Concepts in Rust

Concepts in Rust and Why

Ivan Tham (pickfire)

Rust

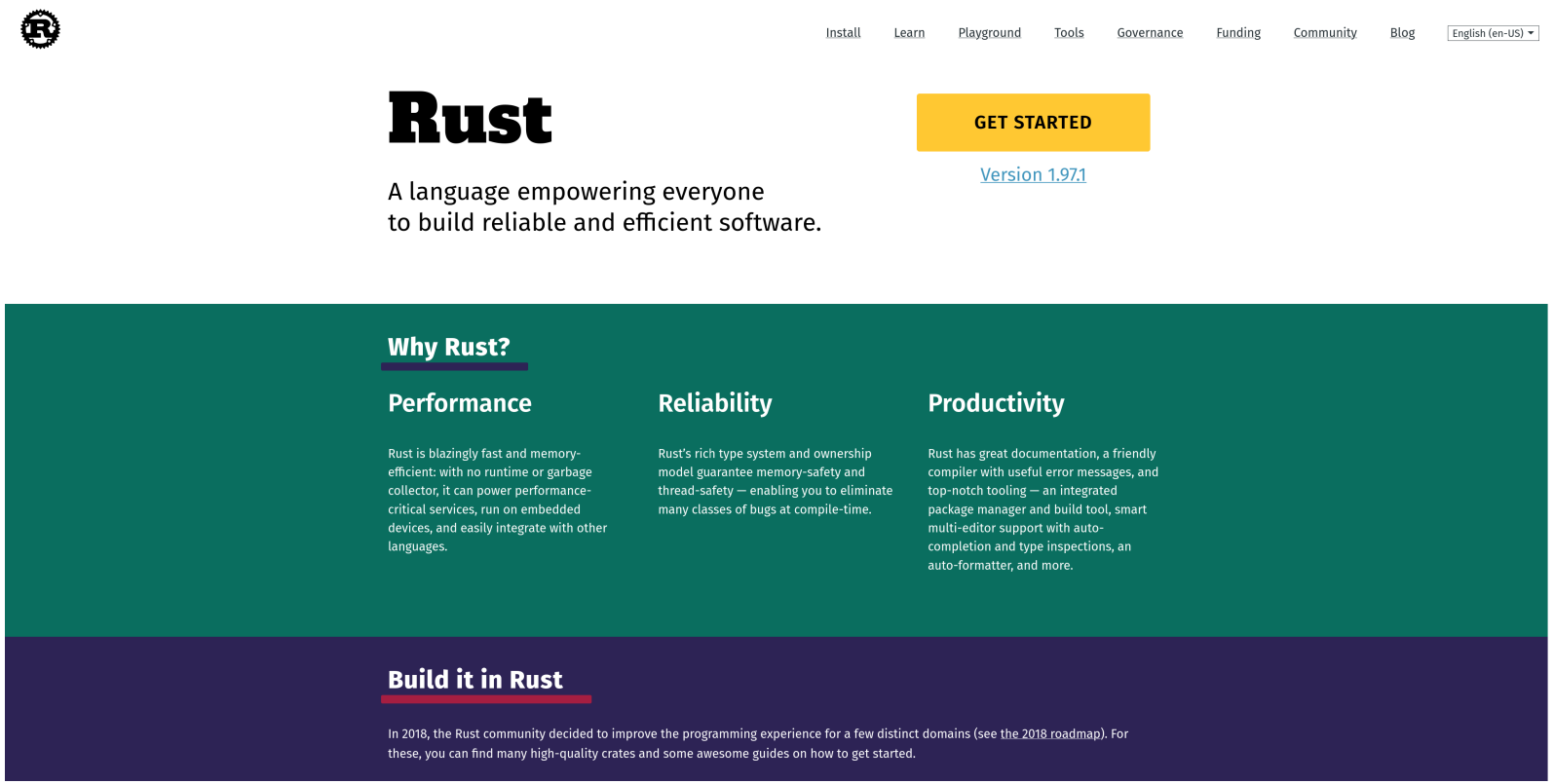
A language empowering everyone to build **reliable** and **efficient** software.

Rust

A language empowering everyone to build **reliable** and **efficient** software.

Why? How?

Rust


Install
Learn
Playground
Tools
Governance
Funding
Community
Blog
English (en-US)
Rust
A language empowering everyone
to build reliable and efficient software.
GET STARTED
Version 1.97.1
Why Rust?
Performance
Rust is blazingly fast and memory-efficient: with no runtime or garbage collector, it can power performance-critical services, run on embedded devices, and easily integrate with other languages.
Reliability
Rust's rich type system and ownership model guarantee memory-safety and thread-safety — enabling you to eliminate many classes of bugs at compile-time.
Productivity
Rust has great documentation, a friendly compiler with useful error messages, and top-notch tooling — an integrated package manager and build tool, smart multi-editor support with auto-completion and type inspections, an auto-formatter, and more.
Build it in Rust
In 2018, the Rust community decided to improve the programming experience for a few distinct domains (see the 2018 roadmap). For these, you can find many high-quality crates and some awesome guides on how to get started.

Rust

- This talk focus on Rust language **reliability**, not Rust ecosystem:
  - `cargo` which is a great package manager (like `uv`)
  - `rustfmt` rust default formatter, looks good (for me)
  - `clippy` low false-positive linter (like `ruff`)
- Rust tries to solve problems *systematically*
- At a cost of higher upfront learning cost
- E.g. beginners usually spend first 2 weeks fighting the borrow checker

Hello world

```rust
fn main() {
    println!("Hello world!");
}
```

Content given in the order of

problems and *solutions*

(concepts in rust)

Billion dollar mistake

NULL

```python
...
x = 1
x += 1
print(x)  # 2
```

NULL

Python

```
...
x = None
...
if False:
    x = 1
x += 1  # or NameError: name 'x' is not defined
```

How?

Mostly afterthought for interpreted languages.

- Python - use `typings` (need to run separate mypy/ty check)
- Java - `@NotNull` annotation (only for docs)
- Go - zero value (may come as surprise) / `nil`

How?

Mostly afterthought for interpreted languages.

- Python - use `typings` (need to run separate mypy/ty check)
- Java - `@NotNull` annotation (only for docs)
- Go - zero value (may come as surprise) / `nil`
- Rust - sum types

Rust

Sum types.

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

Rust

Sum types.

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

```rust
let x = Some(1);
let x = None;
```

Rust

```rust
let x = Some(1);
println!("{x}");
```

Rust

```
   Compiling playground v0.0.1 (/playground)
error[E0277]: `Option<_>` doesn't implement `std::fmt::Display`
 --> src/main.rs:3:15
  |
3 |     println!("{x}");
  |              ^^^ `Option<_>` cannot be formatted with the default formatter
  |
  = help: the trait `std::fmt::Display` is not implemented for `Option<_>`
  = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead

For more information about this error, try `rustc --explain E0277`.
error: could not compile `playground` (bin "playground") due to 1 previous error
```

Rust

```rust
let x = Some(1);
println!("{x:?}");  // Debug print, outputs: Some(1)
```

Rust

```rust
let x = Some(1);
x += 1;
```

```
   Compiling playground v0.0.1 (/playground)
error[E0368]: binary assignment operation `+=` cannot be applied to type `Option<{integer}>`
   --> src/main.rs:3:5
    |
  3 |     x += 1;
    |     -^^^^^
    |     |
    |     cannot use `+=` on type `Option<{integer}>`
    |
note: `Option<{integer}>` does not implement `AddAssign<{integer}>`
   --> /playground/.rustup/toolchains/stable-x86_64-unknown-linux-gnu/lib/rustlib/src/rust/library/core/src/option.rs:597:1
    |
597 | pub enum Option<T> {
    | ^^^^^^^^^^^^^^^^^^ `Option<{integer}>` is defined in another crate

For more information about this error, try `rustc --explain E0368`.
error: could not compile `playground` (bin "playground") due to 1 previous error
```

Rust

```rust
let x = Some(1);
if let Some(n) = x {
    n += 1;
}
```

Rust

```
   Compiling playground v0.0.1 (/playground)
error[E0384]: cannot assign twice to immutable variable `n`
 --> src/main.rs:4:9
  |
3 |     if let Some(n) = x {
  |                 - first assignment to `n`
4 |         n += 1;
  |         ^^^^^^ cannot assign twice to immutable variable
  |
help: consider making this binding mutable
  |
3 |     if let Some(mut n) = x {
  |                 +++
help: to modify the original value, take a borrow instead
  |
3 |     if let Some(ref mut n) = x {
  |                 +++++++
```

Rust

```rust
let x = Some(1);
if let Some(mut n) = x {
    *n += 1;
}
```

Rust

```rust
let x = Some(1);
if let Some(mut n) = x {
    *n += 1;
}
println!("{x:?}");  // Outputs: Some(1)    ???
```

Rust

```rust
let x = Some(1);
if let Some(mut n) = x {
    *n += 1;
}
println!("{x:?}");  // Outputs: Some(1)    ???
```

pass-by-value vs pass-by-reference, due to `Copy` trait.

Rust

```Rust
let mut x = Some(1);          // mut
if let Some(n) = &mut x {     // &mut
    *n += 1;                  // *
}
println!("{x:?}");  // Outputs: Some(2)
```

Rust

or use functional style:

```rust
let mut x = Some(1);
x = x.map(|n| n + 1);
println!("{x:?}");  // Outputs: Some(2)
```

Dealing with enum

Enum in Python

```python
import enum

class Payment(enum.Enum):
    CASH = 1
    CARD = 2

match payment:
    case Payment.CASH: ...
    case Payment.CARD: ...
```

Enum in Python

Adding new variant

```python
class Payment(enum.Enum):
    CASH = 1
    CARD = 2
    BANK_TRANSFER = 3  # this
```

Enum in Python

```python
match payment:
    case Payment.CASH: ...
    case Payment.CARD: ...
    # easily forgotten
```

Enum in Python

Or different states

```python
last_four = 7371  # only for Payment.CARD
match payment:
    case Payment.CASH: ...
    case Payment.CARD:
        if last_four == 7371: ...
        ...
```

Enum in Python

Usual way to fix it in python

```python
class Payment:
    category: Payment
    amount: int         # only for cash
    last_four: int      # only for card
    # these extra variants states are easily missed too
```

How?

- Python - be careful, remember
- Go - be careful, remember
- Java - switch is exhaustive, sealed class for custom values

How?

- Python - be careful, remember
- Go - be careful, remember
- Java - switch is exhaustive, sealed class for custom values
- Rust - enum + exhaustive match

Rust

Rust

```rust
enum Payment {
    Cash(u64),
    Card { last_four: u16 },
}
let payment = Payment::Cash(10);
match payment {
    Payment::Cash(amount) => todo!(),
    Payment::Card { last_four } => todo!(),
}
```

Rust

```rust
enum Payment {
    Cash(u64),
    Card { last_four: u16 },
    BankTransfer(String), // add this
}
```

Rust

```
   Compiling playground v0.0.1 (/playground)
error[E0004]: non-exhaustive patterns: `Payment::BankTransfer(_)` not covered
  --> src/main.rs:9:11
   |
 9 |     match payment {
   |           ^^^^^^^ pattern `Payment::BankTransfer(_)` not covered
   |
note: `Payment` defined here
  --> src/main.rs:1:6
   |
 1 | enum Payment {
   |      ^^^^^^^
...
 4 |     BankTransfer(String),
   |     ------------ not covered
   = note: the matched value is of type `Payment`
help: ensure that all possible cases are being handled by adding a match arm with a wildcard pattern or an explicit pattern as
shown
   |
11 ~         Payment::Card { last_four } => todo!(),
12 ~         Payment::BankTransfer(_) => todo!(),
   |
```

Rust

```rust
match payment {
    Payment::Cash(amount) => todo!(),
    Payment::Card { last_four } => todo!(),
    Payment::BankTransfer(reference) => todo!(),
    // ^ required to compile
}
```

Error handling

Invisible errors hidden in upstream source

```python
def read_config():
    with open("config.json") as f:
        return json.load(f)

def start_app():
    config = read_config()
    start_server(config)
```

Invisible errors hidden in upstream source

So what are the errors? No idea.

One cannot tell from the type/signature whether read_config() can:

- raise `FileNotFoundError`
- raise `JSONDecodeError`
- raise `PermissionError`
- raise something else

Pokemon gotta catch ‘em all!

Python

```python
def start_app():
    try:
        config = read_config()
        start_server(config)
    except BaseException:
        # often result in defensive code
```

Pokemon gotta catch ‘em all!

```
start_app()
    ↓
load_config()
    ↓
read_file()
    ↓
open()
    💥 FileNotFoundError
    ↓
    ↓ propagates automatically
    ↓
    ↓
somewhere much higher
```

How?

- Python
  - dig into implementation, documentation or call stack
  - defensive coding, pokemon gotta catch ‘em all
  - let it crash in prod
- Java: same as python, there are checked exceptions but most are unchecked
- Go: error is visible by convention (`if err != nil`), but not enforced

How?

- Python
  - dig into implementation, documentation or call stack
  - defensive coding, pokemon gotta catch ‘em all
  - let it crash in prod
- Java: same as python, there are checked exceptions but most are unchecked
- Go: error is visible by convention (`if err != nil`), but not enforced
- Rust: enum + explicit error handling

Rust

Gist of rust type to handle errors:

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

Rust

Not handling error (explicitly panic on errors - unwrap)

Rust

```rust
fn read_config() -> String {
    std::fs::read_to_string("config.json").unwrap()
}
fn start_app() {
    let config = read_config();
    start_server(config);
}
```

Rust

Rust

```rust
fn read_config() -> Result<String, std::io::Error> {
    std::fs::read_to_string("config.json")?
}
fn start_app() -> Result<(), io::Error> {
    let config = read_config()?;  // propagate
    start_server(config);
    Ok(())
}
```

Rust

? is a syntactic sugar

```Rust
fn read_config() -> Result<String, std::io::Error> {
    match std::fs::read_to_string("config.json") {
        Ok(config) => Ok(config),
        Err(error) => Err(error.into()),
    }
}
```

Rust

If you want to handle it

```rust
fn start_app() {
    match read_config() {
        Ok(config) => start_server(config),
        Err(error) => eprintln!("Failed to read config: {error}"),
    }
}
```

Common misconception

Rust does not force you to “handle” a `Result`
but forces you to *explicitly acknowledge* one

Error handling

Extra notes on 2 categories of errors.

- panic
  - non-recoverable error or bugs
- `Result`
  - expected failures

Resource cleanups

`defer` in Go

```go
func getUser() ([]byte, error) {
    resp, err := http.Get("https://example.com/user")
    if err != nil {
        return nil, err
    }

    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return nil, err
    }

    return body, nil
}
```

`defer` in Go

```go
func getUser() ([]byte, error) {
    resp, err := http.Get("https://example.com/user")
    if err != nil {
        return nil, err
    }

    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return nil, err
    }

    return body, nil
}
```

Looks perfectly reasonable.

`defer` in Go

```go
func getUser() ([]byte, error) {
    resp, err := http.Get("https://example.com/user")
    if err != nil {
        return nil, err
    }

    body, err := io.ReadAll(resp.Body)
    if err != nil {
        return nil, err
    }

    return body, nil
}
```

Looks perfectly reasonable. Spot the bug?

`defer` in Go

Resource leak due to missing `defer resp.Body.Close()`

```go
func getUser() ([]byte, error) {
    resp, err := http.Get("https://example.com/user")
    if err != nil {
        return nil, err
    }
    defer resp.Body.Close()  // this

    body, err := io.ReadAll(resp.Body)
    ...
}
```

How?

- Python - have context manager, but can use without
- Java - try-with-resources is similar to Python
- Go - cleanup is a convention, read docs and remember to cleanup

How?

- Python - have context manager, but can use without
- Java - try-with-resources is similar to Python
- Go - cleanup is a convention, read docs and remember to cleanup
- Rust - cleanup is tied to scope, lifetime + Drop
  - note, async rust does not have Drop yet

Rust

```rust
fn get_user() -> Result<String, reqwest::Error> {
    let response = reqwest::blocking::get("https://example.com/user")?;
    let body = response.text()?;

    Ok(body)
} // response is dropped here
```

RAII

Resource Acquisition Is Initialization, simple example

Rust

```rust
{
    let file = std::fs::File::open("data.txt")?;

    // use file...

} // ← file is dropped here → file is closed
```

RAII

Even when you return early

Rust

```rust
fn process() -> Result<(), Error> {
    let file = File::open("data.txt")?;
    if something_wrong() {
        return Err(Error::Failed);
    }
    Ok(())
} // file is still dropped
```

RAII

Multiple resources

Rust

```
fn process() -> Result<(), Error> {
    let file = File::open("data.txt")?;
    let connection = TcpStream::connect(addr)?;
    // ...
    Ok(())
} // drop connection then file, if connect failed file still dropped
```

RAII

DIY

```rust
struct Resource;
impl Drop for Resource {
    fn drop(&mut self) { println!("Cleaning up!"); }
}
fn main() {
    let resource = Resource;
    println!("Using resource...");
} // "Cleaning up!"
```

Fearless concurrency

Concurrency is hard because of one simple problem:

multiple things can access the same mutable state

Mutability

JavaScript `const` is misleading

```javascript
const user = { name: "Ivan" };
user = {};          // this won't work as expected
user.name = "Bob"; // nani?
```

Mutability

JavaScript `const` is misleading

```JavaScript
const user = { name: "Ivan" };
user = {};          // this won't work as expected
user.name = "Bob"; // nani?
```

`const` prevents reassignment, not mutation.

Mutability

```rust
let x = String::from("hello");
x.push('!');
```

Mutability

```rust
let x = String::from("hello");
x.push('!');
```

```
   Compiling playground v0.0.1 (/playground)
error[E0596]: cannot borrow `x` as mutable, as it is not declared as mutable
 --> src/main.rs:9:1
  |
9 | x.push('!');       // does not work
  | ^ cannot borrow as mutable
  |
help: consider changing this to be mutable
  |
8 |     let mut x = String::from("hello");
  |         +++

For more information about this error, try `rustc --explain E0596`.
error: could not compile `playground` (bin "playground") due to 1 previous error
```

Mutability

```rust
let mut x = String::from("hello");
x.push('!');        // works
```

Mutability

```rust
let mut x = String::from("hello");
x.push('!');         // works
```

`mut` makes mutation explicit. Immutable by default.

Ownership

Python

```python
def add(items):
    items.append(3)

items = [1, 2]
add(items)
# items is now [1, 2, 3]
```

Ownership

```python
def add(items):
    items.append(3)

items = [1, 2]
add(items)
# items is now [1, 2, 3]
```

The function signature doesn't say that `items` will be changed.

Ownership

```python
def add(items):
    items.append(3)

items = [1, 2]
add(items)
# items is now [1, 2, 3]
```

The function signature doesn't say that `items` will be changed.

Side effects like this might be a surprise at times.

Ownership

```python
def process(data):
    transform(data)
    validate(data)
```

Python

```java
void process(List<Data> data) {
    transform(data);
    validate(data);
}
```

Java

```go
func process(data []Data) {
    transform(data)
    validate(data)
}
```

Go

Can `process` change my data?

Ownership

```python
def process(data):
    transform(data)
    validate(data)
```

Python

```java
void process(List<Data> data) {
    transform(data);
    validate(data);
}
```

Java

```go
func process(data []Data) {
    transform(data)
    validate(data)
}
```

Go

Can `process` change my data?

You have to inspect the implementation.

Ownership

In rust

```rust
// more idiomatic read-only would be &[Data]
fn process(data: Vec<Data>) { ... }        // owns it
fn process(data: &Vec<Data>) { ... }       // borrows it
fn process(data: &mut Vec<Data>) { ... }   // may mutate it
```

Ownership

Rust

```rust
fn add(items: Vec<i32>) {
    // owns items
}

let items = vec![1, 2];
add(items);             // items moved here
println!("{items:?}"); // fail to compile, items were moved
```

Ownership

```rust
fn add(items: Vec<i32>) {
    // owns items
}

let items = vec![1, 2];
add(items);             // items moved here
println!("{items:?}"); // fail to compile, items were moved
```

Ownership is explicit in the type/signature.

Ownership

Don't want to give ownership away?

```rust
fn add(items: &mut Vec<i32>) {
    items.push(3);
}

let mut items = vec![1, 2];
add(&mut items);
```

Here comes borrowing.

Borrowing

Python

```python
items = [1, 2]  # python aliases freely
a = items
b = items

a.append(3)
print(b)  # [1, 2, 3]
```

Borrowing

Rust makes the access explicit:

```rust
let mut items = vec![1, 2];

let a = &items;       // read
let b = &items;       // read
```

Borrowing

Rust makes the access explicit:

```rust
let mut items = vec![1, 2];

let a = &items;       // read
let b = &items;       // read
```

Borrowing does not give ownership.

Many readers

XOR

One writer

Rust borrowing rule

Many readers:

```Rust
let a = &items;
let b = &items;
```

One writer:

```Rust
let a = &mut items;
```

Rust borrowing rule

Two writers:

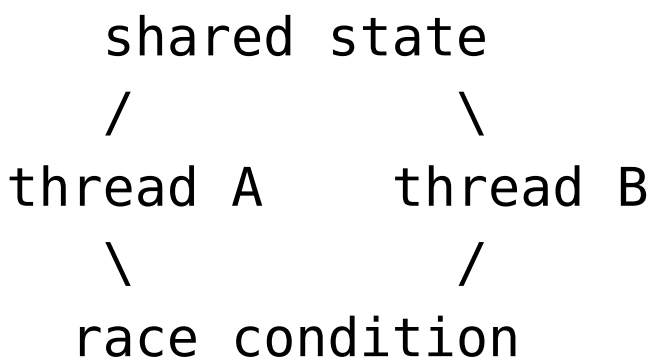
```Rust
let a = &mut items; // first mutable borrow
let b = &mut items; // second mutable borrow, fail to compile
```

Reader + writer:

```Rust
let a = &items;     // immutable borrow
let b = &mut items; // mutable borrow, fail to compile
```

Why?

Because shared mutable state is dangerous.



Why?

Without synchronization:

```python
counter = 0

# A                 # B
counter += 1        counter += 1
```

Possible result: 1 instead of 2.

Why?

To fix it

```python
with lock:
    counter += 1
```

How?

- Python - remember to use lock
- Java - remember to use lock
- Go - remember to use lock
- They work but the programmer must remember
  - compiles/runs fine without

How?

- Python - remember to use lock
- Java - remember to use lock
- Go - remember to use lock
- They work but the programmer must remember
  - compiles/runs fine without
- Rust compiler rejects it by default
  - use `Arc` + `Mutex`

Rust

The same borrowing rule applies to threads.

Rust

```rust
use std::thread;

let mut counter = 0;

thread::spawn(|| {
    counter += 1;
});
```

Rust

The compiler rejects it: the thread could outlive `counter`.

```
   Compiling playground v0.0.1 (/playground)
error[E0373]: closure may outlive the current function, but it borrows `counter`, which is owned by the current function
 --> src/main.rs:6:15
  |
6 | thread::spawn(|| {
  |               ^^ may outlive borrowed value `counter`
7 |     counter += 1;
  |     ------- `counter` is borrowed here
  |
note: function requires argument type to outlive `'static`
 --> src/main.rs:6:1
  |
6 | / thread::spawn(|| {
7 | |     counter += 1;
8 | | });
  | |__^
help: to force the closure to take ownership of `counter` (and any other referenced variables), use the `move` keyword
  |
6 | thread::spawn(move || {
  |               ++++
```

Rust

If we really want shared mutation:

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

let counter = Arc::new(Mutex::new(0));
let a = Arc::clone(&counter);
let b = Arc::clone(&counter);
thread::spawn(move || *a.lock().unwrap() += 1);
thread::spawn(move || *b.lock().unwrap() += 1);
```

Rust

```
Plain Text
1  Arc   → shared ownership
2  Mutex → one thread at a time
```

Rust

```
Arc   → shared ownership
Mutex → one thread at a time
```

Two traits beneath it:

- `Send` - safe to *move* to another thread
- `Sync` - safe to *share* between threads (via reference)

`Mutex` provides `Sync`, `Arc` provides `Send` if data is `Sync`.

Fearless concurrency

A combination of:

- mutability
- ownership
- borrowing
- aliasing rules (one writer xor many readers)
- `Send` + `Sync` trait bounds

Rust doesn’t prevent all concurrency bugs.

It prevents *data races* at compile time.

Why not AI?

Writing code with AI

- getting better but can still make the same mistakes
- can fix fast, even if something broke at prod
- limited to context window 200k or 1M
  - can this fit in all the dependencies source code?

How rust come in

Rust prevents these issues systematically, at a cost

- longer compile time
  - can be improved with tools like subsecond
- higher learning cost
  - but you don't have to teach AI how to write Rust

Why learn Rust?

Why learn a hard language when AI writes code?

Why learn Rust?

Why learn a hard language when AI writes code?

Make life harder. More challenging, more fun!

Why learn Rust?

Why learn a hard language when AI writes code?

~~Make life harder. More challenging, more fun!~~

Because Rust prevents bugs AI can’t see in context windows.

What's next?

- The Rust Book
- Rust by Example
- rustlings
- Drop by Rust Malaysia telegram group
  - official channel https://t.me/rustlangmalaysia
  - lang chat group https://t.me/golangmalaysia
- or come join next workshop in October