PINS

A Rustlang Mystery

MARTIN HOFFMANN

NLNETLABS

Prologue

WHAT?

FUTURES 0.1

```
pub trait Future {
    type Item;
    type Error;

    fn poll(
        &mut self,

    ) -> Poll<Self::Item, Self::Error>;
}
```

STD::FUTURE

```rust
pub trait Future {
    type Output;


    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context,
    ) -> Poll<Self::Output>;
}
```

TOKIO::IO

```rust
pub trait AsyncRead {
    fn poll_read(
        self: Pin<&mut Self>,
        cx: &mut Context,
        buf: &mut ReadBuf,
    ) -> Poll<Result<(), io::Error>>;
}
```

Chapter One

MOVE BY "MEMCPY"

MOVE BY "MEMCPY"

usize

MOVE BY "MEMCPY"

```
struct Buffer {
    buf: [u8; 4096],
    pos: usize,
}
```

MOVE BY "MEMCPY"

```
enum TcpOrTlsStream {
    Tcp {
        stream: TcpStream,
        metrics: TcpMetrics,
    }
    Tls {
        stream: TlsStream,
        metrics: TlsMetrics,
        config: Arc<ClientConfig>,
    }
}
```

MOVE SEMANTICS

```
let stream = TcpStream::connect("127.0.0.1:8080")?;
let metrics = TcpMetrics::default();

let tcp_or_tls = TcpOrTlsStream::Tcp { stream, metrics };
```

MOVE SEMANTICS

```
let stream = TcpStream::connect("127.0.0.1:8080")?;
let metrics = TcpMetrics::default();

let tcp_or_tls = TcpOrTlsStream::Tcp { stream, metrics };

let shared = Arc::new(tcp_or_tls);
tx.send(shared);
```

MOVING POINTERS

```rust
let shared = Arc::new(tcp_or_tls);

let mut vec = Vec::new();

vec.push(shared);
```

Moves the arc only!
tcp_or_tls stays where it is!

SNEAKY MOVE

- *core::mem:*

```
pub fn swap<T>(x: &mut T, y: &mut T) {
    unsafe {
        let a = ptr::read(x);
        let b = ptr::read(y);
        ptr::write(x, b);
        ptr::write(y, a);
    }
}
```

IN SHORT

- Move by “memcpy” can happen for:
  - unadulterated owned `T`
  - `&mut T`
- Foundation for happy use of the stack.

Chapter Two

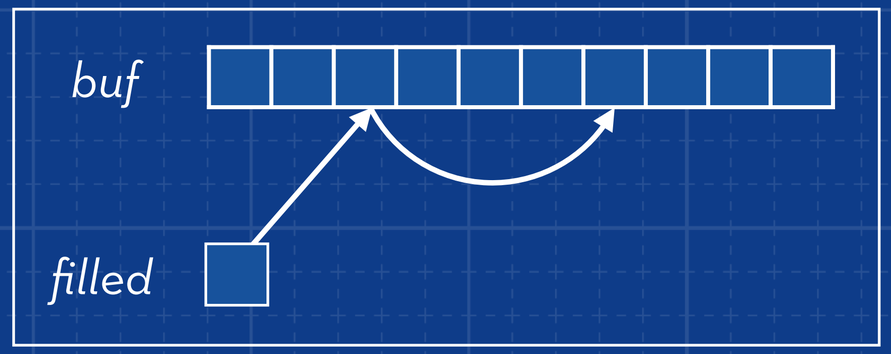
THERE'S ALWAYS SOMETHING

SELF-REFERENTIAL STRUCT

```
struct Buf {
    buf: [u8; 4096],
    filled: &[u8],
}
```

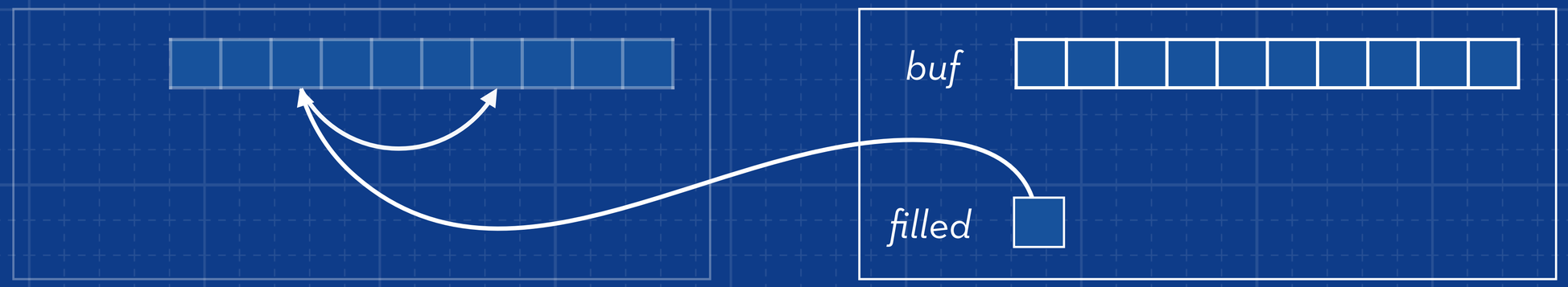
SELF-REFERENTIAL STRUCT

```
struct Buf {
    buf: [u8; 4096],
    filled: &[u8],
}
```



SELF-REFERENTIAL STRUCT

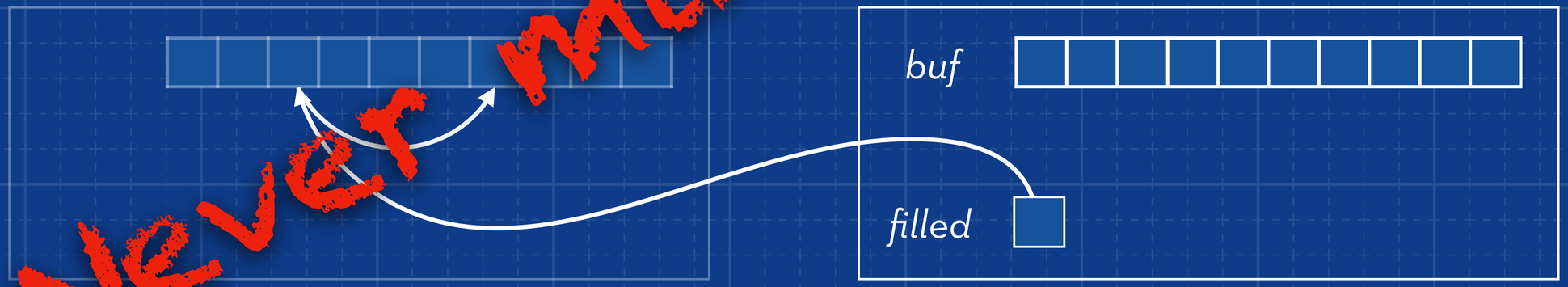
```
struct Buf {
    buf: [u8; 4096],
    filled: &[u8],
}
```



SELF-REFERENTIAL STRUCT

```
struct Buf {
    buf: [u8; 4096],
    filled: &[u8],
}
```

Never mind: Lifetimes!



ASYNC BLOCKS & FUNCTIONS

```
async fn send(msg: Vec<u8>) {
    let (left, right)
        = msg.split_at(2);

    send(left).await;



    send(right).await;
    send(msg).await;
}
```

ASYNC BLOCKS & FUNCTIONS

```
async fn send(msg: Vec<u8>) {
    let (left, right)
        = msg.split_at(2);

    send(left).await;

    send(right).await;
    send(msg).await;
}
```

```
enum SendFuture {
    Start { msg: Vec<u8> },

    Await1 {
        msg: Vec<u8>,
        right: &[u8],
    }
    Await2 { msg: Vec<u8> },
    Await3,
}
```

STD::FUTURE

```rust
pub trait Future {
    type Output;

    fn poll(
        self: Pin<&mut Self>,
        cx: &mut Context,
    ) -> Poll<Self::Output>;
}
```

Pin makes move impossible!

Chapter Three

BUT HOW?

PIN IS FOR POINTERS

```
struct Pin<P> where P: Deref<Target = T>


Pin<&T>
Pin<&mut T>

Pin<Box<T>>
Pin<Arc<T>>
```

NO ACCESS TO T AND &MUT T

- `Pin<Box<T>>`
  - cannot get owned `T` because pointer.
  - cannot call `Box::as_mut` – you need `&mut self` which the pin won't give you

BUT WAIT!

```
impl<P> DerefMut for Pin<P>
where
    P: DerefMut,

{
    fn deref_mut(
        &mut self
    ) -> &mut P::Target
    { ... }
}
```

UNPIN

```rust
impl<P> DerefMut for Pin<P>
where
    P: DerefMut,
    <P as Deref>::Target: Unpin,
{
    fn deref_mut(
        &mut self
    ) -> &mut P::Target
    { ... }
}
```

UNPIN

- Crucial: Unpin is a property of the pointed-to type T, not the pointer P.
- Statement that moving by memcpy is fine.
- Auto-trait a la Send or Sync.
  - all built-ins are Unpin,
  - complex types are Unpin if all their components are Unpin,
  - *everything is Unpin?*

UNPIN

```
struct StringWithSlice {
    data: String,
    slice: ptr::NonNull<String>,
    _pin: marker::PhantomPinned,
}
```

Nope!

IN SHORT

- Pointer types can be pinned,
  - block move by memcpy by blocking forms of access to pointed-to value where that may happen,
  - but don't block that access if the pointed-to value doesn't mind being moved.
- The latter is true nearly always.
  - Everything is fine.

Chapter Four

HAPPILY EVER AFTER

```rust
struct MetricStream {
    stream: tokio::net::TcpStream,
    metrics: TcpMetrics,
}

impl AsyncRead for MetricStream {
    fn poll_read(
        self: Pin<&mut Self>, cx: &mut Context, buf: &mut ReadBuf,
    ) -> Poll<Result<(), io::Error>> {

        // Now what?

    }
}
```

https://docs.rs/tokio/1.28.0/tokio/net/struct.TcpStream.html#impl-Unpin-for-TcpStream

DOCS.RS | tokio-1.28.0 | Platform | Feature flags | Rust | Find crate

TcpStream

Methods

- async_io
- connect
- from_std
- into_split
- into_std
- linger
- local_addr
- nodelay
- peek
- peer_addr
- poll_peek
- poll_read_ready
- poll_write_ready
- readable
- ready
- set_linger
- set_nodelay
- set_ttl

Auto Trait Implementations

```
impl RefUnwindSafe for TcpStream
impl Send for TcpStream
impl Sync for TcpStream
impl Unpin for TcpStream
impl UnwindSafe for TcpStream
```

Blanket Implementations

```
[+] impl<T> Any for T                                   source
    where
        T: 'static + ?Sized,
[+] impl<R> AsyncReadExt for R                          source
    where
        R: AsyncRead + ?Sized,
[+] impl<W> AsyncWriteExt for W                         source
    where
        W: AsyncWrite + ?Sized,
[+] impl<T> Borrow<T> for T                             source
    where
        T: ?Sized,
[+] impl<T> BorrowMut<T> for T                          source
    where
        T: ?Sized,
```

```rust
struct MetricStream {
    stream: tokio::net::TcpStream,
    metrics: TcpMetrics,
}

impl AsyncRead for MetricStream {
    fn poll_read(
        mut self: Pin<&mut Self>, cx: &mut Context, buf: &mut ReadBuf,
    ) -> Poll<Result<(), io::Error>> {

        let stream = &mut self.stream;

    }
}
```

https://doc.rust-lang.org/std/pin/struct.Pin.html#method.new

Pin

Methods

Trait Implementations

```
impl<P> Pin<P>
where
    P: Deref,
    <P as Deref>::Target: Unpin,
```

source

```
pub fn new(pointer: P) -> Pin<P>
```

const: unstable · source

Construct a new `Pin<P>` around a pointer to some data of a type that implements `Unpin`.

Unlike `Pin::new_unchecked`, this method is safe because the pointer `P` dereferences to an `Unpin` type, which cancels the pinning guarantees.

Examples

Run

```
use std::pin::Pin;

let mut val: u8 = 5;
// We can pin the value, since it doesn't care about being moved
let mut pinned: Pin<&mut u8> = Pin::new(&mut val);
```

```
pub fn into_inner(pin: Pin<P>) -> P
```

1.39.0 (const: unstable) · source

Unwraps this `Pin<P>` returning the underlying pointer.

This requires that the data inside this `Pin` implements `Unpin` so that we can ignore the pinning invariants when unwrapping it.

Examples

```
use std::pin::Pin;

let mut val: u8 = 5;
let pinned: Pin<&mut u8> = Pin::new(&mut val);
// Unwrap the pin to get a reference to the value
let r = Pin::into_inner(pinned);
```

```rust
struct MetricStream {
    stream: tokio::net::TcpStream,
    metrics: TcpMetrics,
}

impl AsyncRead for MetricStream {
    fn poll_read(
        mut self: Pin<&mut Self>, cx: &mut Context, buf: &mut ReadBuf,
    ) -> Poll<Result<(), io::Error>> {

        let stream = &mut self.stream;
        let pin = Pin::new(stream);
        pin.poll_read(cx, buf)
    }
}
```

```rust
struct MetricStream {
    stream: tokio::net::TcpStream,
    metrics: TcpMetrics,
}

impl AsyncRead for MetricStream {
    fn poll_read(
        mut self: Pin<&mut Self>, cx: &mut Context, buf: &mut ReadBuf,
    ) -> Poll<Result<(), io::Error>> {

        Pin::new(&mut self.stream).poll_read(cx, buf)
    }
}
```

```rust
struct MetricStream<T> {
    stream: T,
    metrics: Metrics,
}

impl<T: AsyncRead> AsyncRead for MetricStream<T> {
    fn poll_read(
        mut self: Pin<&mut Self>,
        cx: &mut Context, buf: &mut ReadBuf,
    ) -> Poll<Result<(), io::Error>> {
        Pin::new(&mut self.stream).poll_read(cx,buf)
    }
}
```

```
m$ cargo check --all-features
    Checking megaproject v0.1.0 (/Users/m/git/megaproject)
error[E0277]: `T` cannot be unpinned
  --> src/main.rs:18:18
   |
18 |         Pin::new(&mut self.stream).poll_read(cx, buf)
   |         -------- ^^^^^^^^^^^^^^^^ the trait `Unpin` is not implemented for `T`
   |         |
   |         required by a bound introduced by this call
   |
   = note: consider using `Box::pin`
note: required by a bound in `Pin::<P>::new`
  --> /rustc/84c898d65adf2f39a5a98507f1fe0ce10a2b8dbc/library/core/src/pin.rs:501:5
help: consider further restricting this bound
   |
14 | impl<T: AsyncRead + std::marker::Unpin> AsyncRead for MetricStream<T> {
   |                   +++++++++++++++++++++

For more information about this error, try `rustc --explain E0277`.
error: could not compile `megaproject` due to previous error
```

crates.io

pin project

Browse All Crates | Log in with GitHub

Search Results for 'pin project'

Displaying **1-10** of **334** total results

Sort by Relevance

pin-project-lite v0.2.9

A lightweight version of pin-project written with declarative macros.

Repository

All-Time: 100,879,202

Recent: 14,634,007

Updated: about 1 year ago

pin-project v1.0.12

A crate for safe and ergonomic pin-projection.

Repository

All-Time: 70,913,964

Recent: 9,183,042

Updated: 9 months ago

```rust
use pin_project_lite::pin_project;

pin_project! {
    struct MetricStream<T> {
        #[pin]
        stream: T,
        metrics: Metrics,
    }
}

impl<T: AsyncRead> AsyncRead for MetricStream<T> {
    fn poll_read(
        self: Pin<&mut Self>, cx: &mut Context, buf: &mut ReadBuf,
    ) -> Poll<Result<(), io::Error>> {
        let this = self.project();
        this.stream.poll_read(cx, buf)
    }
}
```

Epilogue

TRAGEDY AT THE BURIAL SITE

```rust
use pin_project_lite::pin_project;

pin_project! {
    struct MetricStream<T> {
        #[pin]
        stream: T,
        metrics: Metrics,
    }
}

impl<T> Drop for MetricStream<T> {
    fn drop(&mut self) {
    }
}
```

```
error[E0119]: conflicting implementations of trait `MustNotImplDrop` for type
`MetricStream<_>`
  --> src/main.rs:10:1
   |
10 | / pin_project! {
11 | |     struct MetricStream<T> {
12 | |         #[pin]
13 | |         stream: T,
14 | |         metrics: Metrics,
15 | |     }
16 | | }
   | | ^
   | | |
   | |_first implementation here
   |   conflicting implementation for `MetricStream<_>`
   |
   = note: this error originates in the macro `$crate::__pin_project_make_drop_impl`
which comes from the expansion of the macro `pin_project` (in Nightly builds, run with
-Z macro-backtrace for more info)
```

```rust
use pin_project_lite::pin_project;

pin_project! {
    struct MetricStream<T> {
        #[pin]
        stream: T,
        metrics: Metrics,
    }
}

impl<T> Drop for MetricStream<T> {
    fn drop(&mut self) {
    }
}
```

THANK YOU!

nlnetlabs.nl

martin@nlnetlabs.nl

@partim@social.tchncs.de

NLNETLABS