# Problems: Leaked Pointers

```
auto widget = std::make_unique<Widget> (args);
auto widget_ptr = widget.get();
threads.push_back (safe_thread (entry_point, std::move (widget)));

widget_ptr->do_stuff();
```

# Problems: Summary

- Nested pointers
- `this` pointers
- Global pointers
- Leaked pointers

# 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 `synchronized_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*

```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)));
}
```

```
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));
}
```

```cpp
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)));
}
```

```cpp
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));
}
```

# Same example in Rust

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn entry_point(data: Arc<Mutex<String>>, thread_id: i32) {
    let mut guard = data.lock().unwrap();
    guard.push_str("🔥");
    println!("Thread {}: {}", thread_id, *guard);
}

pub fn main() {
    let shared_data = Arc::new(Mutex::new(String::from("Hello threads")));

    let mut threads = Vec::new();

    const NUM_THREADS: i32 = 15;

    for i in 0..NUM_THREADS {
        // Clone the Arc for this thread
        let data_clone = Arc::clone(&shared_data);

        // Spawn the thread and store its handle
        let handle = thread::spawn(move || {
            entry_point(data_clone, i);
        });

        threads.push(handle);
    }

    for handle in threads {
        handle.join().unwrap();
    }
}
```

# Same example in Rust (with borrows)

```rust
use std::sync::Mutex;
use std::thread;

fn entry_point(data: &Mutex<String>, thread_id: i32) {
    let mut guard = data.lock().unwrap();
    guard.push_str("🔥");
    println!("Thread {}: {}", thread_id, *guard);
}

pub fn main() {
   let shared_data = Mutex::new(String::from("Hello threads"));

    const NUM_THREADS: i32 = 15;

    // Use scope to ensure threads don't outlive our data
    thread::scope(|scope| {
        let mut threads = Vec::new();

        for i in 0..NUM_THREADS {
            let local_data = &shared_data;
            let handle = scope.spawn(move || {
                entry_point(local_data, i);
            });

            threads.push(handle);
        }

        for handle in threads {
            handle.join().unwrap();
        }
    });
}
```

Key changes made in this version:

1. Removed `Arc` and now using direct references (`&Mutex<String>`)
2. Added `thread::scope` to ensure threads don't outlive the borrowed data
3. Changed the thread spawning to use scoped threads via `scope.spawn`
4. Simplified the function signature of `entry_point` to take a reference
5. No more need for explicit cloning since we're using references

# Same example in Rust (with borrows)

```rust
use std::sync::Mutex;
use std::thread;

fn entry_point(data: &Mutex<String>, thread_id: i32) {
    let mut guard = data.lock().unwrap();
    guard.push_str("🔥");
    println!("Thread {}: {}", thread_id, *guard);
}

pub fn main() {
   let shared_data = Mutex::new(String::from("Hello threads"));

    const NUM_THREADS: i32 = 15;

    // Use scope to ensure threads don't outlive our data
    thread::scope(|scope| {
        let mut threads = Vec::new();

        for i in 0..NUM_THREADS {
            let local_data = &shared_data;
            let handle = scope.spawn(move || {
                entry_point(local_data, i);
            });

            threads.push(handle);
        }

        for handle in threads {
            handle.join().unwrap();
        }
    });
}
```

This version has several advantages:

- More efficient (no atomic reference counting)
- Cleaner code (no clone operations)
- Compile-time guarantees about data lifetime
- Still maintains thread safety through the `Mutex`

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

# Back to C++

```
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: Summary

- Nested pointers
- `this` pointers
- Global pointers
- Leaked pointers

# 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

```
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();
                            });
```

```
consteval auto is_send_type (std::meta::info type) -> bool
{
    type = remove_cv (type);

    // Non-member function pointers
    if (is_pointer_type (type)
        && is_function_type (remove_pointer (type))
        && ! is_member_function_pointer_type (type))
       return true;

    // lvalue refs and pointers
    if (is_lvalue_reference_type (type)
        || is_pointer_type (remove_extent (type)))
       return false;

    // POD built-in types
    if (is_arithmetic_type (type))
        return true;

    // Recursive class/struct/lambda members
    if (is_class_type (type))
        return std::ranges::all_of(nonstatic_data_members_of(type),
                                   [](std::meta::info d)
                                   {
                                       return is_send_type (type_of(d));
                                   });

    // Construct from rvalue ref
    if (is_rvalue_reference_type (type)
        && is_constructible_type (type, { remove_reference (type) }))
       return true;

    return false;
}
```

```
template<typename T>
inline constexpr bool is_send_v = is_send (^^T);

te
co
```

```
template<typename T>
consteval auto is_send() -> bool
{
    if (is_send_type (^^T))
       return true;

    return is_sync_v<T>;
}

template<typename T>
inline constexpr bool is_send_v = is_send<T>();

template<typename T>
concept send = is_send_v<T>;
```

```cpp
struct node
{
    node* prev;
    node* next;
};

static_assert(! is_send_v<node>);
```

```cpp
[[maybe_unused]] auto non_capturing = [] (int) {};
static_assert(is_send_v<decltype(non_capturing)>);

int i = 0;
[[maybe_unused]] auto val_capturing = [i] (int) {};
static_assert(is_send_v<decltype(val_capturing)>);

[[maybe_unused]] auto ref_capturing = [&i] (int) {};
static_assert(! is_send_v<decltype(ref_capturing)>);
```

```cpp
struct type
{
    type()
    {
        auto n = std::make_shared<node>();

        [[maybe_unused]] auto this_capturing = [this] { run(); };
        static_assert(! is_send_v<decltype(this_capturing)>);

        [[maybe_unused]] auto this_n_capturing = [this, n] { run(); };
        static_assert(! is_send_v<decltype(this_n_capturing)>);

        [[maybe_unused]] auto n_ref_capturing = [&n] {};
        static_assert(! is_send_v<decltype(n_ref_capturing)>);

        [[maybe_unused]] auto n_val_capturing = [n] {};
        static_assert(! is_send_v<decltype(n_val_capturing)>);
    }

    void run() {}
};
```

# Problems: Summary

- ~~`this` pointers~~
- Global pointers
- Leaked pointers

# Global pointers

```cpp
void set_global_string (std::string*);

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

```rust
fn entry_point(data: &Mutex<String>, thread_id: i32) {
    let mut guard = data.lock().unwrap();
    guard.push_str("🔥");
    println!("Thread {}: {}", thread_id, *guard);
}
```

# Wrapping with Reflection

- P2996 - Reflection for C++26
  *Accepted* ✅

- P3294 - Code Injection with Token Sequences
  *Hopeful for C++29* 😔

- 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;
};
```

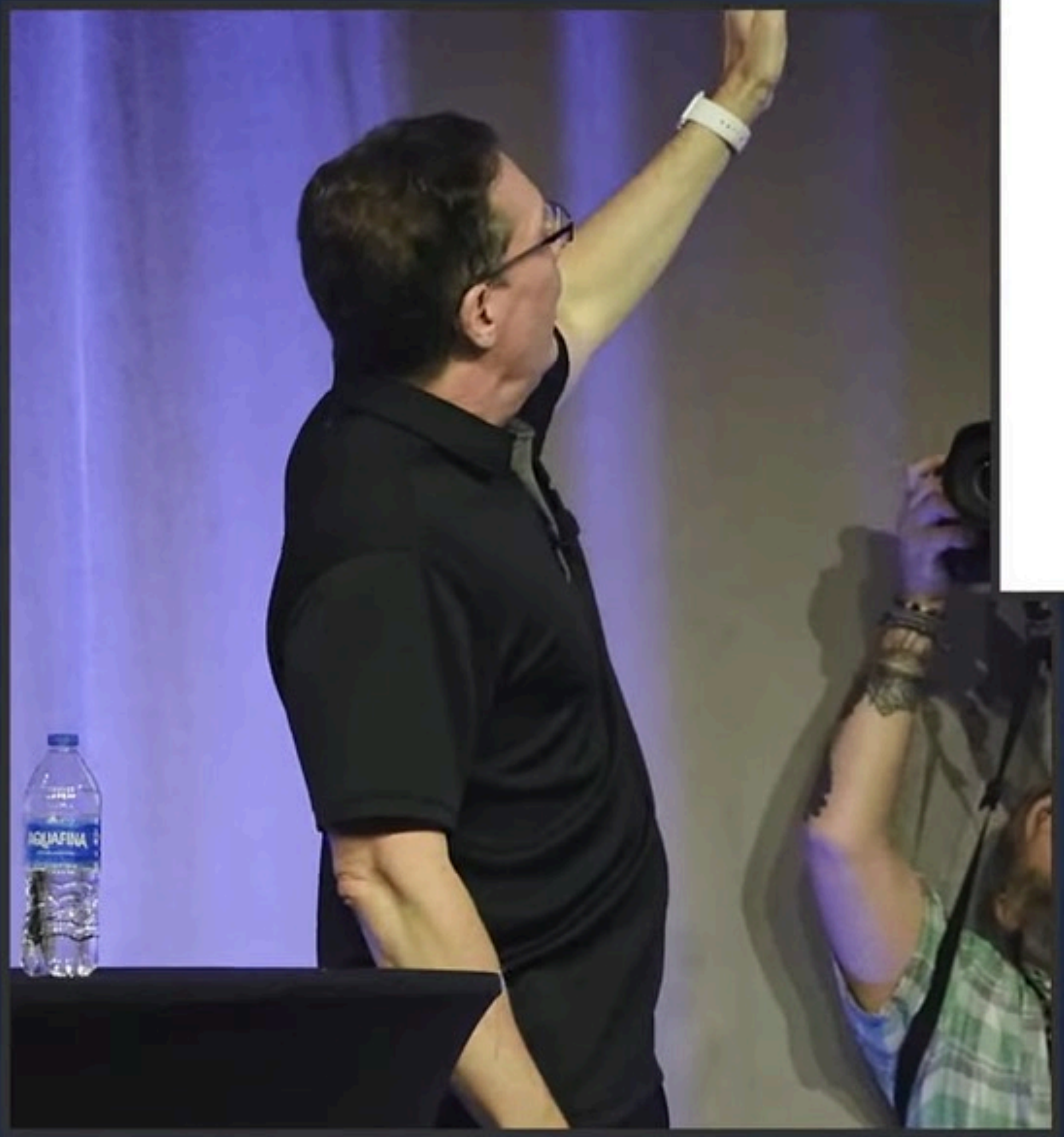
```
class person
{
public:
    person() = default;

    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;
    mutable synchronized_value<__person> 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_.get_first_name();
    }

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

    // Repeat for last_name

private:
    class __person;
    std::mutex mutex;
    mutable __person person_;
};

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_.get_first_name();
    }

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

    // Repeat for last_name

private:
    class __person;
    std::shared_mutex mutex;
    mutable __person person_;
};

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");
    //...
}
```

```cpp
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");
    //...
}
```

```
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");
    //...
}
```

# Problems: Leaked Pointers

```
auto widget = std::make_unique<Widget> (args);
auto widget_ptr = widget.get();
threads.push_back (safe_thread (entry_point, std::move (widget)));

widget_ptr->do_stuff();
```

```
void entry_point (std::shared_ptr<person> p, int tid)
{
    auto person_ptr = p.get();
}
```

# Wrapped std::shared_ptr

arc metaclass

```
class person(arc)
{
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 person_->get_first_name();
    }

    void set_first_name (std::string_view new_first)
    {
        person_->set_first_name (new_first);
    }

    // Repeat for last_name

private:
    class __person;
    std::shared_ptr<__person> person_;
};
```

# Look familiar? Swift `classes`

```swift
class Person
{
    private var first_name: String = "";
    private var last_name: String = "";

    func get_first_name() -> String
    {
        return first_name
    }

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

    // Repeat for last_name
}
```

```cpp
class person(arc)
{
public:
    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;
};
```

# Combined

arc & mutex metaclass

```
class person(mutex, arc)
{
public:
   //...
```

```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");
    //...
}
```

# Swift `class`: Breaking Cycles

- Cyclic references cause memory leaks
- References in Swift are **strong** by default
- To break a cycle **weak** references can be used
- These are `nil`ed when the last strong reference is destroyed
- Must be checked before dereferencing

```
var p = Person()
p.set_first_name (new_first: "Dave")
print (p.get_first_name())

weak var p2 = p
p2?.set_first_name (new_first: "John")
```

# Wrapped std::weak_ptr

```
class person
{
public:
    class weak_ref
    {
    public:
        weak_ref() = default;
        weak_ref (person p)
            : person_ (p.person_) {}

        std::optional<person> get() const
        {
            if (auto valid = person_.lock())
                return person (std::move (valid));

            return std::nullopt;
        }

    private:
        std::weak_ptr<__person> person_;
    };

    //... rest of class as before

private:
    person (std::shared_ptr<__person>&& other)
        : person_ (other) {}
};
```

```
person p1;
//... do stuff with p1

person::weak_ref p2; // create uninitialised
p2 = p1;             // assign from strong-ref

if (auto valid_person = p3.get())
    std::println ("p3 {}", valid_person->get_first_name());

p2.get().transform ([] (auto valid_person) {
                        valid_person.set_first_name (“John");
                        return valid_person;
                    });
```

# Swift `structs`

- Value semantics
  - Two instances of a `struct` have distinct objects
  - Are `@Sendable` implicitly if all members are `@Sendable`
  - Scalar (pod etc.) and self-contained objects (like `String`) are `@Sendable`
- Can be implemented with copy-on-write for efficiency

# Swift structs

```swift
struct Person
{
    private var first_name: String = "";
    private var last_name: String = "";

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

    func get_first_name() -> String
    {
        return first_name
    }

    // Repeat for last_name
}
```

```cpp
struct person
{
    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;
};
```

# Copy on Write

c.o.w. metaclass

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

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

    // Repeat for last_name

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

```
struct person
{
    std::string get_first_name() const {
        return person_->get_first_name();
    }

    void set_first_name (std::string_view new_first) {
        copy_if_shared();
        person_->set_first_name (new_first);
    }

    // Repeat for last_name

private:
    struct __person;
    static_assert (std::is_copy_constructible_v<__person>);

    std::shared_ptr<__person> person_
        = std::make_shared<__person>();

    void copy_if_shared() {
        if (person_.use_count() > 1)
            person_ = std::make_shared<__person> (*person_);
    }
};
```

# Copy on Write `structs`

```
struct person
{
    std::string get_first_name() const {
        return person_->get_first_name();
    }

    void set_first_name (std::string_view new_first) {
        copy_if_shared();
        person_->set_first_name (new_first);
    }

    // Repeat for last_name

private:
    struct __person;
    static_assert (std::is_copy_constructible_v<__person>);

    std::shared_ptr<__person> person_
        = std::make_shared<__person>();

    void copy_if_shared() {
        if (person_.use_count() > 1)
            person_ = std::make_shared<__person> (*person_);
    }
};
```

- Each `person` has its own `shared_ptr` instance
    - *This is never shared*
- As soon as a non-const function is called, a unique copy is made
    - The internal `__person` may be shared, but that's fine as there will only be *readers*

# Copy on Write `structs`

- Only works if there are no *pointers* or *references* to a `person`
  - `send` enforces this when passed to a thread
  - Delete `operator new` to avoid heap allocations
  - Delete `operator&` to avoid taking the address
- Doesn't stop references
- Doesn't stop references/pointers when used as a member in another object
  - Would require viral checking/static analysis

```
struct person
{
    //...
    // Wrapped __person functions
    //...
};
```

# Mutable Value Semantics

- Hylo's thread safety comes from avoiding shared state
- Objects are mutable within a function - local reasoning
- Similar to Swift with only `struct` types
- Implemented efficiently

# Review

- `send` trait introduces an "isolation boundary" between threads
  - Objects can only be ***copied*** or ***moved*** between them
- `sync` trait tells the compiler an object is data-race free
  - And is implicitly `send`
- These traits need to be checked recursively for all members
  - *C++23 can not do this*
  - C++26 reflection should enable this checking
- Lifetime safety is inherently intertwined with thread safety
  - *Solved in other languages with borrow checking or mutable value semantics*
- We need to encapsulate pointers in value types to ensure they're not exposed to abuse
  - C++26 Reflection generation (and future metaclasses) can make this simple

# Limitations

- Not the most efficient
  - E.g. `mutex` wraps the whole class, not individual members
- Can `arc`, `cow` or `mutex` metaclasses be inherited?
  - If any original functions were `virtual`, this would break protections e.g. `cow`, `mutex`
  - *Derived classes could possibly inherit the metaclasses?*
  - *Could only work on non-virtual classes*
  - *Could be disabled by adding* `final` *to the generated class*
- Very early days!
  - *Need implementation experience*

# Concerns

- Bakes data-race safety and lifetime management in to the type
  - May not be suitable for every use case
  - Could pay performance cost for simple, single thread uses
  - Not the most efficient (*borrow checking*)
  - Great success in existing languages e.g. Swift
- Not “C++”?
  - Contradicts “Don’t pay for what you don’t use”

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 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 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); });
}
```

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

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

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

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

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

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

```swift
var p = Person();

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

```cpp
person p;

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

# Actors: Problems

- Thread/Lifetime safety issues with function arguments
  - `assert` the arguments are `send`?
  - Reflect on the lambda type to ensure it's `send?`
  - Forward arguments like we did for `safe_thread`?

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

    co_return co_await stdexec::then (stdexec::schedule (get_scheduler()),
                                      [this, =]
                                      { return person.set_first_name (new_first); });
}
```

# Actors: Problems

- In practice may need different pools
  - Serialises all actors on to a single thread
- Could use "Annotations for Reflection" P3394
  - Different pool tags
  - Different schedular types

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

```
struct [[=LowPriority]] person(actor)
//…

struct LowPriority_tag;

template<typename PoolType>
auto get_scheduler()
{
    static exec::static_thread_pool pool(1);
    // init low-priority
    return pool.get_scheduler();
}
```

```
struct [[=MainActor]] person(actor)
//…

template<typename PoolType>
auto get_main_scheduler()
{
    static exec::run_loop loop {};
    // Needs to be dispatched by main thread
    return loop.get_scheduler();
}
```

# Actors: Problems

- Huge overhead to queue every operation on a thread
- Re-entrant functions should execute synchronously

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



   co_return co_await stdexec::then (stdexec::schedule (get_scheduler()),
                                     [this, =]
                                     { return person.set_first_name (new_first); });
}
```

# Run-time Data Race Detection

# Existing Strategy: TSan

- Only available in clang and gcc (no Visual Studio support)
- Requires separate running
- Mutually exclusive with other sanitisers (ASan, UBSan etc.)
- Only as good as test coverage
  - *Fuzzing can help*
- Extremely heavyweight
  - 5-15x slower execution
  - 5-10x increase in memory usage
  - 2-3x increase in binary size
  - Moderate increase in compilation time

# Lightweight Data Race Detection

| | No Readers No Writers | Active Reader | Active Writer |
|---|---|---|---|
| **Read Enter** | *No race* | *No race* | DATA RACE |
| **Write Enter** | *No race* | DATA RACE | DATA RACE |

# Lightweight Data Race Detection

```cpp
void read_started (check_state& state)
{
    ++state.num_readers; // must be first

    if (state.is_writing)
        std::terminate();
        // read during active write
}

void write_started (check_state& state)
{
    // must be first
    if (state.is_writing.exchange (true))
        std::terminate();
        // write during active write

    if (state.num_readers > 0)
        std::terminate();
        // write during active read
}
```

```cpp
struct check_state
{
    std::atomic<size_t> num_readers { 0 };
    std::atomic<bool> is_writing { false };
};
```

```cpp
void read_ended (check_state& state)
{
    --state.num_readers;
}

void write_ended (check_state& state)
{
    state.is_writing = false;
}
```

# Lightweight Data Race Detection

```
enum class check_type
{
    read,
    write
};

template<check_type type>
struct scoped_check
{
    scoped_check (check_state& check_state)
        : state (check_state)
    {
        if constexpr (type == check_type::read)
            read_started (state);
        else
            write_started (state);
    }

    ~scoped_check()
    {
        if constexpr (type == check_type::read)
            read_ended (state);
        else
            write_ended (state);
    }

    check_state& state;
};
```

# Wrapped Data Race Detection

data_race_checker metaclass

```cpp
struct person(data_race_checker)
{
    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;
};
```

```cpp
struct person
{
    std::string get_first_name() const
    {
        scoped_check<check_type::read> _ (check_state);
        return person_.get_first_name();
    }

    void set_first_name (std::string_view new_first)
    {
        scoped_check<check_type::write> _ (check_state);
        person_.set_first_name (new_first);
    }

    // Repeat for last_name

private:
    struct __person;
      person person_;
    mutable check_state check_state;
};
```

# std Containers

## 16 Library introduction

### 16.4 Library-wide requirements

#### 16.4.6 Conforming implementations

##### 16.4.6.10 Data race avoidance

1 This subclause specifies requirements that implementations shall meet to prevent data race
function shall meet each requirement unless otherwise specified. Implementations may pr
other than those specified below.

2 A C++ standard library function shall not directly or indirectly access objects ([intro.multithread]) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's arguments, including `this`.

3 A C++ standard library function shall not directly or indirectly modify objects ([intro.multithread]) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's non-const arguments, including `this`.

4 [*Note 1*: This means, for example, that implementations can't use an object with static storage duration for internal purposes without synchronization because doing so can cause a data race even in programs that do not explicitly share objects between threads. — *end note*]

5 A C++ standard library function shall not access objects indirectly accessible via its arguments or via elements of its container arguments except by invoking functions required by its specification on those container elements.

6 Operations on iterators obtained by calling a standard library container or string member function may access the underlying container, but shall not modify it.

[*Note 2*: In particular, container operations that invalidate iterators conflict with operations on iterators associated with that container. — *end note*]

7 Implementations may share their own internal objects between threads if the objects are not visible to users and are protected against data races.

8 Unless otherwise specified, C++ standard library functions shall perform all operations solely within the current thread if those operations have effects that are visible to users.

9 [*Note 3*: This allows implementations to parallelize operations if there are no visible side effects. — *end note*]

## 23 Containers library [containers]

### 23.2 Requirements [container.requirements]

#### 23.2.3 Container data races [container.requirements.dataraces]

1 For purposes of avoiding data races ([res.on.data.races]), implementations shall consider the following functions to be `const`: `begin`, `end`, `rbegin`, `rend`, `front`, `back`, `data`, `find`, `lower_bound`, `upper_bound`, `equal_range`, `at` and, except in associative or unordered associative containers, `operator[]`.

2 Notwithstanding [res.on.data.races], implementations are required to avoid data races when the contents of the contained object in different elements in the same container, excepting `vector<bool>`, are modified concurrently.

3 [*Note 1*: For a `vector<int> x` with a size greater than one, `x[1] = 5` and `*x.begin() = 10` can be executed concurrently without a data race, but `x[0] = 5` and `*x.begin() = 10` executed concurrently can result in a data race. As an exception to the general rule, for a `vector<bool> y`, `y[0] = true` can race with `y[1] = true`. — *end note*]

# std Containers

- A C++ standard library function shall not directly or indirectly modify objects ([intro.multithread]) accessible by threads other than the current thread unless the objects are accessed directly or indirectly via the function's non-const arguments, including this.

- For purposes of avoiding data races ([res.on.data.races]), implementations shall consider the following functions to be const: begin, end, rbegin, rend, front, back, data, find, lower _bound, upper_bound, equal_range, at and, except in associative or unordered associative containers, operator[].

# `std` Containers

All `const` member functions can be called concurrently by different threads on the same container.

# Avoiding ABI Breaks

Sutter's Mill

Herb Sutter on software development

## My little New Year's Week project (and maybe one for you?)

Herb Sutter 2025-01-02 7 Minutes

*[Updates: Clarified that an intrusive discriminator would be far beyond what most people mean by "C++ ABI break." Mentioned unique addresses and common initial sequences. Added "unknown" state for passing to opaque functions.]*

Here is my little New Year's Week project: Trying to write a small library to enable compiler support for automatic raw `union` member access checking.

### The problem, and what's needed

During 2024, I started thinking: **What would it take to make C/C++ `union` accesses type-checked?** Obviously, the ideal is to change naked `union` types to something safe.**(*)** But because it will take time and effort for the world to adopt any solution that requires making source code changes, I wondered how much of the safety we might be able to get, at what overhead cost, just by recompiling existing code in a way that instruments ordinary `union` objects?

Search ...

**Follow by email**

Email Address

Subscribe

I'm an author and speaker, and a programming language nerd whose focus is on enabling our program code to be both clean *and* fast. I've been writing about programming since 1993, usually about C++ or about concurrency and parallelism. I'm the designer

# Avoiding ABI Breaks: Extrinsic Storage

```
//  That's it. Here's an example:
//    {
//      union Test { int a; double b; };
//      Test t = {42};                          union_registry<>::on_set_alternative(&u,0);
//      std::cout << t.a;                       union_registry<>::on_get_alternative(&u,0);
//      t.b = 3.14159;                          union_registry<>::on_set_alternative(&u,1);
//      std::cout << t.b;                       union_registry<>::on_get_alternative(&u,1);
//    }                                         union_registry<>::on_destroy(&u);
//
```

# Avoiding ABI Breaks: Extrinsic Storage

```
class data_race_registry {
    static inline auto tags    = extrinsic_storage<check_state>{};

public:
   static inline auto get_state(void* pobj) noexcept {
        return *tags.find_or_insert(pobj);
    }

    static inline auto on_destroy(void* pobj) noexcept -> void {
        tags.erase(pobj);
    }
};
```

```
constexpr const_reference operator[](size_type __pos) const noexcept {
  _LIBCPP_ASSERT_VALID_ELEMENT_ACCESS(__pos <= size(), "string index out of bounds");
  scoped_check<check_type::read> _ (data_race_registry::get_state (this));

  if (__builtin_constant_p(__pos) && !__fits_in_sso(__pos))
    return *(__get_long_pointer() + __pos);

  return *(data() + __pos);
}
```

# Data Races as Contract Violations

```
void read_started (check_state& state)
{
    ++state.num_readers;
    contract_assert (! state.is_writing); // read during active write
}

void write_started (check_state& state)
{
    contract_assert (! state.is_writing.exchange (true)) // write during active write
    contract_assert (state.num_readers == 0) // write during active read
}
```

```
constexpr const_reference operator[](size_type __pos) const noexcept {
  _LIBCPP_ASSERT_VALID_ELEMENT_ACCESS(__pos <= size(), "string index out of bounds");
  scoped_check<check_type::read> _ (data_race_registry::get_state (this));

  if (__builtin_constant_p(__pos) && !__fits_in_sso(__pos))
    return *(__get_long_pointer() + __pos);

  return *(data() + __pos);
}
```

# Avoiding ABI Breaks: Extrinsic Storage

```
constexpr const_reference operator[](size_type __pos) const noexcept
    pre (can_read(data_race_registry::get_state (this)))
{
    //...
```

```
basic_string& replace(size_type __pos1, size_type __n1, const basic_string& __str)
    pre (can_write(data_race_registry::get_state (this)))
{
    //...
```

# Data Race Detection

- Extremely limited
  - Works on function entry/exit, not memory
  - All bets are off if functions return references/pointers
  - Only works on types that don't expose their memory
  - Member function delegation not shown
- Could be used to check container contracts
- Use TSan!

# C++ Profiles?

## 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.

# C++ Profiles?

*Look into the possibility of statically detecting aliases in more than one thread to mutable data*

# C++ Profiles?

## 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.

# C++ Profiles?

***Mutability**: Prefer to pass (and keep) pointers to **const**.*

# C++ Profiles?

## 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.

# C++ Profiles?

***Aliasing**: statically detect if a pointer is passed to another thread. <snip>*

# Conclusion

- C++ needs a way to identify “isolation boundaries”
  - I.e. `send`
- This introduces strong aliasing and lifetime requirements
- This is not compatible with existing pointers/references
- Reflection can help us write in the styles of other languages which have better thread safety
  - Safely encapsulates pointers
- For “C++ performance” and “Don’t pay for what you don’t use” we need borrow checking:
  - Sean Baxter: “Safe C++” wg21.link/P3390

# What Can C++ Learn About Thread Safety From Other Languages

David Rowland
@drowaudio

## *Questions?*

*Slides/video:*

drowaudio.github.io/presentations