Introduction to Rust

Quality of Life Beyond Memory Safety

Benno Lossin <benno.lossin@proton.me>

Rust-for-Linux Core Team

Linux Plumbers Conference 18. September 2024 Vienna

Goals of Rust

Goals of Rust

- Reduce number of logic bugs,

Goals of Rust

- Reduce number of logic bugs,
- Relieve reviewers of mindless checks,

Goals of Rust

- Reduce number of logic bugs,
- Relieve reviewers of mindless checks,
- Simplify work of driver authors.

Goals of Rust

- Reduce number of logic bugs,
- Relieve reviewers of mindless checks,
- Simplify work of driver authors.

Want to achieve these goals through:

Goals of Rust

- Reduce number of logic bugs,
- Relieve reviewers of mindless checks,
- Simplify work of driver authors.

Want to achieve these goals through:

- Replacing manual coding conventions by compiler-enforced guardrails,

Goals of Rust

- Reduce number of logic bugs,
- Relieve reviewers of mindless checks,
- Simplify work of driver authors.

Want to achieve these goals through:

- Replacing manual coding conventions by compiler-enforced guardrails,
- Designing robust APIs,

Goals of Rust

- Reduce number of logic bugs,
- Relieve reviewers of mindless checks,
- Simplify work of driver authors.

Want to achieve these goals through:

- Replacing manual coding conventions by compiler-enforced guardrails,
- Designing robust APIs,
- Good documentation.

Outline

Rust Basics

```
struct foo {
    int value;
};

int foo_value(const struct foo* foo) {
    return foo->value;
}
```

Rust Basics

```c
struct foo {
    int value;
};

int foo_value(const struct foo* foo) {
    return foo->value;
}
```

```rust
struct Foo {
    value: i32,
}
```

Rust Basics

```c
struct foo {
    int value;
};

int foo_value(const struct foo* foo) {
    return foo->value;
}
```

```rust
struct Foo {
    value: i32,
}

impl Foo {
    fn new() -> Self {
        Self { value: 42 }
    }
}
```

Rust Basics

```c
struct foo {
    int value;
};

int foo_value(const struct foo* foo) {
    return foo->value;
}
```

```rust
struct Foo {
    value: i32,
}

impl Foo {
    fn new() -> Self {
        Self { value: 42 }
    }
}

let foo = Foo::new();
```

Rust Basics

```c
struct foo {
    int value;
};

int foo_value(const struct foo* foo) {
    return foo->value;
}
```

```rust
struct Foo {
    value: i32,
}

impl Foo {
    fn new() -> Self {
        Self { value: 42 }
    }
}

let foo = Foo::new();
```

```rust
struct Foo(i32);

impl Foo{
    fn new() -> Self {
        Self(42)
    }
}
```

Rust Basics

```c
struct foo {
    int value;
};

int foo_value(const struct foo* foo) {
    return foo->value;
}
```

```rust
struct Foo {
    value: i32,
}

impl Foo {
    fn new() -> Self {
        Self { value: 42 }
    }
}

let foo = Foo::new();
```

```rust
struct Foo(i32);

impl Foo{
    fn new() -> Self {
        Self(42)
    }

    fn value(&self) -> i32 {
        self.0
    }
}
```

Rust Basics

```c
struct foo {
    int value;
};

int foo_value(const struct foo* foo) {
    return foo->value;
}
```

```rust
struct Foo {
    value: i32,
}

impl Foo {
    fn new() -> Self {
        Self { value: 42 }
    }
}

let foo = Foo::new();
```

```rust
struct Foo(i32);

impl Foo{
    fn new() -> Self {
        Self(42)
    }

    fn value(self: &Self) -> i32 {
        self.0
    }
}
```

Rust Basics

```c
struct foo {
    int value;
};

int foo_value(const struct foo* foo) {
    return foo->value;
}
```

```rust
struct Foo {
    value: i32,
}

impl Foo {
    fn new() -> Self {
        Self { value: 42 }
    }
}

let foo = Foo::new();
```

```rust
struct Foo(i32);

impl Foo{
    fn new() -> Self {
        Self(42)
    }

    fn value(self: &Self) -> i32 {
        self.0
    }
}

let value = foo.value();
```

Rust Basics

```c
struct foo {
    int value;
};

int foo_value(const struct foo* foo) {
    return foo->value;
}
```

```rust
struct Foo {
    value: i32,
}

impl Foo {
    fn new() -> Self {
        Self { value: 42 }
    }
}

let foo = Foo::new();
```

```rust
struct Foo(i32);

impl Foo{
    fn new() -> Self {
        Self(42)
    }

    fn value(self: &Self) -> i32 {
        self.0
    }
}

let value = foo.value();
```

For other basics, see [1].

Enums in C and Rust

```c
enum state {
    DISABLED = 0,
    ACTIVE,

};
```

Enums in C and Rust

```c
enum state {
    DISABLED = 0,
    ACTIVE,

};

char* name(const enum state* state) {
    switch (*state) {
        case DISABLED:
            return "disabled";
        case ACTIVE:
            return "active";
        default:
            return "unknown";
    }
}

void do_work(enum state* state);
```

Enums in C and Rust

```c
enum state {
    DISABLED = 0,
    ACTIVE,

};

char* name(const enum state* state) {
    switch (*state) {
        case DISABLED:
            return "disabled";
        case ACTIVE:
            return "active";
        default:
            return "unknown";
    }
}

void do_work(enum state* state);
```

```rust
enum State {
    Disabled,

    Active,
}
```

Enums in C and Rust

```c
enum state {
    DISABLED = 0,
    ACTIVE,

};

char* name(const enum state* state) {
    switch (*state) {
        case DISABLED:
            return "disabled";
        case ACTIVE:
            return "active";
        default:
            return "unknown";
    }
}

void do_work(enum state* state);
```

```rust
enum State {
    Disabled,

    Active,
}

fn name(state: &State) -> &str {
    match state {

        State::Disabled => "disabled",
        State::Active => "active",
    }
}

fn do_work(state: &mut State);
```

Enums in C and Rust

```c
enum state {
    DISABLED = 0,
    ACTIVE,

};

char* name(const enum state* state) {
    switch (*state) {
        case DISABLED:
            return "disabled";
        case ACTIVE:
            return "active";
        default:
            return "unknown";
    }
}

void do_work(enum state* state);
```

```rust
enum State {
    Disabled,

    Active,
}

fn name(state: &State) -> &str {
    match state {

        State::Disabled => "disabled",
        State::Active => "active",
    }
}

fn do_work(state: &mut State);
```

Need to introduce a new state: `waiting`.

Enums in C and Rust

```c
enum state {
    DISABLED = 0,
    ACTIVE,
    WAITING,
};

char* name(const enum state* state) {
    switch (*state) {
        case DISABLED:
            return "disabled";
        case ACTIVE:
            return "active";
        default:
            return "unknown";
    }
}

void do_work(enum state* state);
```

```rust
enum State {
    Disabled,
    Waiting,
    Active,
}

fn name(state: &State) -> &str {
    match state {

        State::Disabled => "disabled",
        State::Active => "active",
    }
}

fn do_work(state: &mut State);
```

Enums in C and Rust

```c
enum state {
    DISABLED = 0,
    ACTIVE,
    WAITING,
};

char* name(const enum state* state) {
    switch (*state) {
        case DISABLED:
            return "disabled";
        case ACTIVE:
            return "active";
        default:
            return "unknown";
    }
}

void do_work(enum state* state);
```

```rust
enum State {
    Disabled,
    Waiting,
    Active,
}

fn name(state: &State) -> &str {
    match state {
          ^^^^^
  pattern `&State::Waiting` not covered
          State::Disabled => "disabled",
          State::Active => "active",
    }
}

fn do_work(state: &mut State);
```

Enums in C and Rust

```c
enum state {
    DISABLED = 0,
    ACTIVE,
    WAITING,
};

char* name(const enum state* state) {
    switch (*state) {
        case DISABLED:
            return "disabled";
        case ACTIVE:
            return "active";
        default:
            return "unknown";
    }
}

void do_work(enum state* state);
```

```rust
enum State {
    Disabled,
    Waiting,
    Active,
}

fn name(state: &State) -> &str {
    match state {

        State::Waiting => "waiting",
        State::Disabled => "disabled",
        State::Active => "active",
    }
}

fn do_work(state: &mut State);
```

Enums Carrying Data

Add a reason for waiting:

Enums Carrying Data

Add a reason for waiting:

```
struct state {
    enum state_kind kind;
    enum wait_reason wait_reason;
};

enum state_kind {
    DISABLED = 0,
    ACTIVE,
    WAITING,
};

enum wait_reason {
    USER = 0,
    SYSTEM,
    PROCESSING,
};
```

Enums Carrying Data

Add a reason for waiting:

```
struct state {
    enum state_kind kind;
    enum wait_reason wait_reason;
};

enum state_kind {
    DISABLED = 0,
    ACTIVE,
    WAITING,
};

enum wait_reason {
    USER = 0,
    SYSTEM,
    PROCESSING,
};
```

```
enum State {
    Disabled,
    Waiting(WaitReason),
    Active,
}

enum WaitReason {
    User,
    System,
    Processing,
}
```

Enums Carrying Data

Add a reason for waiting:

```
struct state {
    enum state_kind kind;
    enum wait_reason wait_reason;
};

enum state_kind {
    DISABLED = 0,
    ACTIVE,
    WAITING,
};

enum wait_reason {
    USER = 0,
    SYSTEM,
    PROCESSING,
};
```

```
enum State {
    Disabled,
    Waiting(WaitReason),
    Active,
}

enum WaitReason {
    User,
    System,
    Processing,
}

fn do_work(state: &mut State) {
    match *state {
        State::Active => /* ... */
        State::Waiting(reason) => /* ... */
        State::Disabled => /* ... */
    }
}
```

Important Predefined Enums

Important Predefined Enums

```
enum Option<T> {
    None,
    Some(T),
}
```

Important Predefined Enums

```
enum Option<T> {
    None,
    Some(T),
}
```

- `Option<T>` is a nullable `T`,

Important Predefined Enums

```
enum Option<T> {
    None,
    Some(T),
}
```

- `Option<T>` is a nullable `T`,
- Must check if there is a value in order to use the value[1],

[1] actually, there are `unsafe` and panicking functions that allow direct access

Important Predefined Enums

```
enum Option<T> {
    None,
    Some(T),
}
```

- `Option<T>` is a nullable `T`,
- Must check if there is a value in order to use the value[1],
- Guaranteed optimizations: `Option<&mut T>` is pointer-sized,

[1] actually, there are `unsafe` and panicking functions that allow direct access

Important Predefined Enums

```
enum Option<T> {
    None,
    Some(T),
}
```

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- `Option<T>` is a nullable `T`,
- Must check if there is a value in order to use the value[1],
- Guaranteed optimizations: `Option<&mut T>` is pointer-sized,

[1] actually, there are `unsafe` and panicking functions that allow direct access

Important Predefined Enums

```
enum Option<T> {
    None,
    Some(T),
}
```

- `Option<T>` is a nullable `T`,
- Must check if there is a value in order to use the value[1],
- Guaranteed optimizations: `Option<&mut T>` is pointer-sized,

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- Is either a `T` or an error of type `E`,

[1] actually, there are `unsafe` and panicking functions that allow direct access

Important Predefined Enums

```
enum Option<T> {
    None,
    Some(T),
}
```

- `Option<T>` is a nullable `T`,
- Must check if there is a value in order to use the value[1],
- Guaranteed optimizations: `Option<&mut T>` is pointer-sized,

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- Is either a `T` or an error of type `E`,
- Cannot forget to check the error,

[1] actually, there are `unsafe` and panicking functions that allow direct access

Important Predefined Enums

```
enum Option<T> {
    None,
    Some(T),
}
```

- `Option<T>` is a nullable `T`,
- Must check if there is a value in order to use the value[1],
- Guaranteed optimizations: `Option<&mut T>` is pointer-sized,

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- Is either a `T` or an error of type `E`,
- Cannot forget to check the error,
- Similar to the `ERR_PTR` API.

[1] actually, there are `unsafe` and panicking functions that allow direct access

Important Predefined Enums

```
enum Option<T> {
    None,
    Some(T),
}
```

- `Option<T>` is a nullable `T`,
- Must check if there is a value in order to use the value[1],
- Guaranteed optimizations: `Option<&mut T>` is pointer-sized,

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

- Is either a `T` or an error of type `E`,
- Cannot forget to check the error,
- Similar to the `ERR_PTR` API.

For more on enums, see [2]. For error handling, see [3].

[1] actually, there are `unsafe` and panicking functions that allow direct access

Visibility in Rust

Visibility in Rust

- Everything is private by default,

Visibility in Rust

- Everything is private by default,
- Explicitly opt in to exporting items:

Visibility in Rust

- Everything is private by default,
- Explicitly opt in to exporting items:
  - `pub` for a global export,

Visibility in Rust

- Everything is private by default,
- Explicitly opt in to exporting items:
  - `pub` for a global export,
  - `pub(crate)` to only allow the crate itself to access the item.

Visibility in Rust

- Everything is private by default,
- Explicitly opt in to exporting items:
  - `pub` for a global export,
  - `pub(crate)` to only allow the crate itself to access the item.
- Structs can only be constructed directly if all fields are visible from the current module,

Visibility in Rust

- Everything is private by default,
- Explicitly opt in to exporting items:
  - `pub` for a global export,
  - `pub(crate)` to only allow the crate itself to access the item.
- Structs can only be constructed directly if all fields are visible from the current module,

```
pub mod cfg {
    pub struct Config {
        pub size: usize,
        name: String,
    }

}
```

Visibility in Rust

- Everything is private by default,
- Explicitly opt in to exporting items:
  - `pub` for a global export,
  - `pub(crate)` to only allow the crate itself to access the item.
- Structs can only be constructed directly if all fields are visible from the current module,

```rust
pub mod cfg {
    pub struct Config {
        pub size: usize,
        name: String,
    }

}
```

```rust
pub(crate) mod cfg_user {
    use crate::cfg::Config;

    pub fn my_config() -> Config {
        let size = 100;
        let name = format!("MyConfig");
        Config { size, name }
    }
}
```

Visibility in Rust

- Everything is private by default,
- Explicitly opt in to exporting items:
  - `pub` for a global export,
  - `pub(crate)` to only allow the crate itself to access the item.
- Structs can only be constructed directly if all fields are visible from the current module,

```
pub mod cfg {
    pub struct Config {
        pub size: usize,
        name: String,
    }



}
```

```
pub(crate) mod cfg_user {
    use crate::cfg::Config;

    pub fn my_config() -> Config {
        let size = 100;
        let name = format!("MyConfig");
        Config { size, name }
    }                  ^^^^
}          field `name` of struct
           cfg::Config is private
```

Visibility in Rust

- Everything is private by default,
- Explicitly opt in to exporting items:
  - `pub` for a global export,
  - `pub(crate)` to only allow the crate itself to access the item.
- Structs can only be constructed directly if all fields are visible from the current module,

```
pub mod cfg {
    pub struct Config {
        pub size: usize,
        name: String,
    }
    impl Config {
        pub fn new(
            size: usize,
            name: String,
        ) -> Self {
            Self { size, name }
        }
    }
}
```

```
pub(crate) mod cfg_user {
    use crate::cfg::Config;

    pub fn my_config() -> Config {
        let size = 100;
        let name = format!("MyConfig");
        Config { size, name }
    }                      ^^^^
}           field `name` of struct
            cfg::Config is private
```

Visibility in Rust

- Everything is private by default,
- Explicitly opt in to exporting items:
  - `pub` for a global export,
  - `pub(crate)` to only allow the crate itself to access the item.
- Structs can only be constructed directly if all fields are visible from the current module,

```rust
pub mod cfg {
    pub struct Config {
        pub size: usize,
        name: String,
    }
    impl Config {
        pub fn new(
            size: usize,
            name: String,
        ) -> Self {
            Self { size, name }
        }
    }
}
```

```rust
pub(crate) mod cfg_user {
    use crate::cfg::Config;

    pub fn my_config() -> Config {
        let size = 100;
        let name = format!("MyConfig");
        Config::new(size, name)
    }
}
```

Newtype Pattern

- Wrap an existing type in a new struct with the single field being private,

Newtype Pattern

- Wrap an existing type in a new struct with the single field being private,
- Explicitly choose which operations are exported,

Newtype Pattern

- Wrap an existing type in a new struct with the single field being private,
- Explicitly choose which operations are exported,
- Can add new functions and methods.

Newtype Pattern

- Wrap an existing type in a new struct with the single field being private,
- Explicitly choose which operations are exported,
- Can add new functions and methods.

```
pub struct Handle {
    /* ... */
}

impl Handle {
    pub fn foo(&self) {
        /* ... */
    }
    pub fn bar(&self) {
        /* ... */
    }
}
```

Newtype Pattern

- Wrap an existing type in a new struct with the single field being private,
- Explicitly choose which operations are exported,
- Can add new functions and methods.

```rust
pub struct Handle {
    /* ... */
}

impl Handle {
    pub fn foo(&self) {
        /* ... */
    }
    pub fn bar(&self) {
        /* ... */
    }
}
```

```rust
pub struct DebugHandle(Handle);

impl DebugHandle {
    pub fn foo(&self) {
        self.0.foo();
    }
    // `bar` not exposed
}
```

Newtype Pattern

- Wrap an existing type in a new struct with the single field being private,
- Explicitly choose which operations are exported,
- Can add new functions and methods.

```rust
pub struct Handle {
    /* ... */
}

impl Handle {
    pub fn foo(&self) {
        /* ... */
    }
    pub fn bar(&self) {
        /* ... */
    }
}
```

```rust
pub struct DebugHandle(Handle);

impl DebugHandle {
    pub fn foo(&self) {
        self.0.foo();
    }
    // `bar` not exposed
}
```

Read more at [4].

Untrusted Data

Untrusted Data

- Mark external data (e.g. from hardware or userspace) as untrusted,

Untrusted Data

- Mark external data (e.g. from hardware or userspace) as untrusted,
- Don't allow people to access it without validating it first,

Untrusted Data

- Mark external data (e.g. from hardware or userspace) as untrusted,
- Don't allow people to access it without validating it first,
- Use the newtype pattern for this:

Untrusted Data

- Mark external data (e.g. from hardware or userspace) as untrusted,
- Don't allow people to access it without validating it first,
- Use the newtype pattern for this:

```
pub struct Untrusted<T>(T);

impl<T> Untrusted<T> {
    pub fn new_untrusted(value: T) -> Self {
        Untrusted(value)
    }

    /* no way to access the inner value publicly */
```

Untrusted Data

- Mark external data (e.g. from hardware or userspace) as untrusted,
- Don't allow people to access it without validating it first,
- Use the newtype pattern for this:

```
pub struct Untrusted<T>(T);

impl<T> Untrusted<T> {
    pub fn new_untrusted(value: T) -> Self {
        Untrusted(value)
    }

    /* no way to access the inner value publicly */

    pub fn validate<V: Validate<T>>(self) -> Result<V, Error> {
        V::validate(self.0)
    }
}
```

Patch on the LKML: [5].

Traits

- “Traits: Defining Shared Behavior” [6].

Traits

- “Traits: Defining Shared Behavior” [6].
- Require that generic types offer certain operations,

Traits

- “Traits: Defining Shared Behavior” [6].
- Require that generic types offer certain operations,

```
pub trait Validate<T> {
    fn validate(untrusted: T) -> Result<Self, Error>;
}
```

Traits

- “Traits: Defining Shared Behavior” [6].
- Require that generic types offer certain operations,

```
pub trait Validate<T> {
    fn validate(untrusted: T) -> Result<Self, Error>;
}

impl Validate<&[u8]> for MyType {
    fn validate(untrusted: &[u8]) -> Result<Self, Error> {
        /* ... */
    }
}
```

Traits

- “Traits: Defining Shared Behavior” [6].
- Require that generic types offer certain operations,

```
pub trait Validate<T> {
    fn validate(untrusted: T) -> Result<Self, Error>;
}

impl Validate<&[u8]> for MyType {
    fn validate(untrusted: &[u8]) -> Result<Self, Error> {
        /* ... */
    }
}
```

There are special traits:

- Send for marking types which you can send between threads [7],

Traits

- “Traits: Defining Shared Behavior” [6].
- Require that generic types offer certain operations,

```
pub trait Validate<T> {
    fn validate(untrusted: T) -> Result<Self, Error>;
}

impl Validate<&[u8]> for MyType {
    fn validate(untrusted: &[u8]) -> Result<Self, Error> {
        /* ... */
    }
}
```

There are special traits:

- `Send` for marking types which you can send between threads [7],
- `Sync` for marking types which you can share between threads [7],

Traits

- "Traits: Defining Shared Behavior" [6].
- Require that generic types offer certain operations,

```
pub trait Validate<T> {
    fn validate(untrusted: T) -> Result<Self, Error>;
}

impl Validate<&[u8]> for MyType {
    fn validate(untrusted: &[u8]) -> Result<Self, Error> {
        /* ... */
    }
}
```

There are special traits:

- `Send` for marking types which you can send between threads [7],
- `Sync` for marking types which you can share between threads [7],
- Using them we can prevent that a lock is unlocked on a different thread,

Further Concepts

- Ownership [8]

Further Concepts

- Ownership [8]
- Lifetimes and borrowing [9]

Further Concepts

- Ownership [8]
- Lifetimes and borrowing [9]
- Declarative and procedural macros [10]

Further Concepts

- Ownership [8]
- Lifetimes and borrowing [9]
- Declarative and procedural macros [10]
- Pinning (data with a stable address) [11]

Further Concepts

- Ownership [8]
- Lifetimes and borrowing [9]
- Declarative and procedural macros [10]
- Pinning (data with a stable address) [11]
- Documentation [12]

References

[1] URL: https://doc.rust-lang.org/stable/book/ch03-00-common-programming-concepts.html.

[2] URL: https://doc.rust-lang.org/stable/book/ch06-00-enums.html.

[3] URL: https://doc.rust-lang.org/stable/book/ch09-02-recoverable-errors-with-result.html.

[4] URL: https://doc.rust-lang.org/stable/book/ch19-04-advanced-types.html#using-the-newtype-pattern-for-type-safety-and-abstraction.

[5] URL: https://lore.kernel.org/rust-for-linux/20240913112643.542914-1-benno.lossin@proton.me/.

[6] URL: https://doc.rust-lang.org/stable/book/ch10-02-traits.html.

[7] URL: https://doc.rust-lang.org/stable/book/ch16-04-extensible-concurrency-sync-and-send.html.

[8] URL: https://doc.rust-lang.org/stable/book/ch04-00-understanding-ownership.html.

[9] URL: https://doc.rust-lang.org/stable/book/ch10-03-lifetime-syntax.html.

[10] URL: https://doc.rust-lang.org/stable/book/ch19-06-macros.html.

[11] URL: https://doc.rust-lang.org/std/pin/index.html.

[12] URL: https://doc.rust-lang.org/stable/book/ch14-02-publishing-to-crates-io.html.