pin-init: Solving Address Stability in Rust

Benno Lossin <benno.lossin@proton.me>

15. November 2023
Initialization and Address Stability
A Motivation for Rust
Address Stability in the Kernel
Address Stability Support in Rust
A Problem with Initialization
The Solution: pin-init

Field Projections
Field Projections
The Problem with Pin<P>
A possible solution: Pin-Projections
Other Kinds of Field Projections
A Motivation for Rust

Initialization should not be that hard...
A Motivation for Rust

Initialization should not be that hard...

$ git log --oneline --since 2023-01-01 | grep 'fix.*uninitialized'

cca202a5e595 fbdev: hyperv_fb: fix uninitialized local variable use
fc12a722e6b7 exfat: fix setting uninitialized time to ctime/atime
2a76e7679b59 media: platform: mtk-mdp3: fix uninitialized variable in mdp_path_config()
8f8abb863fa5 net: usb: dm9601: fix uninitialized variable use in dm9601_mdio_read
72151ad0cba8 ASoC: codecs: wsa-macro: fix uninitialized stack variables with name prefix
9147b9ded499 btrfs: fix some -Wmaybe-uninitialized warnings in ioctl.c
ec10a35abb3da net: ethernet: mtk_eth_soc: fix uninitialized variable
1c9fd080dffe kunit: fix uninitialized variables bug in attributes filtering
13a0d1088c8f power: supply: qcom_pmi8998_charger: fix uninitialized variable
222a6c4e9ef octeontx2-af: Initialize 'cntr_val' to fix uninitialized symbol error
8362bf82fb54 Input: mcs-touchkey - fix uninitialized use of error in mcs_touchkey_probe()
f72200645c0db netdevsim: fix uninitialized data in nsim_dev_trap_fa_cookie_write()
f61d2d5cf142 sfc: fix uninitialized variable use
97deb66ed4f9 selftests/mm: fix a "possibly uninitialized" warning in pkey-x86.h
df14afeed2e6 ksmbd: fix uninitialized pointer read in smb2_create_link()
48b47f0c9988 ksmbd: fix uninitialized pointer read in ksmbd_vfs_rename()
8fd9f4232d81 btrfs: fix an uninitialized variable warning in btrfs_log_inode
0d9b41daa590 nfc: llcp: fix possible use of uninitialized variable in nfc_llcp_send_connect()
714dd3c29a22 phy: mediatek: hdmi: mt8195: fix uninitialized variable usage in pll_calc
8ba7d5f5ba93 btrfs: fix uninitialized variable warnings
17caf0ba3aa f2fs: fix uninitialized skipped gc_rwsem
08570b7c8db6 gpu: host1x: fix uninitialized variable use
e88a4d4ac27a drm-rockchip: vop2: fix uninitialized variable possible_crtcs
05107edc9101 selftests: sigaltstack: fix -Wuninitialized
7d31677bb7b1 gpu: host1x: fix uninitialized variable use
dc934c183d43 accel/habanalabs: fix a maybe-uninitialized compilation warnings

Benno Lossin <benno.lossin@proton.me>
A Motivation for Rust

- Rust can help to avoid all of these patches.
A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.

But an escape hatch is sometimes needed.

⇒ unsafe code is this escape hatch.

But unsafe code has its own problems:

- Similar to C code it’s easy to make mistakes.
- Allows unchecked operations that the programmer needs to take care of.
- Needs more careful review.

⇒ try to avoid unsafe code.
A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  ⟷ unsafe code is this escape hatch.

Benno Lossin <benno.lossin@proton.me>
A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  \[\rightarrow\text{unsafe} \text{ code is this escape hatch.}\]
- But \text{unsafe} code has its own problems:
A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  \[\Rightarrow\] *unsafe* code is this escape hatch.
- But *unsafe* code has its own problems:
  - Similar to C code it’s easy to make mistakes.
A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  \[\Rightarrow\text{ unsafe code is this escape hatch.}\]
- But \textit{unsafe} code has its own problems:
  - Similar to C code it’s easy to make mistakes.
  - Allows unchecked operations that the programmer needs to take care of.
A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  \[\Rightarrow\text{unsafe}\text{ code is this escape hatch.}\]
- But \texttt{unsafe} code has its own problems:
  - Similar to C code it’s easy to make mistakes.
  - Allows unchecked operations that the programmer needs to take care of.
  - Needs more careful review.
A Motivation for Rust

- Rust can help to avoid all of these patches.
- The compiler will statically check for correctness and reject bad code with compile errors.
- But an escape hatch is sometimes needed.
  \[\implies\] \textit{unsafe} code is this escape hatch.
- But \textit{unsafe} code has its own problems:
  - Similar to C code it’s easy to make mistakes.
  - Allows unchecked operations that the programmer needs to take care of.
  - Needs more careful review.
  \[\implies\] try to avoid \textit{unsafe} code.
Address Stability in the Kernel

Why is address stability needed in the Kernel?
Address Stability in the Kernel

Why is address stability needed in the Kernel?
   ▶ Several data structures such as linked lists need to have a stable address, because other structures have pointers to it.
Address Stability in the Kernel

Why is address stability needed in the Kernel?

- Several data structures such as linked lists need to have a stable address, because other structures have pointers to it.

```
list_head
next
prev
```
Address Stability in the Kernel

Why is address stability needed in the Kernel?

► Several data structures such as linked lists need to have a stable address, because other structures have pointers to it.

![Diagram of list_head, next, and prev]
Why is address stability needed in the Kernel?

- Several data structures such as linked lists need to have a stable address, because other structures have pointers to it.
Address Stability Support in Rust

- All types are *moveable*. 

fn swap<T>(a: &mut T, b: &mut T);
Address Stability Support in Rust

- All types are *movable*.
- Moves are common and frequent.
Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<&mut T>`.
Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>` for example: `Pin<&mut T>`.

How does the compiler ensure that no moves happen?

```rust
fn swap<T>(a: &mut T, b: &mut T);
```

⇒ cannot give access to `&mut T` from `Pin<&mut T>`
Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>` for example: `Pin<&mut T>`.
  \[\Rightarrow\] pointee has a stable address until dropped
Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>` for example: `Pin<&mut T>`.
  \[\Rightarrow\text{ pointee has a stable address until dropped}\]
- How does the compiler ensure that no moves happen?
Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>` for example: `Pin<&mut T>`.
  → pointee has a stable address until dropped
- How does the compiler ensure that no moves happen?

```rust
fn swap<T>(a: &mut T, b: &mut T);
```
Address Stability Support in Rust

- All types are *moveable*.
- Moves are common and frequent.
- Pointers can be *pinned* by wrapping them in `Pin<P>` for example: `Pin<&mut T>`.
  - pointee has a stable address until dropped
- How does the compiler ensure that no moves happen?

```rust
fn swap<T>(a: &mut T, b: &mut T);

⇒ cannot give access to `&mut T` from `Pin<&mut T>`
```
A Problem with Initialization

Consider this bad piece of C code:

```c
struct list_head new_list_head(void) {
    struct list_head head;
    head.next = &head;
    head.prev = &head;
    return head;
}
```

Rust needs to prevent the equivalent code from compiling.

To find a solution in Rust, we take a look at the C solution:

```c
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```
A Problem with Initialization

Consider this *bad* piece of C code:

```c
struct list_head new_list_head(void) {
    struct list_head head;
    head.next = &head;
    head.prev = &head;
    return head;
}
```

Rust needs to prevent the equivalent code from compiling.

To find a solution in Rust, we take a look at the C solution:

```c
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```
A Problem with Initialization

Consider this *bad* piece of C code:

```c
struct list_head new_list_head(void) {
    struct list_head head;
    head.next = &head;
    head.prev = &head;
    return head;
}
```

▶ Rust needs to prevent the equivalent code from compiling.
A Problem with Initialization

Consider this bad piece of C code:

```c
struct list_head new_list_head(void) {
    struct list_head head;
    head.next = &head;
    head.prev = &head;
    return head;
}
```

- Rust needs to prevent the equivalent code from compiling.
- To find a solution in Rust, we take a look at the C solution:
A Problem with Initialization

Consider this *bad* piece of C code:

```c
struct list_head new_list_head(void) {
    struct list_head head;
    head.next = &head;
    head.prev = &head;
    return head;
}
```

▶ Rust needs to prevent the equivalent code from compiling.
▶ To find a solution in Rust, we take a look at the C solution:

```c
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}

Translating to Rust:

unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
A Problem with Initialization

```c
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```

Translating to Rust:

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```
A Problem with Initialization

```c
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```

Translating to Rust:

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```
A Problem with Initialization

```c
void init_list_head(struct list_head* head) {
    head->prev = head;
    head->next = head;
}
```

Translating to Rust:

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

But this requires unsafe code!
A Problem with Initialization

```rust
define unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```
A Problem with Initialization

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Problems with `unsafe`:

- Who ensures that `head` is a valid pointer?
A Problem with Initialization

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Problems with `unsafe`:

- Who ensures that `head` is a valid pointer?
- Still not ensured that `ListHead` stays/even is pinned.
A Problem with Initialization

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Problems with `unsafe`:
- Who ensures that `head` is a valid pointer?
- Still not ensured that `ListHead` stays/even is pinned.
- How to call this without `unsafe`?
A Problem with Initialization

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Problems with `unsafe`:

▶ Who ensures that `head` is a valid pointer?
▶ Still not ensured that `ListHead` stays/even is pinned.
▶ How to call this without `unsafe`?

Solution in C: use convention and rely on the programmer to do it correctly.
A Problem with Initialization

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Problems with `unsafe`:

- Who ensures that `head` is a valid pointer?
- Still not ensured that `ListHead` stays/even is pinned.
- How to call this without `unsafe`?

Solution in C: use convention and rely on the programmer to do it correctly.

Rust aims to offload most of this work to the compiler.
The Solution: `pin-init`

Turn this:

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Into this:

```rust
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```
The Solution: pin-init

Turn this:

```rust
unsafe fn init_list_head(head: *mut ListHead) {
    unsafe {
        (*head).prev = head;
        (*head).next = head;
    }
}
```

Into this:

```rust
fn new() -> impl PinInit<
```
The Solution: `pin-init`

```rust
def new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

The API guarantees:
- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization (i.e., it will have a stable address),
- No uninitialized memory can be used accidentally,
- The only way to initialize the struct is `pin-init`,
- No runtime cost: it is a zero-cost abstraction.

It is a feature-rich API, so if you need help just ask on Zulip:
https://rust-for-linux.zulipchat.com

Benno Lossin <benno.lossin@proton.me>
The Solution: pin-init

```rust
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!
The Solution: `pin-init`

```rust
fn new() -> impl PinInit<ListHead> {
    pin_init!(
        &this in ListHead {
            prev: this,
            next: this,
        }
    )
}
```

No `unsafe` to be found!
The API guarantees:

- All fields of the struct are initialized (none can be forgotten),
The Solution: `pin-init`

```rust
def new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!

The API guarantees:
- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization (i.e. it will have a stable address),
The Solution: pin-init

```rust
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!

The API guarantees:
- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization (i.e. it will have a stable address),
- No uninitialized memory can be used accidentally,
The Solution: pin-init

```rust
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No `unsafe` to be found!

The API guarantees:

- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization (i.e. it will have a stable address),
- No uninitialized memory can be used accidentally,
- The only way to initialize the struct is pin-init,
The Solution: pin-init

```rust
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No unsafe to be found!

The API guarantees:

- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization (i.e. it will have a stable address),
- No uninitialized memory can be used accidentally,
- The only way to initialize the struct is pin-init,
- No runtime cost: it is a zero-cost abstraction.
The Solution: pin-init

```rust
fn new() -> impl PinInit<ListHead> {
    pin_init!(&this in ListHead {
        prev: this,
        next: this,
    })
}
```

No **unsafe** to be found!

The API guarantees:

- All fields of the struct are initialized (none can be forgotten),
- The struct stays pinned after initialization
  (i.e. it will have a stable address),
- No uninitialized memory can be used accidentally,
- The only way to initialize the struct is `pin-init`,
- No runtime cost: it is a zero-cost abstraction.

It is a feature-rich API, so if you need help just ask on zulip:
https://rust-for-linux.zulipchat.com
The pin-init API in action

Code from the rust branch without the pin-init API:
The pin-init API in action

Code from the rust branch without the pin-init API:

```rust
let mut state = Pin::from(UniqueRef::try_new(Self {
    // SAFETY: `condvar_init!` is called below.
    state_changed: unsafe { CondVar::new() },
    // SAFETY: `mutex_init!` is called below.
    inner: unsafe { Mutex::new(SharedStateInner { token_count: 0 }) },
})?);

let pinned = unsafe {
    state.as_mut().map_unchecked_mut(|s| &mut s.state_changed)
};
kernel::condvar_init!(pinned, "SharedState::state_changed");

let pinned = unsafe {
    state.as_mut().map_unchecked_mut(|s| &mut s.inner)
};
kernel::mutex_init!(pinned, "SharedState::inner");

Ok(state.into())
```

Benno Lossin <benno.lossin@proton.me>
The pin-init API in action

Code from the rust branch without the pin-init API:

```rust
let mut state = Pin::from(UniqueRef::try_new(Self {
    // SAFETY: `condvar_init!` is called below.
    state_changed: unsafe { CondVar::new() },
    // SAFETY: `mutex_init!` is called below.
    inner: unsafe { Mutex::new(SharedStateInner { token_count: 0 }) },
});

// SAFETY: `state_changed` is pinned when `state` is.
let pinned = unsafe {
    state.as_mut().map_unchecked_mut(|s| &mut s.state_changed)
};
kernel::condvar_init!(pinned, "SharedState::state_changed");

// SAFETY: `inner` is pinned when `state` is.
let pinned = unsafe {
    state.as_mut().map_unchecked_mut(|s| &mut s.inner)
};
kernell::mutex_init!(pinned, "SharedState::inner");
Ok(state.into())
```

This requires unsafe code!

Benno Lossin <benno.lossin@proton.me>
The pin-init API in action

Improved code with the pin-init API:

```rust
pin_init!(Self {
    state_changed <- new_condvar!("SharedState::state_changed"),
    inner <- new_mutex!(
        SharedStateInner { token_count: 0 },
        "SharedState::Inner",
    ),
})
```

No unsafe to be found!
Initialization and Address Stability
A Motivation for Rust
Address Stability in the Kernel
Address Stability Support in Rust
A Problem with Initialization
The Solution: pin-init

Field Projections
Field Projections
The Problem with Pin<P>
A possible solution: Pin-Projections
Other Kinds of Field Projections
Field Projections

- Having a pointer `&mut Struct` to a struct:

```rust
define Struct {
    field: Field,
}
```

- Turning that pointer into a pointer "of the same type" to a field of that struct:

```
&mut Struct ⇝ &mut Field
```

- This is possible in Rust:

```
&mut my_struct.field
```

- In Rust a different pointer type can carry additional information:

```
&mut MaybeUninit<Struct>
```

- Points to a possibly uninitialized `Struct` (and only provides `unsafe` access)

```
&mut MaybeUninit<Struct> ⇝ &mut MaybeUninit<Field>
```

- But this is not (safely) possible in Rust at the moment
Field Projections

- Having a pointer `&mut Struct` to a struct:

  ```rust
  struct Struct {
    field: Field,
  }
  ```

- Turning that pointer into a pointer “of the same type” to a field of that struct:
Field Projections

- Having a pointer `&mut Struct` to a struct:

  ```rust
  struct Struct {
    field: Field,
  }
  ```

- Turning that pointer into a pointer “of the same type” to a field of that struct:

  ```rust
  &mut Struct ~→ &mut Field
  ```
Field Projections

- Having a pointer `&mut Struct` to a struct:

```rust
default Struct {
    field: Field,
}
```

- Turning that pointer into a pointer “of the same type” to a field of that struct:

  ```rust
  &mut Struct ⇝ &mut Field
  ```

- This is possible in Rust: `&mut my_struct.field`
Field Projections

- Having a pointer \&mut Struct to a struct:

```rust
struct Struct {
    field: Field,
}
```

- Turning that pointer into a pointer “of the same type” to a field of that struct:

```rust
\&mut Struct \mapsto \&mut Field
```

- This is possible in Rust: \&mut my_struct.field

In Rust a different pointer type can carry additional information:
Field Projections

- Having a pointer `&mut Struct` to a struct:

```rust
code
struct Struct {
    field: Field,
}
```

- Turning that pointer into a pointer “of the same type” to a field of that struct:

```rust
code
&mut Struct ⇝ &mut Field
```

- This is possible in Rust: `&mut my_struct.field`

In Rust a different pointer type can carry additional information:

- `&mut MaybeUninit<Struct>` points to a possibly uninitialized `Struct` (and only provides `unsafe` access)
Field Projections

- Having a pointer `&mut Struct` to a struct:

```rust
group Struct {
    field: Field,
}
```

- Turning that pointer into a pointer “of the same type” to a field of that struct:

```rust
&mut Struct ⇝ &mut Field
```

- This is possible in Rust: `&mut my_struct.field`

In Rust a different pointer type can carry additional information:

```
&mut MaybeUninit<Struct> points to a possibly uninitialized Struct (and only provides `unsafe` access)
```

```
natural projection:

&mut MaybeUninit<Struct> ⇝ &mut MaybeUninit<Field>
```
Field Projections

- Having a pointer `&mut Struct` to a struct:

```rust
class Struct {
    field: Field,
}
```

- Turning that pointer into a pointer “of the same type” to a field of that struct:

```rust
&mut Struct ↦ &mut Field
```

- This is possible in Rust: `&mut my_struct.field`

In Rust a different pointer type can carry additional information:

- `&mut MaybeUninit<Struct>` points to a possibly uninitialized `Struct` (and only provides `unsafe` access) natural projection:

```rust
&mut MaybeUninit<Struct> ↦ &mut MaybeUninit<Field>
```

- But this is not (safely) possible in Rust at the moment
The Problem with `Pin<P>`

- All mutating functions on the pinned type need to take `Pin<&mut Self>`.

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

`Foo` needs to be pinned because of the `ListHead` field.

How to modify `count`?
The Problem with `Pin<P>`

- All mutating functions on the pinned type need to take `Pin<&mut Self>`.
- Remember: no access to `&mut Self` allowed (because `swap` exists), so how can one modify the fields?

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

`Foo` needs to be pinned because of the `ListHead` field. How to modify `count`?
The Problem with `Pin<P>`

- All mutating functions on the pinned type need to take `Pin<&mut Self>`.
- Remember: no access to `&mut Self` allowed (because `swap` exists), so how can one modify the fields?
- For example:

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```
The Problem with Pin<P>

▶ All mutating functions on the pinned type need to take Pin<&mut Self>.
▶ Remember: no access to &mut Self allowed (because swap exists), so how can one modify the fields?
▶ For example:

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

▶ Foo needs to be pinned because of the ListHead field.
The Problem with `Pin<P>`

- All mutating functions on the pinned type need to take `Pin<&mut Self>.
- Remember: no access to `&mut Self` allowed (because `swap` exists), so how can one modify the fields?
- For example:

```
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

- Foo needs to be pinned because of the `ListHead` field.
- How to modify `count`?
A possible solution: Pin-Projections

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`) ⇒ allow access via `&mut usize`
2. Fields that require to be pinned (like `list_head`) ⇒ only allow access via `Pin<&mut ListHead>`

▶ Special case of field projections:

`Pin<&mut Foo> ⇝ &mut usize`

`Pin<&mut Foo> ⇝ Pin<&mut ListHead>`

▶ These are called pin projections, they depend on the "intended usecase" of the field and are determined on a field by field basis.
A possible solution: Pin-Projections

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`) ⇒ allow access via `&mut usize`
2. Fields that require to be pinned (like `list_head`) ⇒ only allow access via `Pin<&mut ListHead>`

▶ Special case of field projections:

- `Pin<&mut Foo> ⇝ &mut usize`
- `Pin<&mut Foo> ⇝ Pin<&mut ListHead>`

These are called pin projections, they depend on the “intended use case” of the field and are determined on a field by field basis.
A possible solution: Pin-Projections

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)

Benno Lossin <benno.lossin@proton.me>
A possible solution: Pin-Projections

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)

2. Fields that require to be pinned (like `list_head`)

Benno Lossin <benno.lossin@proton.me>
A possible solution: Pin-Projections

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)
   ⇒ allow access via `&mut usize`
2. Fields that require to be pinned (like `list_head`)
   ⇒ only allow access via `Pin<&mut ListHead>`

▶ Special case of field projections:

- `Pin<&mut Foo> ⇝ &mut usize`
- `Pin<&mut Foo> ⇝ Pin<&mut ListHead>`

These are called pin projections, they depend on the "intended use case" of the field and are determined on a field by field basis.
A possible solution: Pin-Projections

```rust
class Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)
   \[\Rightarrow\text{allow access via } &\text{mut }\text{ usize}\]

2. Fields that require to be pinned (like `list_head`)
   \[\Rightarrow\text{only allow access via } \text{Pin}<&\text{mut }\text{ ListHead}>\]
A possible solution: Pin-Projections

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)
   \[\rightarrow\] allow access via `&mut usize`

2. Fields that require to be pinned (like `list_head`)
   \[\rightarrow\] only allow access via `Pin<&mut ListHead>`

▶ Special case of field projections:

- `Pin<&mut Foo> \rightarrow &mut usize`
- `Pin<&mut Foo> \rightarrow Pin<&mut ListHead>`
A possible solution: Pin-Projections

```rust
struct Foo {
    list_head: ListHead,
    count: usize,
}
```

Observe that there are two types of fields:

1. Fields that do not require to be pinned (like `count`)
   \[\Rightarrow\] allow access via `&mut usize`

2. Fields that require to be pinned (like `list_head`)
   \[\Rightarrow\] only allow access via `Pin<&mut ListHead>`

▶ Special case of field projections:

\[
\text{Pin<&mut Foo>} \leftrightarrow &\text{mut usize}
\]

\[
\text{Pin<&mut Foo>} \leftrightarrow \text{Pin<&mut ListHead>}
\]

▶ These are called pin projections, they depend on the “intended usecase” of the field and are determined on a field by field basis.
Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
- `VolatileMem<T>`: all memory must be accessed by volatile operations
  ⇒ the fields also have to be accessed in a volatile manner. Therefore we can allow projections: `&mut VolatileMem<Struct>` ⇝ `&mut VolatileMem<Field>`
- Another important usage would be for field access via raw pointers:
  - Allow projecting: `*mut Struct` ⇝ `*mut Field`
  - Improve ergonomics by introducing projection operator `->`: `foo->bar` instead of `unsafe { addr_of!((*foo).bar) }
- RFC for adding general field projection support to Rust: [http://github.com/rust-lang/rfcs/pull/3318](http://github.com/rust-lang/rfcs/pull/3318)
Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- **MaybeUninit<T>:** as shown before
- **VolatileMem<T>:** all memory must be accessed by volatile operations

Another important usage would be for field access via raw pointers:

- Allow projecting:
  
  ```
  *mut Struct ⇝ *mut Field
  ```

  Improve ergonomics by introducing projection operator `->`:

  ```
  foo->bar
  ```

  instead of

  ```
  unsafe { addr_of!((*foo).bar) }
  ```

RFC for adding general field projection support to Rust: [http://github.com/rust-lang/rfcs/pull/3318](http://github.com/rust-lang/rfcs/pull/3318)
Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- **MaybeUninit<T>:** as shown before
- **VolatileMem<T>:** all memory must be accessed by volatile operations ➞ the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

  &mut VolatileMem<Struct> ⇝ &mut VolatileMem<Field>

Another important usage would be for field access via raw pointers:

- Allow projecting:
  
  *mut Struct ⇝ *mut Field

- Improve ergonomics by introducing projection operator ➞:
  
  foo->bar instead of unsafe { addr_of!((*foo).bar) }

RFC for adding general field projection support to Rust:

http://github.com/rust-lang/rfcs/pull/3318

Benno Lossin <benno.lossin@proton.me>
Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

▶ MaybeUninit<T>: as shown before
▶ VolatileMem<T>: all memory must be accessed by volatile operations ⟷ the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

\[
&\text{mut VolatileMem<Struct>} \leftrightarrow &\text{mut VolatileMem<Field>}
\]
Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
- `VolatileMem<T>`: all memory must be accessed by volatile operations → the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

  ```
  &mut VolatileMem<Struct> ↦&mut VolatileMem<Field>
  ```

Another important usage would be for field access via raw pointers:

```
Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- **MaybeUninit<T>:** as shown before
- **VolatileMem<T>:** all memory must be accessed by volatile operations → the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

  ```rust
  &mut VolatileMem<Struct> ↦ &mut VolatileMem<Field>
  ```

Another important usage would be for field access via raw pointers:

- Allow projecting:

  ```rust
  *mut Struct ↦ *mut Field
  ```
Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- **MaybeUninit<T>:** as shown before
- **VolatileMem<T>:** all memory must be accessed by volatile operations → the fields also have to be accessed in a volatile manner. Therefore we can allow projections:

  ```rust
c &mut VolatileMem<Struct> ↦ &mut VolatileMem<Field>
  ```

Another important usage would be for field access via raw pointers:

- Allow projecting:

  ```rust
  *mut Struct ↦ *mut Field
  ```

- Improve ergonomics by introducing projection operator ->:

  ```rust
  foo->bar instead of unsafe { addr_of!((*foo).bar) }
  ```

Benno Lossin <benno.lossin@proton.me>
Other Kinds of Projections

The concept of projecting a value is could be useful in many other situations:

- `MaybeUninit<T>`: as shown before
- `VolatileMem<T>`: all memory must be accessed by volatile operations → the fields also have to be accessed in a volatile manner. Therefore we can allow projections:
  
  ```rust
  &mut VolatileMem<Struct> ⇝ &mut VolatileMem<Field>
  ```

Another important usage would be for field access via raw pointers:

- Allow projecting:
  ```rust
  *mut Struct ⇝ *mut Field
  ```
- Improve ergonomics by introducing projection operator ->:
  ```rust
  foo->bar instead of unsafe { addr_of!((*foo).bar) }
  ```
- RFC for adding general field projection support to Rust:
Thanks for Your Attention!

Follow my work:

► RFC for adding general field projection support to Rust:
  https://github.com/rust-lang/rfcs/pull/3318

► pin-init userspace library:
  https://github.com/Rust-for-Linux/pinned-init

Contact me on:

► https://rust-for-linux.zulipchat.com