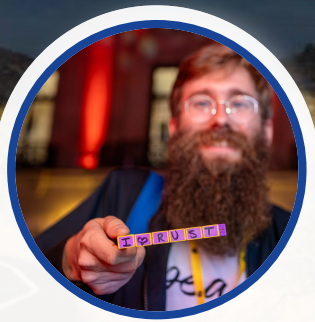
Eclipse Software Defined Vehicle

Self-Referential Structs in Rust

February 2025



Pete LeVasseur
Eclipse uProtocol Maintainer

woven by TOYOTA

background

What’s a self-referential struct in Rust?

It’s everyone’s favorite recurring topic: self-referential structs

help

```
struct Struct { view: &mut u8, stashed: u8 };
let stashed = 1u8;
let s = Struct { view: &mut stashed, stashed };
```

Attempting to store something in a struct that references something else in the same struct

The C or C++ equivalent of the above code would compile and be usable. *Why’s Rust different?*

Rust Programming Language Users Forum: *It’s everyone’s favorite recurring topic: self-referential structs*
https://users.rust-lang.org/t/its-everyones-favorite-recurring-topic-self-referential-structs/91105

Aliasing: Why self-referential structs are hard in Rust

"*Aliasing* occurs when one pointer or reference points to a "span" of memory that overlaps with the span of another pointer or reference. A span of memory is similar to how a slice works: there's a base byte address as well as a length in bytes." [1]

> "…to emphasize, one you didn't list here and the main one in general is that aliasing something covered by a &mut (or owned by a Box) is UB. That rule is at the heart of Rust's memory and concurrency guarantees.
>
> Alternatively put, once you "use" something, any exclusive references to it must no longer be valid." [2]

References:
[1] Rust's Unsafe Coding Guidelines: *Glossary* https://rust-lang.github.io/unsafe-code-guidelines/glossary.html#aliasing
[2] Rust Programming Language Users Forum: *It's everyone's favorite recurring topic: self-referential structs* https://users.rust-lang.org/t/its-everyones-favorite-recurring-topic-self-referential-structs/91105/3

motivation

Occurrence in the wild: mcap crate

We want to:

1. use the `memmap` crate to memory map in an .mcap file to a `Mmap`
2. use the `MessageStream` struct from the mcap crate to take a reference to that `Mmap`
3. so that we can keep the `Mmap` alive long enough for the duration that `MessageStream` is alive

```rust
pub struct MemmapMessageStream<'a> {
    // keep the Mmap here so it's alive as long as
    mmap: Mmap,
    // ... needed by the MessageStream
    message_stream: MessageStream<'a>,
}
```

Note: All code snippets in this presentation are sample code to explain concepts!

Trying to use the MemmapMessageStream

```rust
use anyhow::Result;
use mcap::{McapError, Message, MessageStream};
use memmap::Mmap;
use std::{fs::File, path::Path};

pub struct MemmapMessageStream<'a> {
    // keep the Mmap here so it's alive as long as
    mmap: Mmap,
    // ... needed by the MessageStream
    message_stream: MessageStream<'a>,
}

impl<'a> MemmapMessageStream<'a> {
    pub fn new<P: AsRef<Path>>(p: P) -> Result<MemmapMessageStream<'a>> {
        let file = File::open(p)?;
        let mmap = unsafe { Mmap::map(&file) }?;
        let message_stream = MessageStream::new(&mmap)?;
        Ok(MemmapMessageStream {
            mmap,
            message_stream,
        })
    }

    pub fn next(&mut self) -> Option<std::result::Result<Message, McapError>> {
        self.message_stream.next()
    }
}
```

```
error[E0515]: cannot return value referencing local variable `mmap`
  --> src/memmap_message_stream.rs:18:9
   |
17 |           let message_stream = MessageStream::new(&mmap)?;
   |                                                    ----- `mmap` is borrowed here
18 | /         Ok(MemmapMessageStream {
19 | |             mmap,
20 | |             message_stream,
21 | |         })
   | |__________^ returns a value referencing data owned by the current function
```

```
error[E0505]: cannot move out of `mmap` because it is borrowed
  --> src/memmap_message_stream.rs:19:13
   |
13 |   impl<'a> MemmapMessageStream<'a> {
   |        -- lifetime `'a` defined here
...
16 |           let mmap = unsafe { Mmap::map(&file) }?;
   |               ---- binding `mmap` declared here
17 |           let message_stream = MessageStream::new(&mmap)?;
   |                                                    ----- borrow of `mmap` occurs here
18 | /         Ok(MemmapMessageStream {
19 | |             mmap,
   | |             ^^^^ move out of `mmap` occurs here
20 | |             message_stream,
21 | |         })
   | |__________- returning this value requires that `mmap` is borrowed for `'a`
```


ouroboros

ouroboros, designed for self-referential structs

struct using ouroboros

```rust
use anyhow::Result;
use mcap::{McapError, Message, MessageStream};
use memmap::Mmap;
use ouroboros::self_referencing;
use std::{fs::File, path::Path};

#[self_referencing]
pub struct MemmapMessageStream {
    // keep the Mmap here so it's alive as long as
    mmap: Mmap,
    // ... needed by the MessageStream
    #[not_covariant]
    #[borrows(mmap)]
    message_stream: MessageStream<'this>,
}
```

`self_referencing` attribute macro

`#[not_covariant]`: covariance is a bit out of scope, but this says that the `'this` lifetime cannot be shortened

`'this`: the `'this` lifetime is created by the `#[self_referencing]` macro and should be used on all references marked by the `#[borrows]` macro

`#[borrows(mmap)]` informs that this struct member will borrow mmap

ouroboros, designed for self-referential structs

impl using ouroboros

```rust
impl MemmapMessageStream {
    pub fn new_stream<P: AsRef<Path>>(p: P) -> Result<MemmapMessageStream> {
        let file = File::open(p)?;
        let mmap = unsafe { Mmap::map(&file) }?;
        Ok(MemmapMessageStreamBuilder {
            mmap,
            message_stream_builder: |m: &Mmap| {
                MessageStream::new(m).expect("Unable to initialize with Mmap")
            },
        }
        .build())
    }

    pub fn next(&mut self) -> Option<std::result::Result<Message, McapError>> {
        self.with_message_stream_mut(|ms| ms.next())
    }
}
```

`MemmapMessageStreamBuilder` created by `#[self_referencing]` which allows us to pass in a closure accepting the `Mmap`

The `with_message_stream_mut` function created allows us mutable access to the `MessageStream`

macro

ouroboros, designed for self-referential structs

Expanded struct

`cargo rustc -- -Zunpretty=expanded`

```
1 pub struct MemmapMessageStream {
2     actual_data: ::core::mem::MaybeUninit<MemmapMessageStreamInternal>,
3 }
4 struct MemmapMessageStreamInternal {
5     #[doc(hidden)]
6     message_stream: MessageStream<'static>,
7     #[doc(hidden)]
8     mmap: ::ouroboros::macro_help::AliasableBox<Mmap>,
9 }
```

ouroboros, designed for self-referential structs

Expanded MemmapMessageStreamBuilder

`message_stream_builder` is an `FnOnce`, a closure which accepts a `Mmap` with lifetime `'this` and returns a `MessageStream` of lifetime `'this`

```rust
pub(super) struct MemmapMessageStreamBuilder<
    MessageStreamBuilder_: for<'this> ::core::ops::FnOnce(&'this Mmap) -> MessageStream<'this>,
> {
    pub(super) mmap: Mmap,
    pub(super) message_stream_builder: MessageStreamBuilder_,
}
impl<
        MessageStreamBuilder_: for<'this> ::core::ops::FnOnce(&'this Mmap) -> MessageStream<'this>,
    > MemmapMessageStreamBuilder<MessageStreamBuilder_>
{
    pub(super) fn build(self) -> MemmapMessageStream {
        MemmapMessageStream::new(self.mmap, self.message_stream_builder)
    }
}
```

ouroboros, designed for self-referential structs

Expanded MemmapMessageStream::new()

```rust
impl MemmapMessageStream {
    pub(super) fn new(
        mmap: Mmap,
        message_stream_builder: impl for<'this> ::core::ops::FnOnce(&'this Mmap) -> MessageStream<'this>,
    ) -> MemmapMessageStream {
        let mmap = ::ouroboros::macro_help::aliasable_boxed(mmap);
        let mmap_illegal_static_reference =
            unsafe { ::ouroboros::macro_help::change_lifetime(&*mmap) };
        let message_stream = message_stream_builder(mmap_illegal_static_reference);
        unsafe {
            Self {
                actual_data: ::core::mem::MaybeUninit::new(MemmapMessageStreamInternal {
                    mmap,
                    message_stream,
                }),
            }
        }
    }

    // ...
}
```

ouroboros, designed for self-referential structs
aliasable_boxed()

```
pub fn aliasable_boxed<T>(data: T) -> AliasableBox<T> {
    AliasableBox::from_unique(UniqueBox::new(data))
}
```

rust-aliasable

Rust library providing basic aliasable (non `core::ptr::Unique`) types
Documentation hosted on docs.rs.

```
aliasable = "0.1"
```

Why?

Used for escaping `noalias` when multiple raw pointers may point to the same data.

crates.io: *aliasable* https://crates.io/crates/aliasable

ouroboros, designed for self-referential structs
change_lifetime()

```rust
/// Converts a reference to an object to a static reference This is
/// obviously unsafe because the compiler can no longer guarantee that the
/// data outlives the reference.  It is up to the consumer to get rid of the
/// reference before the container is dropped. The + 'static ensures that
/// whatever we are referring to will remain valid indefinitely, that there
/// are no limitations on how long the pointer itself can live.
///
/// # Safety
///
/// The caller must ensure that the returned reference is not used after the originally passed
/// reference would become invalid.
pub unsafe fn change_lifetime<'old, 'new: 'old, T: 'new>(data: &'old T) -> &'new T {
    &*(data as *const _)
}
```

crates.io: *aliasable* https://crates.io/crates/aliasable

ouroboros, designed for self-referential structs

Drop impl

The `MaybeUninit` held inside of `MemmapMessageStream` is dropped by the `Drop` impl, ensuring that we deallocate the `mmap_illegal_static_reference`

```rust
#[repr(transparent)]
pub struct MemmapMessageStream {
    actual_data: ::core::mem::MaybeUninit<MemmapMessageStreamInternal>,
}
struct MemmapMessageStreamInternal {
    #[doc(hidden)]
    message_stream: MessageStream<'static>,
    #[doc(hidden)]
    mmap: ::ouroboros::macro_help::AliasableBox<Mmap>,
}
impl ::core::ops::Drop for MemmapMessageStream {
    fn drop(&mut self) {
        unsafe { self.actual_data.assume_init_drop() };
    }
}
```

Rust stdlib documentation: *MaybeUninit*
https://doc.rust-lang.org/std/mem/union.MaybeUninit.html#method.assume_init_drop

ouroboros, designed for self-referential structs
with_message_stream_mut()

Allows us to pass in an `FnOnce` to be able to do mutable operations on the `MessageStream`

```rust
#[inline(always)]
pub(super) fn with_message_stream_mut<'outer_borrow, ReturnType>(
    &'outer_borrow mut self,
    user: impl for<'this> ::core::ops::FnOnce(&'outer_borrow mut MessageStream<'this>) -> ReturnType,
) -> ReturnType {
    let field = &mut unsafe { self.actual_data.assume_init_mut() }.message_stream;
    user(field)
}
```

Rust stdlib documentation: *MaybeUninit*
https://doc.rust-lang.org/std/mem/union.MaybeUninit.html#method.assume_init_drop

run it

Reading an MCAP file using `MemmapMessageStream`

```
$ cargo run --bin mmap_backed

   Compiling memmap-message-stream-works v0.1.0
(/home/peter/presentations/self-referential-structs/memmap-message-stre
am-works)

    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.89s

     Running `target/debug/mmap_backed`

next: Ok(Message { channel: Channel { id: 0, topic: "ping", schema:
Some(Schema { name: "Buffer", encoding: "", .. }), message_encoding:
"", metadata: {} }, sequence: 0, log_time: 1736458452082329110,
publish_time: 1736458452065969963, data: [0, 1, 0, 0, 4, 0, 0, 0, 32,
0, 0, 0] })
```

Takeaway: the self-referential strategy to keep the `Mmap` alive worked!


efficiency

ouroboros, designed for self-referential structs

Use cargo-show-asm on with_message_stream_mut()

Want to ensure that `MemmapMessageStream::next()` compiles away the closure

Looks like in the generated assembly we directly call `Iterator::next()` as implemented on `MessageStream`. Great!

```rust
#[self_referencing]
pub struct MemmapMessageStream {
    // keep the Mmap here so it's alive as long as
    mmap: Mmap,
    // ... needed by the MessageStream
    #[not_covariant]
    #[borrows(mmap)]
    message_stream: MessageStream<'this>,
}

impl MemmapMessageStream {
    // ...

    #[inline(never)]
    pub fn next(&mut self) -> Option<std::result::Result<Message, McapError>> {
        self.with_message_stream_mut(|ms| ms.next())
    }
}
```

```
.section ".text.memmap_message_stream_works::memmap_message_stream::<impl memmap_message_stream_works::memmap_message_stream::ouroboros_impl_memmap_mes
sage_stream::MemmapMessageStream>::next","ax",@progbits
        .p2align        4, 0x90
.type   memmap_message_stream_works::memmap_message_stream::<impl memmap_message_stream_works::memmap_message_stream::ouroboros_impl_memmap_message_str
eam::MemmapMessageStream>::next,@function
memmap_message_stream_works::memmap_message_stream::<impl memmap_message_stream_works::memmap_message_stream::ouroboros_impl_memmap_message_stream::Mem
mapMessageStream>::next:
        .cfi_startproc
        jmp qword ptr [rip + <mcap::read::MessageStream as core::iter::traits::iterator::Iterator>::next@GOTPCREL]
```

crates.io: *cargo plugin - cargo-show-asm* https://crates.io/crates/cargo-show-asm

safety

Any undefined behavior (UB) using ouroboros?

Generally the Rust compiler takes responsibility for ensuring certain safe properties of code, e.g. memory-safety which could lead to undefined behavior (UB)

As we saw, in order to accomplish self-referential structs, ouroboros uses `unsafe` in key locations where the compiler may be too conservative and would have caused a compilation error.

> *"Unsafe Rust exists because, by nature, static analysis is conservative. When the compiler tries to determine whether or not code upholds the guarantees, it's better for it to reject some valid programs than to accept some invalid programs."*

Key point: "`unsafe`" moves the responsibility for checking safe properties of code from compiler to the engineer, e.g. memory-safety. Code marked as `unsafe` deserves more attention during design and code review.

The Rust Programming Language: *Unsafe Rust* https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html

Running miri to check for Undefined Behavior (UB)

File-backed memory mappings unsupported

```
$MIRIFLAGS=-Zmiri-disable-isolation cargo miri run --bin mmap_backed
```

```
error: unsupported operation: Miri does not support file-backed memory mappings
  --> /home/peter/.cargo/registry/src/index.crates.io-6f17d22bba15001f/memmap-0.7.0/src/unix.rs:49:23
   |
49 |                 let ptr = libc::mmap(
   |  _______________________^
50 | |                 ptr::null_mut(),
51 | |                 aligned_len as libc::size_t,
52 | |                 prot,
...  |
55 | |                 aligned_offset as libc::off_t,
56 | |             );
   | |_____________^ Miri does not support file-backed memory mappings
   |
```

Running miri to check for Undefined Behavior (UB)

Use a Vec-backed approximation

```
$cargo miri run --bin vec_backed
```

```
error: unsupported operation: can't call foreign function `ZSTD_createDCtx` on OS `linux`
   --> /home/peter/.cargo/registry/src/index.crates.io-6f17d22bba15001f/zstd-safe-5.0.2+zstd.1.5.2/src/lib.rs:707:35
    |
707 |             NonNull::new(unsafe { zstd_sys::ZSTD_createDCtx() })?,
    |                                   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ can't call foreign function `ZSTD_createDCtx` on OS `linux`
    |
    = help: if this is a basic API commonly used on this target, please report an issue with Miri
    = help: however, note that Miri does not aim to support every FFI function out there; for instance, we will not support APIs for things such as GUI
s, scripting languages, or databases
```

Running miri to check for Undefined Behavior (UB)
Just check the ouroboros tests!

No exact test for this scenario, hmmm

GitHub: *ouroboros crate source*
https://github.com/someguynamedjosh/ouroboros/blob/7316c75b988ce97140c824837253161bc411eb22/examples/src/ok_tests.rs

Running miri to check for Undefined Behavior (UB)

Write a test for ouroboros then! - test setup

```rust
struct PhraseRef<'a> {
    data: &'a mut String,
}

impl<'a> PhraseRef<'a> {
    fn change_phrase(&mut self) {
        *self.data = self.data.replace("Hello", "Goodbye");
    }
}

#[self_referencing]
struct DataAndCustomRef {
    data: String,
    #[borrows(mut data)]
    #[not_covariant]
    phrase: PhraseRef<'this>,
}
```

Running miri to check for Undefined Behavior (UB)

Write a test for ouroboros then! - test

```rust
#[test]
fn custom_ref() {
    let mut instance = DataAndCustomRefBuilder {
        data: "Hello world!".to_owned(),
        phrase_builder: |data| PhraseRef { data },
    }
    .build();
    instance.with_phrase_mut(|phrase| phrase.change_phrase());
    let modified_data = instance.into_heads().data;
    assert_eq!(modified_data, "Goodbye world!");
}
```

```
$ cargo miri test --features="miri"
…
test ok_tests::custom_ref ... ok
```

Running miri to check for Undefined Behavior (UB)

Write a test for ouroboros then! - merged upstream

Add test for struct holding original data and a struct which holds a reference to original data. #130

Merged **someguynamedj...** merged 2 commits into `someguynamedjosh:main` from `PLeVasseur:feature/test-struct-holding-ref` 2 days ago

Conversation 0 | Commits 2 | Checks 6 | Files changed 1

PLeVasseur commented 3 days ago — Contributor

closes #129

Reviewers

No reviews

Assignees

No one assigned

Test merged upstream:
https://github.com/someguynamedjosh/ouroboros/pull/130

std::pin::Pin

Question: Can we use `std::pin::Pin` instead?

Yes! Here's the code

```rust
use anyhow::Result;
use mcap::{McapError, Message, MessageStream};
use memmap::Mmap;
use std::{fs::File, marker::PhantomPinned, path::Path, pin::Pin};

pub struct MemmapMessageStream {
    // No lifetime parameter needed anymore
    mmap: Mmap,
    message_stream: MessageStream<'static>, // We'll transmute the lifetime
    _pin: PhantomPinned,                    // Prevent Unpin implementation
}
```

```rust
impl MemmapMessageStream {
    pub fn new_stream<P: AsRef<Path>>(p: P) -> Result<Pin<Box<Self>>> {
        let file = File::open(p)?;
        let mmap = unsafe { Mmap::map(&file) }?;

        // Create a struct instance without the message_stream first
        let mut boxed = Box::new(Self {
            mmap,
            // Create with dummy message stream, we'll initialize it properly below
            message_stream: unsafe { std::mem::transmute(MessageStream::new(&[])?) },
            _pin: PhantomPinned,
        });

        // Now we can create the message stream with a reference to the pinned mmap
        let message_stream = unsafe {
            // This transmute is safe because the mmap will live as long as the struct
            // and we ensure the struct can't be moved via PhantomPinned
            std::mem::transmute(MessageStream::new(&boxed.mmap)?)
        };
        boxed.message_stream = message_stream;

        // Pin the box
        Ok(Box::into_pin(boxed))
    }

    #[inline(never)]
    pub fn next(self: &mut Pin<Box<Self>>) -> Option<std::result::Result<Message, McapError>> {
        // Safety: we only access message_stream which is okay to mutate
        unsafe { self.as_mut().get_unchecked_mut() }
            .message_stream
            .next()
    }
}
```

We use `PhantomPinned` to ensure that this struct's contents cannot be moved in memory

Using `std::mem::transmute` here to extend the lifetime of mmap. As the stdlib docs note:

"This is advanced, very `unsafe` Rust!"

Question: Can we use `std::pin::Pin` instead?

Miri passes without finding anything unsound

```
$cargo miri run --bin vec_backed
```

```
1     Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.11s
2      Running `/home/peter/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/bin/cargo-miri runner target/miri/x86_64-unknown-linux-gnu/debug/vec_back
3 ed`
4 next: Ok(Message { channel: Channel { id: 0, topic: "foo", schema: None, message_encoding: "", metadata: {} }, sequence: 0, log_time: 0, publish_time:
5 0, data: [0] })
6 Hello, world!
```

Question: Can we use `std::pin::Pin` instead?

Are there differences in generated assembly of `next()`?

Using ouroboros

No! They are identical

```
.section ".text.memmap_message_stream_works::memmap_message_stream::<impl memmap_message_stream_works::memmap_message_stream::ouroboros_impl_memmap_mes
sage_stream::MemmapMessageStream>::next","ax",@progbits
        .p2align        4, 0x90
.type   memmap_message_stream_works::memmap_message_stream::<impl memmap_message_stream_works::memmap_message_stream::ouroboros_impl_memmap_message_str
eam::MemmapMessageStream>::next,@function
memmap_message_stream_works::memmap_message_stream::<impl memmap_message_stream_works::memmap_message_stream::ouroboros_impl_memmap_message_stream::Mem
mapMessageStream>::next:
        .cfi_startproc
        jmp qword ptr [rip + <mcap::read::MessageStream as core::iter::traits::iterator::Iterator>::next@GOTPCREL]
```

Using `Pin`

```
.section .text.mmap_backed::memmap_message_stream::MemmapMessageStream::next,"ax",@progbits
        .p2align        4, 0x90
.type   mmap_backed::memmap_message_stream::MemmapMessageStream::next,@function
mmap_backed::memmap_message_stream::MemmapMessageStream::next:
        .cfi_startproc
        jmp qword ptr [rip + <mcap::read::MessageStream as core::iter::traits::iterator::Iterator>::next@GOTPCREL]
```

lessons

When does it make sense to use each?

Generally ouroboros has less sharp edges, recommended

Using `std::pin::Pin`

Pros	Cons
Built into the standard library	Requires `unsafe` code to implement correctly
Well-documented and understood (used heavily in async/await)	Easy to make mistakes with the `unsafe` code that could lead to undefined behavior
More flexible if we need to do `unsafe` (somewhat of a con too) or have specific memory layout requirements	More verbose implementation & have to work with Pin's API which can be awkward

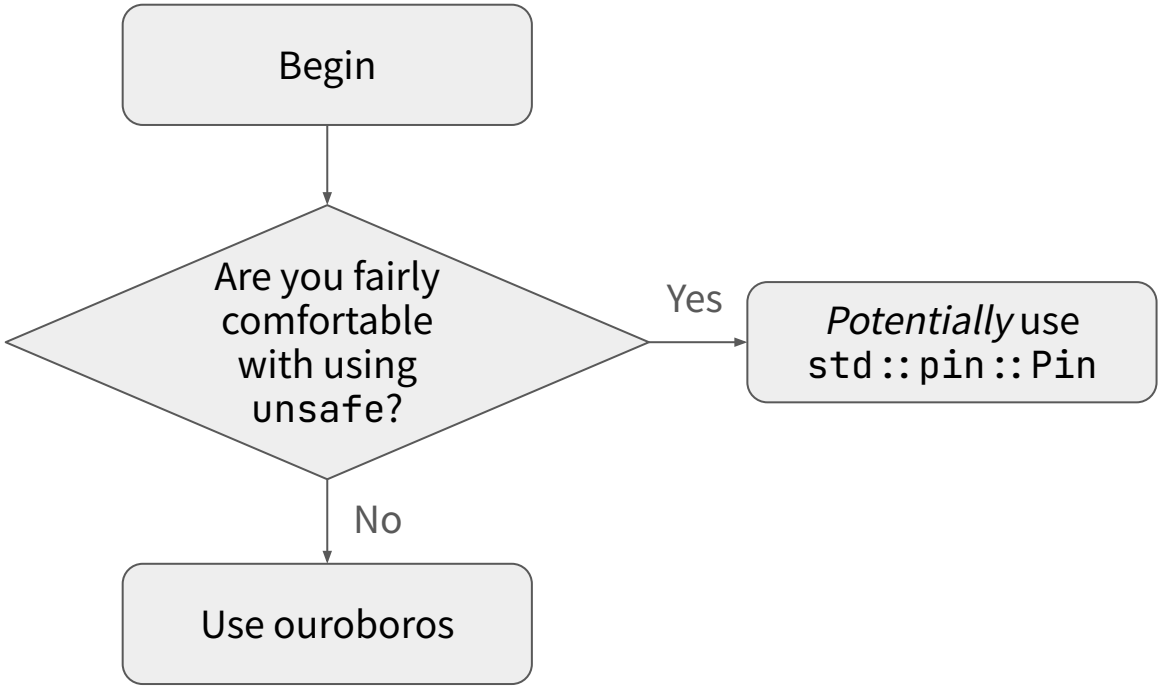
Using ouroboros

Pros	Cons
No `unsafe` code needed in implementation & ouroboros handles all the unsafe details	Additional dependency
More ergonomic API for common use cases	Procedural macros can make compilation slower
Less boilerplate code required	Less flexible for unusual cases

**Refer to slide 24 for definition of `unsafe` and further details*

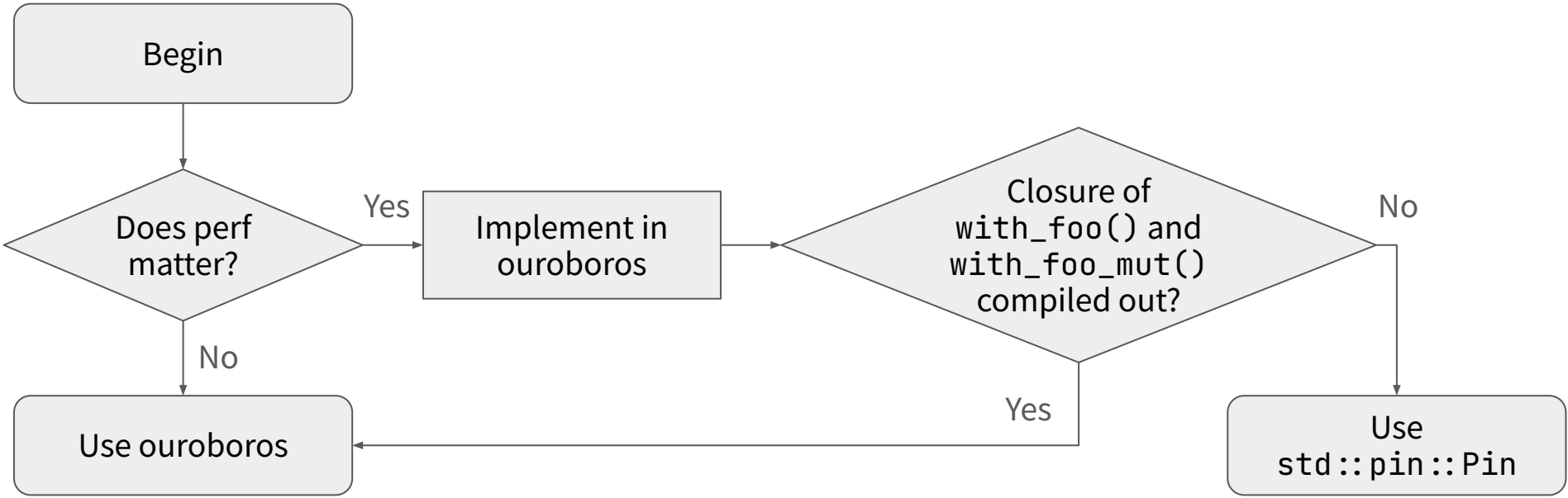
Thinking about ouroboros vs `std::pin::Pin`

Rust expertise



Thinking about ouroboros vs `std::pin::Pin`

Performance



SDV™
Eclipse Software Defined Vehicle

Thank You!

JOIN US ON GITHUB!

Eclipse uProtocol GitHub Project:
https://github.com/eclipse-uprotocol
Eclipse SDV Blueprint: Service-to-Signal:
https://github.com/eclipse-sdv-blueprints/service-to-signal

Eclipse uProtocol Website:
https://uprotocol.org/