Rust Trainings All in One

- High-level intro about Rust
- Ownership, borrow check, and lifetime
- Typesystem and data structures
- Concurrency - primitives
- Concurrency - async/await
- Networking and security
- FFI with C/Elixir/Swift/Java
- WASM/WASI
- Rust for real-world problems

High-level Intro About Rust

Why Rust?

Let's talk about values and tradeoffs first

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- Performance
- Portability
- Productivity
- Resilliency
- Rigor
- Safety
- Security
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

C

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- **Performance**
- Portability
- Productivity
- Resilliency
- Rigor
- Safety
- Security
- **Simplicity**
- Stability
- Thoroughness
- **Transparent**
- Velocity

Erlang/Elixir

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- Performance
- Portability
- **Productivity**
- **Resilliency**
- Rigor
- **Safety**
- Security
- **Simplicity**
- Stability
- Thoroughness
- Transparent
- Velocity

Python

- **Approachability**
- Availability
- Compatibility
- Composability
- Debuggability
- **Expressiveness**
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- Performance
- Portability
- **Productivity**
- Resilliency
- Rigor
- Safety
- Security
- **Simplicity**
- Stability
- Thoroughness
- Transparent
- Velocity

Java (in early days)

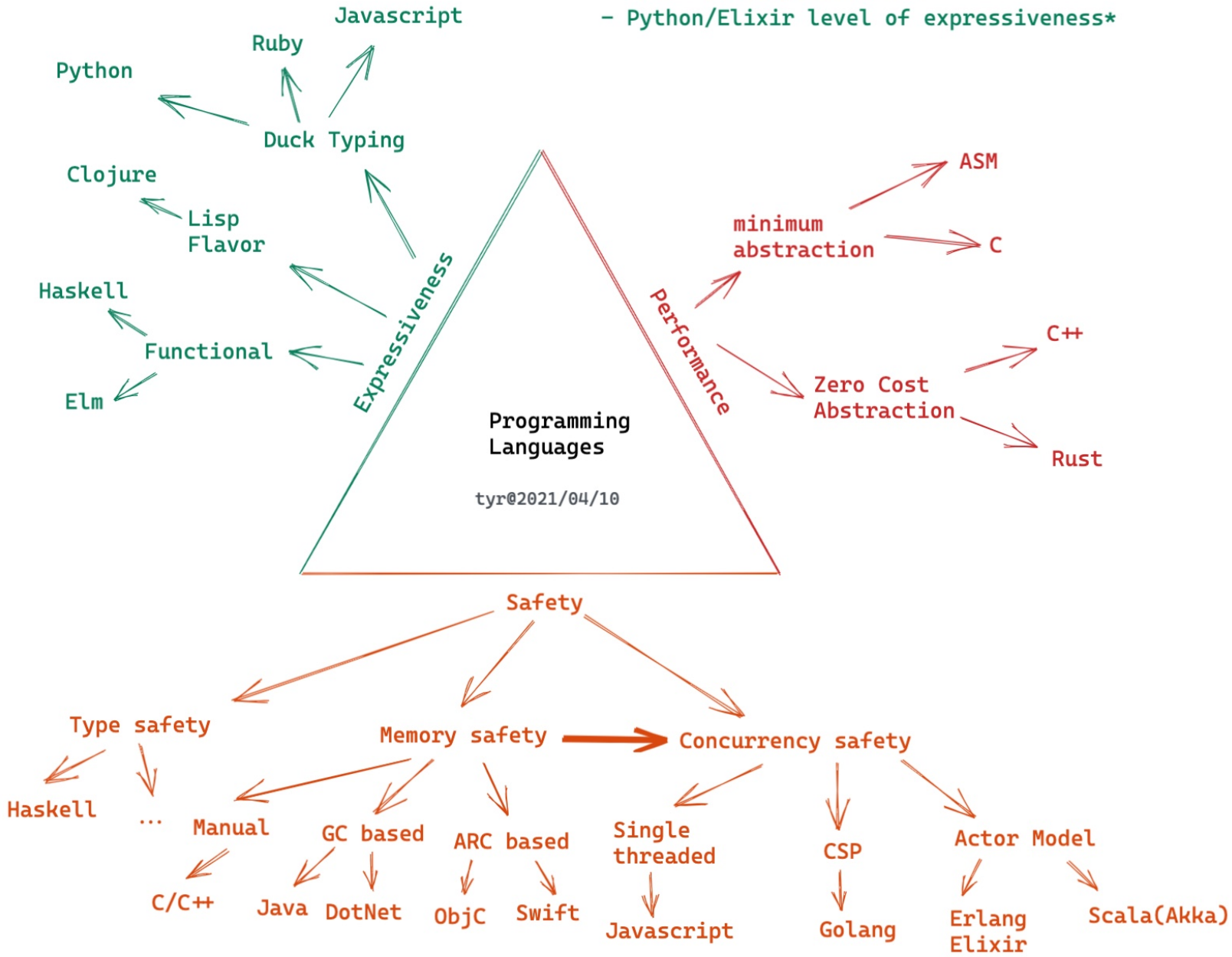
- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- **Performance**
- **Portability**
- Productivity
- Resilliency
- Rigor
- **Safety (memory)**
- **Security**
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

Rust

- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- **Expressiveness**
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- **Performance**
- Portability
- **Productivity**
- Resilliency
- Rigor
- **Safety!!!**
- Security
- Simplicity
- Stability
- Thoroughness
- Transparent
- Velocity

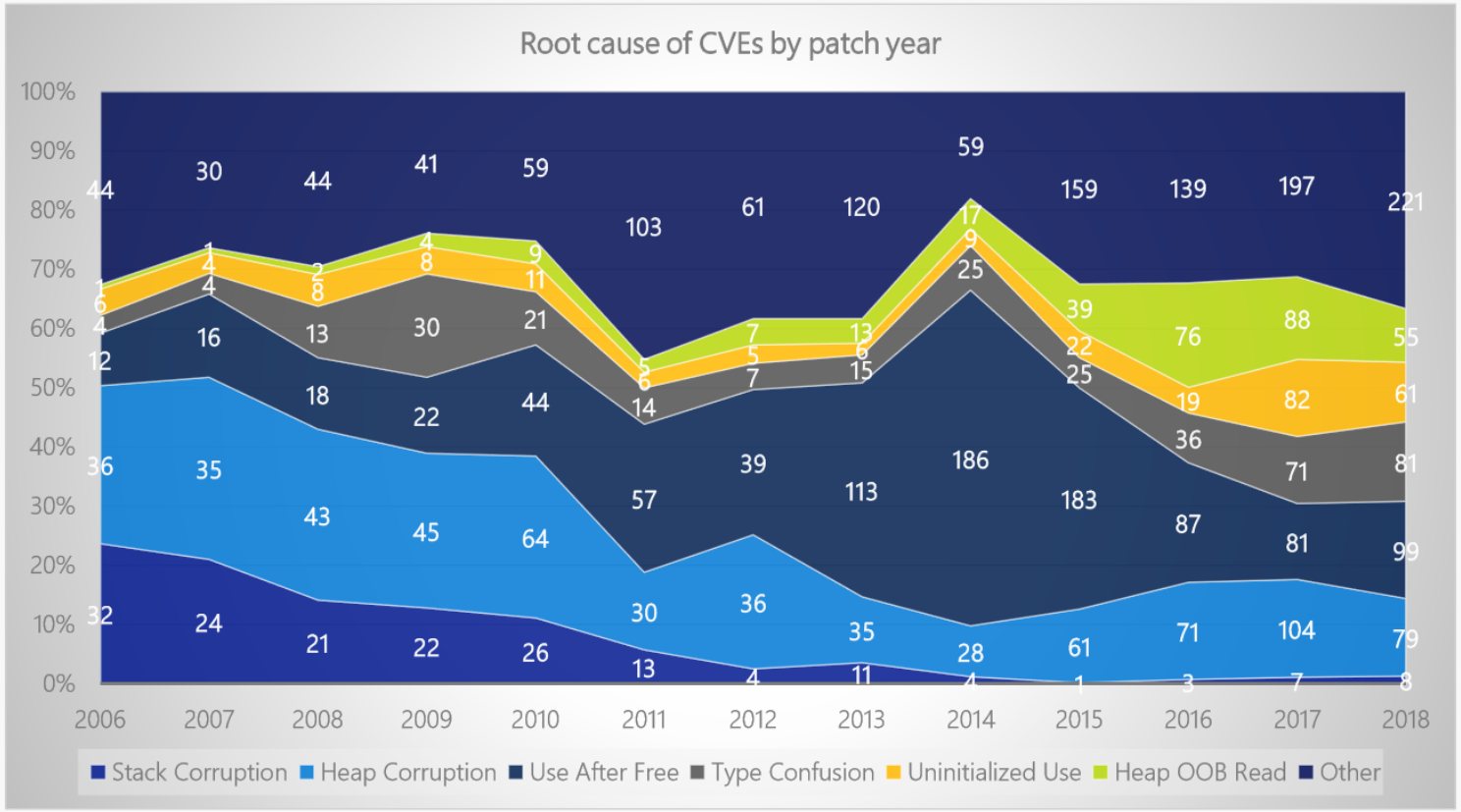
Rust:

- Type, Memory and Concurrency safety
- C/C++ level performance
- Python/Elixir level of expressiveness*



Why safety is important?

Drilling down into root causes



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

- #1: heap out-of-bounds
- #2: use after free
- #3: type confusion
- #4: uninitialized use

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

(**Source:** **Memory Safety Issues Are Still the Leading Source of Security Vulnerabilities**)

Safety is hard!

- memory safety is not easy (you need to understand the corner cases)
- conccurency safety is really hard (without certain tradeoffs)
- Often you have to bear the extra layer of abstractions
  - normally it means performance hit

Memory safety

- Manually - C/C++: painful and error-prone
- Smart Pointers - C++/ObjC/Swift: be aware of cyclical references
- GC - Java/DotNet/Erlang: mubch bigger memory consumption, and STW
- Ownership - Rust: learning curve

Concurrency safety

- single-threaded - Javascript: cannot leverage multicore
- GIL - Python/Ruby: multithreading is notorious inefficient
- Actor model - Erlang/Akka: at the cost of memory copy and heap allocation
- CSP - Golang: at the cost of memory copy and heap allocation
- Ownership + Type System - Rust: super **elegant** and **no extra cost**!

How Rust achieves

memory and conccurency safety

without extra cost?

tyr@04/10/2021

value in a scope

⑤

i32, usize, [T;4]*
copy on move

Box, Arc, MutexGuard, RwLockGuard

Allow a type to do dereference or mutable deferences

Captured value is passed by reference unless explicitly `move`

Deref/DerefMut

Closure (Fn/FnMut)

Copy Trait

Exception: Rc

Type could be moved between threads safely

Send Trait

Type Safety

Sync Trait

Exception: Rc, Cell, RefCell

Type could be shared between threads safely

①

When passing a value to other scope, move occurs.

Can have only one owner in a scope. When owner left scope, value dropped.

```
{
    let arr = vec![1];
    let arr1 = vec![2];
    sum(arr);
    println!("arr: {:?}", arr);
}
```

I'm the owner

new owner: sum

arr1 dropped

②

can have multiple immutable references

```
{
  let arr = vec![1];
  let total = sum(&arr);
  let v = max(&arr);
}
```

I'm the owner

1st borrower: sum

2nd borrower: max

③

can have and only have one mutable reference exclusively

```
{
  let arr = vec![1];
  let ir = &arr;
  push(&mut arr);
  println!("ir: {:?}", ir);
}
```

immutable ref

I'm the owner

mutable ref

④

reference cannot outlive owner

```
{
  let arr = vec![1];

  thread::spawn(|| {
    println!("{:?}", arr);
  });
}
```

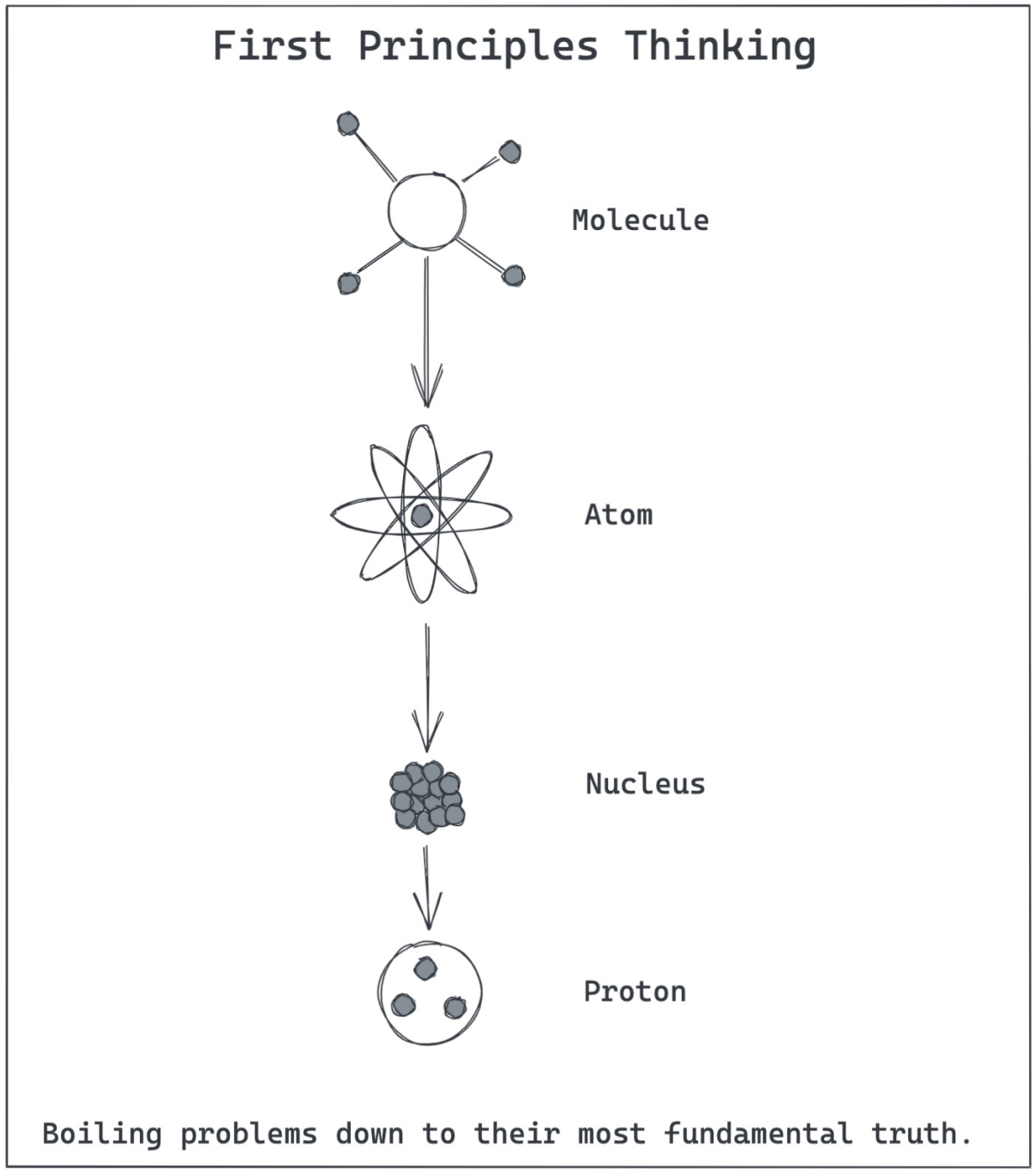
closure may outlive

reference here

Show me the code!

```rust
fn main() {
    let mut arr: Vec<i32> = vec![1, 2, 3];    move occurs because `arr` has type `Vec<i32>`, which does not implement the `Copy` trait
    arr.push(4);

    let _result: Result<(), Error> = process(arr);    value moved here
    let _v: Option<i32> = arr.pop(); // failed since arr is moved    borrow of moved value: `arr`

    // you can have multiple immutable references
    let mut arr1: Vec<i32> = vec![1, 2, 3];
    let ir1: &Vec<i32> = &arr1;
    let ir2: &Vec<i32> = &arr1;    immutable borrow occurs here

    println!("ir1: {:?} ir2: {:?}", ir1, ir2);

    // but you can't have both mutable and immutable references
    let mr1: &mut Vec<i32> = &mut arr1;    cannot borrow `arr1` as mutable because it is also borrowed as immutable
    // let mr2 = &mut arr1;

    println!("mr1: {:?} mr2: {:?}", mr1, ir2);    immutable borrow later used here

    // by default, closure borrows the data
    let mut arr2: Vec<i32> = vec![1, 2, 3];
    thread::spawn(|| {    closure may outlive the current function, but it borrows `arr2`, which is owned by the current function
        arr2.push(4);    `arr2` is borrowed here
    });

    // we shall move the data explicitly
    let mut arr3: Vec<i32> = vec![1, 2, 3];
    thread::spawn(move || arr3.push(4));
}

fn thread_safety() {
    // but certain types cannot be moved to other thread safely
    let mut rc1: Rc<Vec<i32>> = Rc::new(vec![1, 2, 3]);
    thread::spawn(move || {    `Rc<Vec<i32>>` cannot be sent between threads safely
        rc1.push(4);
    });
}
```

```rust
fn thread_safety_reasoning() {
    let mut map: HashMap<&str, &str> = HashMap::new();    move occurs because `map` has type `HashMap<&str, &str>`, which does not implem
    map.insert(k: "hello", v: "world");

    // Arc is an atomic reference counter which can be moved safely across threads
    let mut ir: Arc<HashMap<&str, &str>> = Arc::new(data: map);    variable does not need to be mutable
    map.insert(k: "hello1", v: "world1"); // you can't do this since map is moved    borrow of moved value: `map`
    let ir1: Arc<HashMap<&str, &str>> = ir.clone(); // this is cheap, just reference counter clone
    thread::spawn(move || assert_eq!(ir1.get("hello"), Some(&"world")));
    // but arc is immutable, so this would fail
    thread::spawn(move || ir.insert(k: "hello2", v: "world2"));    cannot borrow data in an `Arc` as mutable

    // the compiler guide you to use types that provides mutable reference for threads

    // use Mutex - you can't clone a Mutex, thus you can't make it availabe for multiple threads
    let mut map1: HashMap<&str, &str> = HashMap::new();
    map1.insert(k: "hello", v: "world");
    let mr: Mutex<HashMap<&str, &str>> = Mutex::new(map1);
    let mr1 = mr.clone();    no method named `clone` found for struct `Mutex<HashMap<&str, &str>>` in the current scope
    thread::spawn(move || mr.lock().unwrap().insert(k: "hello1", v: "world1"));
    mr1.lock().unwrap().insert("hello2", "world2");

    // use Mutex with Arc - now you have mutable access and multi-thread cloning
    let mut map2: HashMap<&str, &str> = HashMap::new();
    map2.insert(k: "hello", v: "world");
    let mr: Arc<Mutex<HashMap<&str, &str>>> = Arc::new(data: Mutex::new(map2));
    let mr1: Arc<Mutex<HashMap<&str, &str>>> = mr.clone();

    thread::spawn(move || mr.lock().unwrap().insert(k: "hello1", v: "world1"));
    thread::spawn(move || mr1.lock().unwrap().insert(k: "hello2", v: "world2"));

    // can I use Box (smart pointer for heap allocation)?
    let mut map1: HashMap<&str, &str> = HashMap::new();
    map1.insert(k: "hello", v: "world");
    let mr: Arc<Box<HashMap<&str, &str>>> = Arc::new(data: Box::new(map1));
    let mr1: Arc<Box<HashMap<&str, &str>>> = mr.clone();
    thread::spawn(move || (**mr).insert(k: "hello1", v: "world1"));    cannot borrow data in an `Arc` as mutable
    mr1.insert(k: "hello2", v: "world2");    cannot borrow data in an `Arc` as mutable
}
```

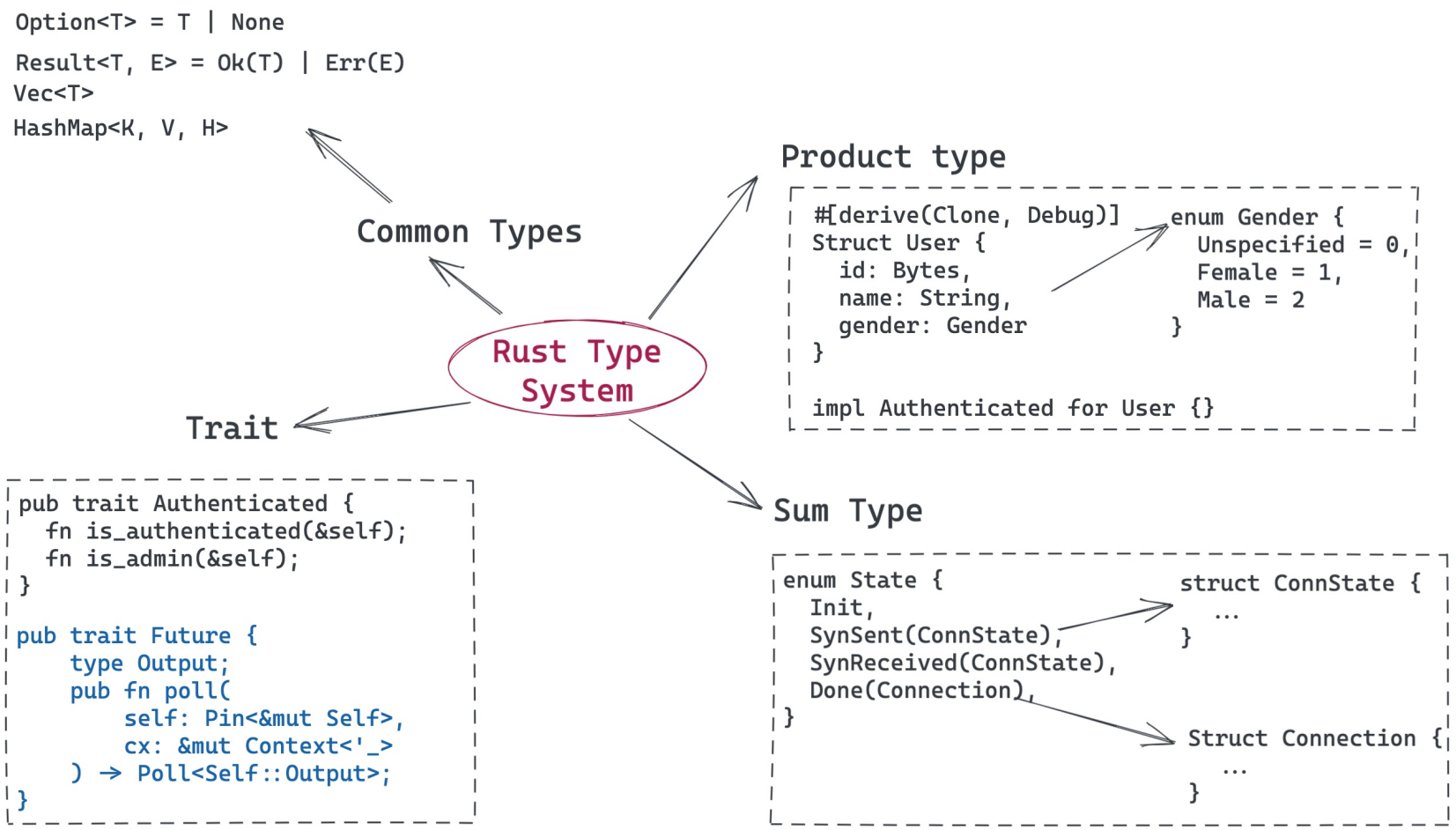


Recap

- One and only one owner
- Multiple immutable references
- mutable reference is mutual exclsive
- Reference cannot outlive owner
- **use type safety for thread safety**

With these simple rules, Rust achieved safety with zero cost abstraction

A glance at Rust Type System



How's Productivity of Rust?

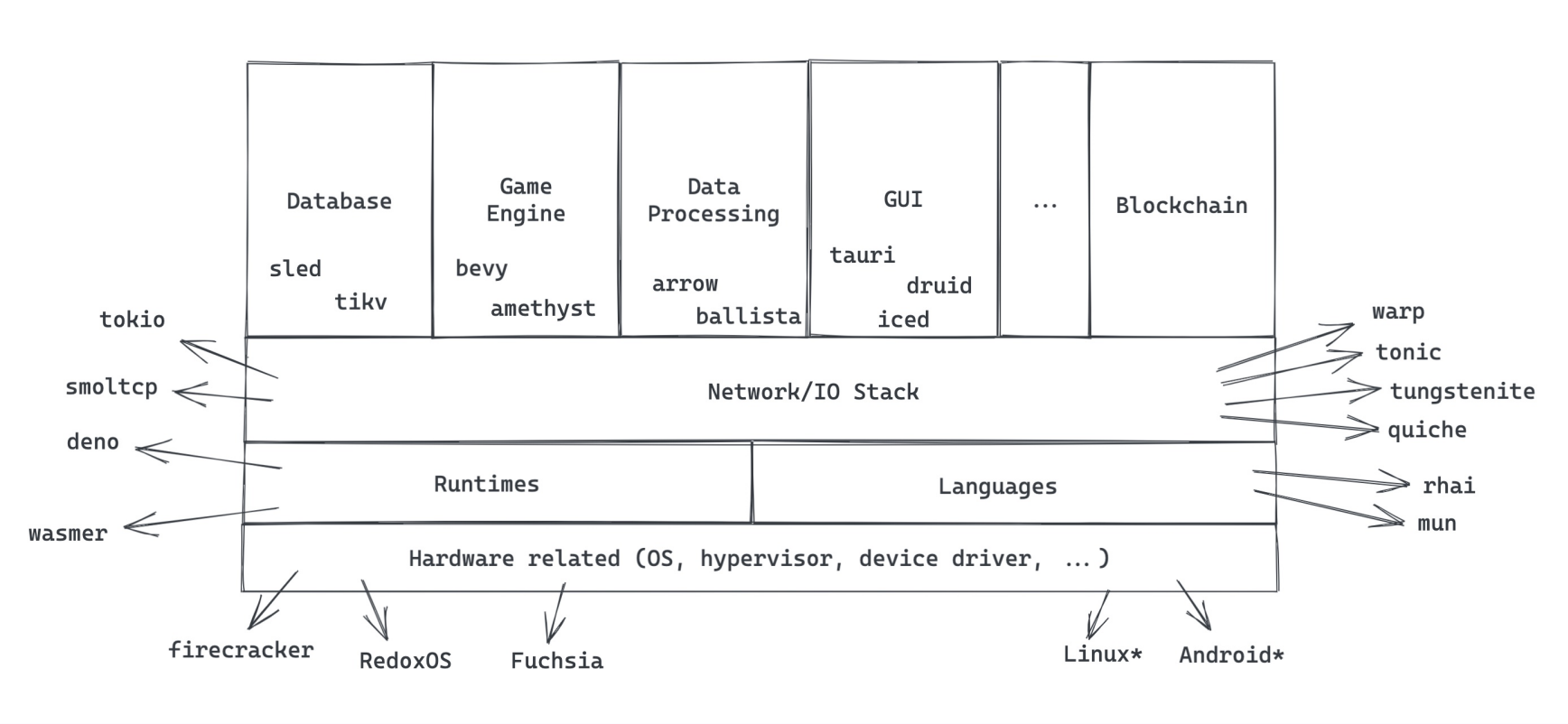
```
# client configuration

domain = "localhost"

[cert]
pem = """-----BEGIN CERTIFICATE-----
MIIBeTCCASugAwIBAgIBKjAFBgMrZXAwNzELMAkGA1UEBgwCVVMxFDASBgNVBAoM
C0RvbWFpbiBJbmMuMRIwEAYDVQQDDAlEb21haW4gQ0EwHhcNMjEwMzE0MTg0NTU2
WhcNMzEwMzEyMTg0NTU2WjA3MQswCQYDVQQGDAJVUzEUMBIGA1UECgwLRG9tYWlu
IEluYy4xEjAQBgNVBAMMCURvbWFpbiBDQTAqMAUGAytlcAMhAAzhorM9IPsXjBTx
ZxykGl5xZrsj3X2XqKjaAVutnf7po1wwWjAUBgNVHREEDTALgglsb2NhbGhvc3Qw
HQYDVR0OBBYEFD+NqChBZDOs5FMgefHJSIWiRTHXMBIGA1UdEwEB/wQIMAYBAf8C
ARAwDwYDVR0PAQH/BAUDAwcGADAFBgMrZXADQQA9sligQcYGaBqTxR1+JadSelMK
Wp35+yhVVuu4PTL18kWdU819w3cVlRe/GHt+jjlbk1i22TvfO5AaNmdxySkO
-----END CERTIFICATE-----"""

# server configuration

[identity]
key = """-----BEGIN PRIVATE KEY-----
MFMCAQEwBQYDK2VwBCIEII0kozd0PJsbNfNUS/oqI/Q/enDiLwmdw+JUnTLpR9xs
oSMDIQAtkhJiFdF9SYBIMcLikWPRIgca/Rz9ngIgd6HuG6HI3g==
-----END PRIVATE KEY-----"""

[identity.cert]
pem = """-----BEGIN CERTIFICATE-----
MIIBazCCAR2gAwIBAgIBKjAFBgMrZXAwNzELMAkGA1UEBgwCVVMxFDASBgNVBAoM
C0RvbWFpbiBJbmMuMRIwEAYDVQQDDAlEb21haW4gQ0EwHhcNMjEwMzE0MTg0NTU2
WhcNMjIwMzE0MTg0NTU2WjA5MQswCQYDVQQGDAJVUzEUMBIGA1UECgwLRG9tYWlu
IEluYy4xFDASBgNVBAMMC0dSUEMgU2VydmVyMCowBQYDK2VwAyEALZISYhXRfUmA
SDHC4pFj0SIHGv0c/Z4CIHeh7huhyN6jTDBKMBQGA1UdEQQNMAuCCWxvY2FsaG9z
dDATBgNVHSUEDDAKBggrBgEFBQcDATAMBgNVHRMEBTADAQEAMA8GA1UdDwEB/wQF
AwMH4AAwBQYDK2VwA0EAy7EOIZp73XtcqaSopqDGWU7Umi4DVvIgjmY6qbJZP0sj
ExGdaVq/7MOlZl1I+vY7G0NSZWZIUilX0CoOkrn0DA==
-----END CERTIFICATE-----"""
```

```rust
// you could also build your config with cert and identity separately. See tests.
let config: ServerTlsConfig = toml::from_str(config_file).unwrap();
let acceptor = config.tls_acceptor().unwrap();
let listener = TcpListener::bind(addr).await.unwrap();
tokio::spawn(async move {
    loop {
        let (stream, peer_addr) = listener.accept().await.unwrap();
        let stream = acceptor.accept(stream).await.unwrap();
        info!("server: Accepted client conn with TLS");

        let fut = async move {
            let (mut reader, mut writer) = split(stream);
            let n = copy(&mut reader, &mut writer).await?;
            writer.flush().await?;
            debug!("Echo: {} - {}", peer_addr, n);
        }

        tokio::spawn(async move {
            if let Err(err) = fut.await {
                error!("{:?}", err);
            }
        });
    }
});
```

Client: You, a month ago • init the project

```rust
let msg = b"Hello world\n";
let mut buf = [0; 12];

// you could also build your config with cert and identity separately. See tests.
let config: ClientTlsConfig = toml::from_str(config_file).unwrap();
let connector = config.tls_connector(Uri::from_static("localhost")).unwrap();

let stream = TcpStream::connect(addr).await.unwrap();
let mut stream = connector.connect(stream).await.unwrap();
info!("client: TLS conn established");

stream.write_all(msg).await.unwrap();

info!("client: send data");

let (mut reader, _writer) = split(stream);

reader.read_exact(buf).await.unwrap();

info!("client: read echoed data");
```

Things built with Rust



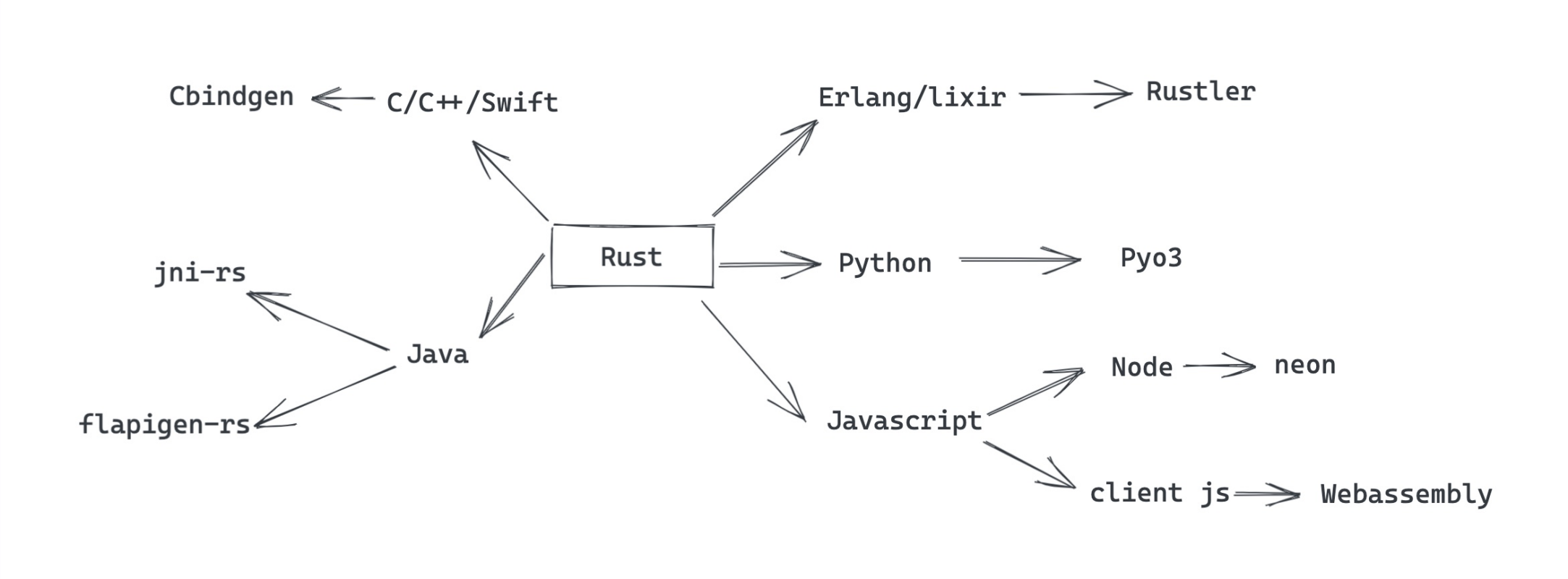
Should I use Rust?

- Rust is ideal when you need a system that reliable and performant
- Sometimes you don't, sometimes you do, sometimes you need that later
- it's all about tradeoffs

Rust for our use cases

- parts of the system that are bottlenecks
  - bottleneck on computation
  - bottleneck on memory consumption
  - bootleneck on I/O
- parser/decoder/encoder
- wants to leverage existingC/C++/Rust ecosystem (e.g. you need blake3 for hashing)

Rust FFI

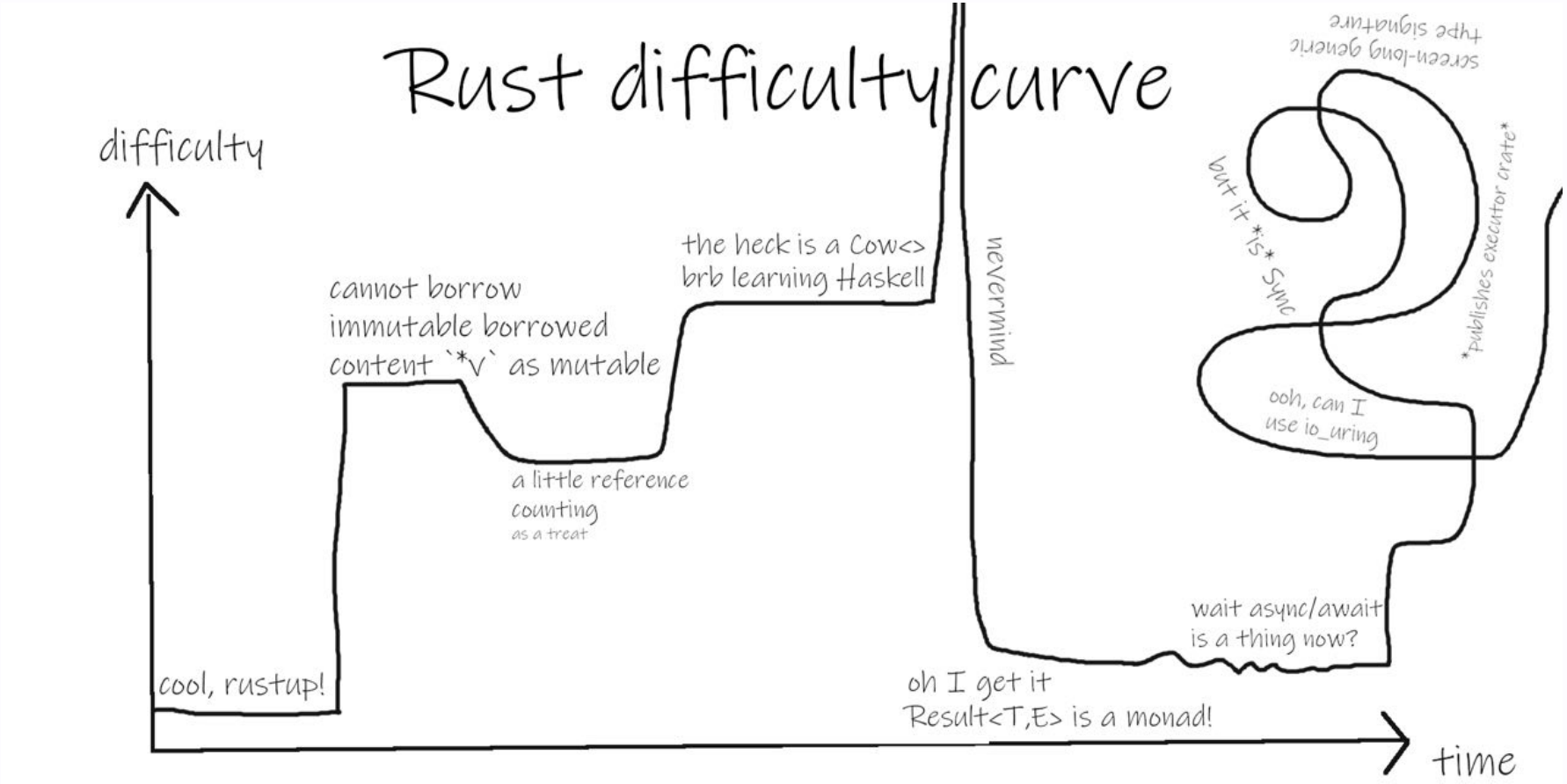

Cbindgen
C/C++/Swift
Erlang/lixir
Rustler
Rust
Python
Pyo3
jni-rs
Java
Node
neon
flapigen-rs
Javascript
client js
Webassembly

Learning rust as a(n)...

- Elixir eng: ownership model, type system, oh no mutation
- Scala eng: ownership model, oh no mutation
- Typescript eng: ownership model, multi-threaded programming
- Swift/Java eng: ownership model
- Python eng: ownership model, type system

The common misunderstandings

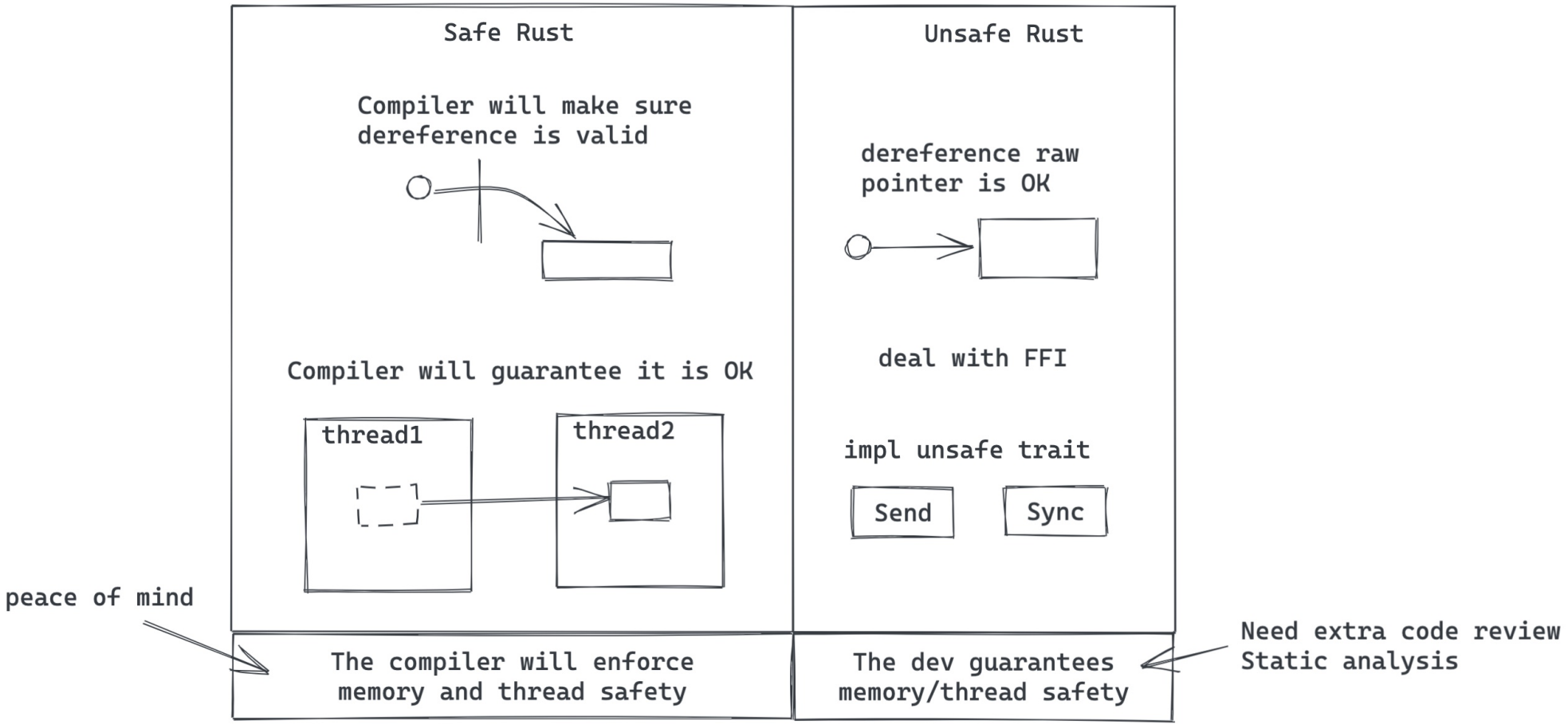
1. Rust is super hard to learn...



Rust is explicit

- Lots of knowledge about computer science is suddenly explicit to you
- If all your pain to learn a lang is 100%:
  - Rust:
    - Compiler help to reduce that to 90%
    - Then you suffer 70% the pains in first 3-6 months
    - Then the rest 20% in 3-5 years
  - Other:
    - You suffer 10-30% in first 3-6 months
    - Then 70%-90% in next 3-5 years

2. Unsafe Rust is evil...

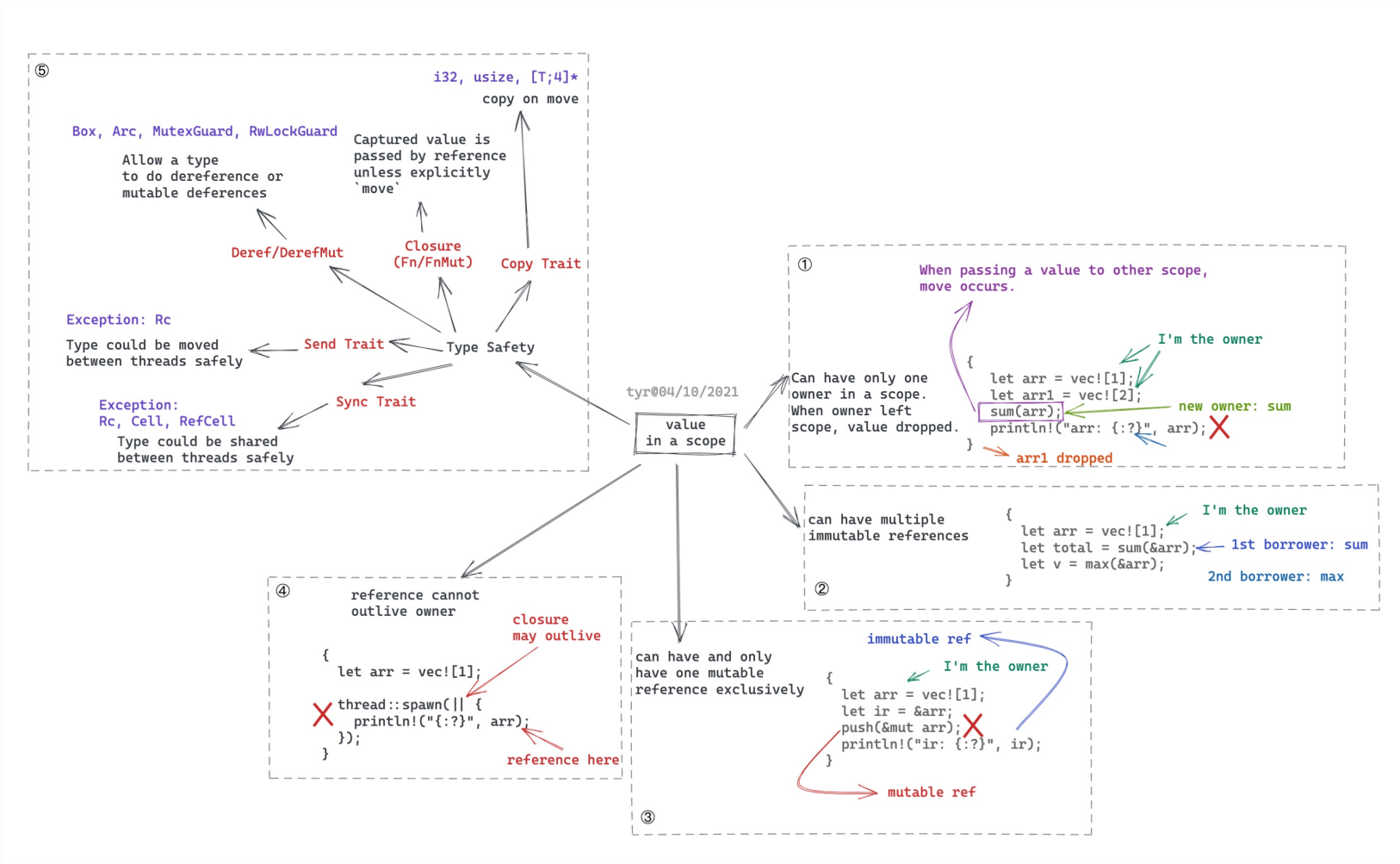


References

- The pain of real linear types in Rust
- Substructural type system
- Rust official book
- Rust official site
- Awesome Rust
- Are we web yet?
- Are we async yet?
- Are we gui yet?
- Are we learning yet?
- Are we game yet?
- Are we quantum yet?
- Are we IDE yet?
- Rust is for Professionals

Ownership, borrow check, and lifetime

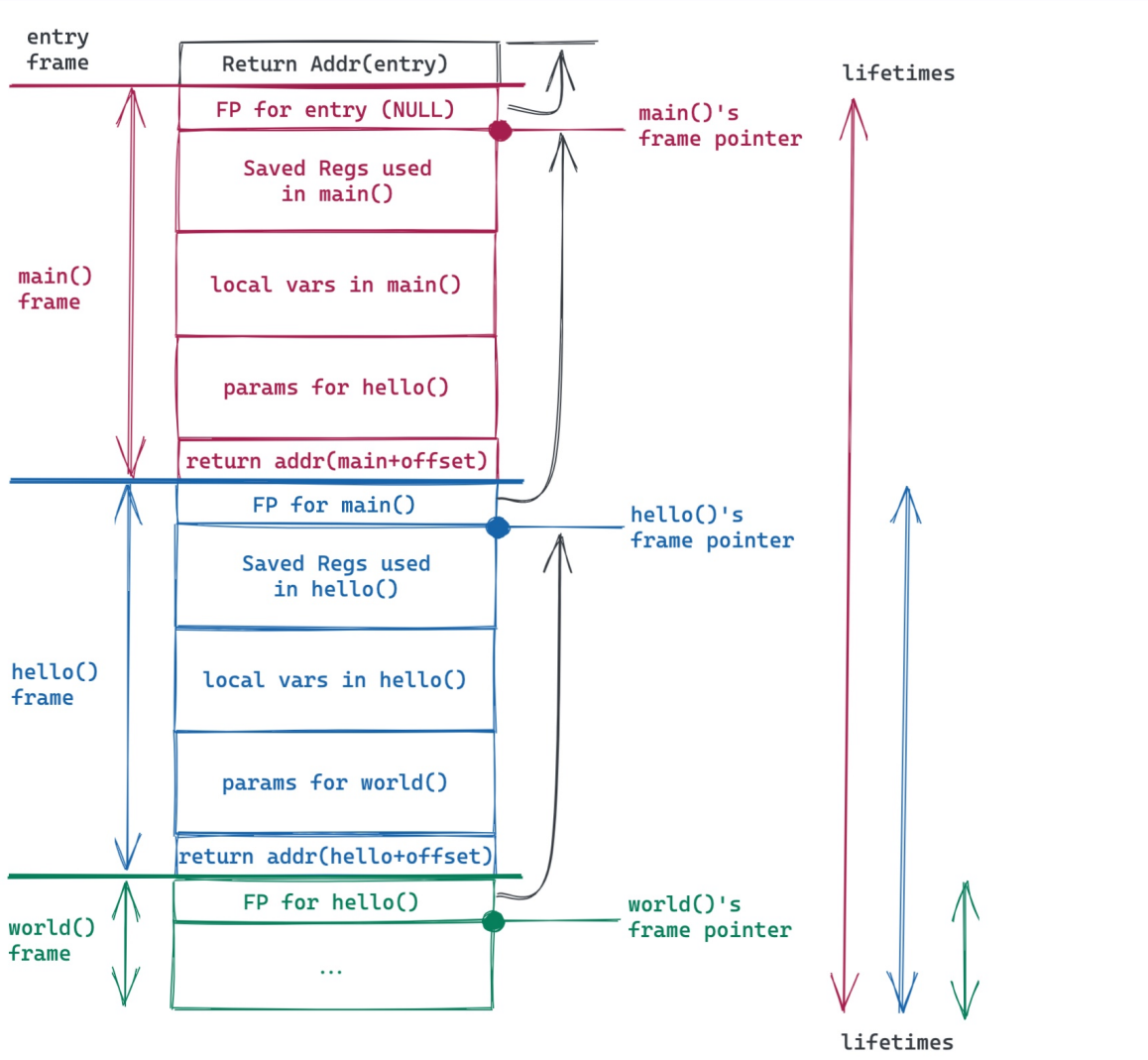
Ownership/Borrow Rules Review



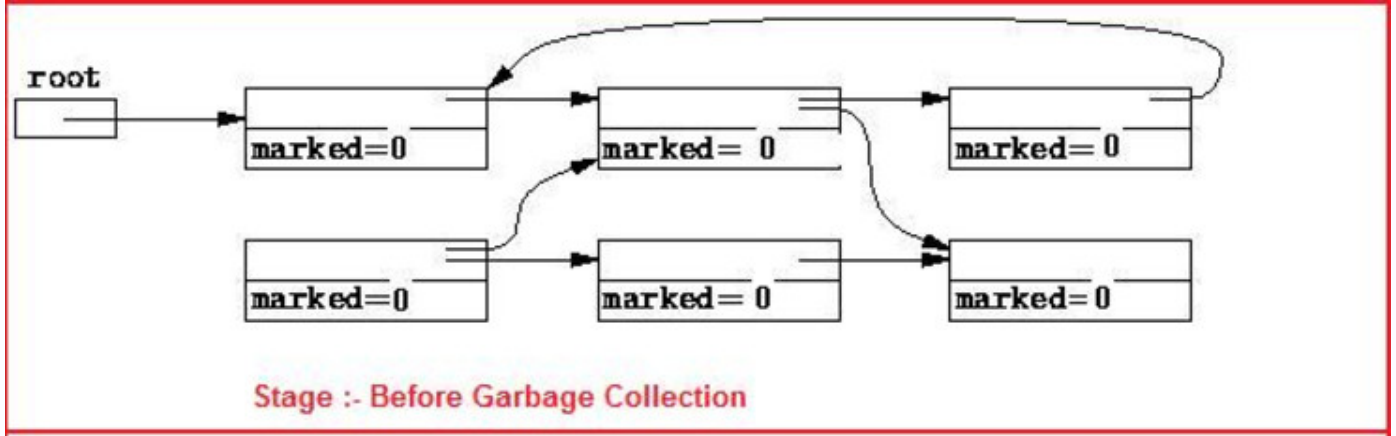
Lifetime, not a new idea

Lifetime: Stack memory

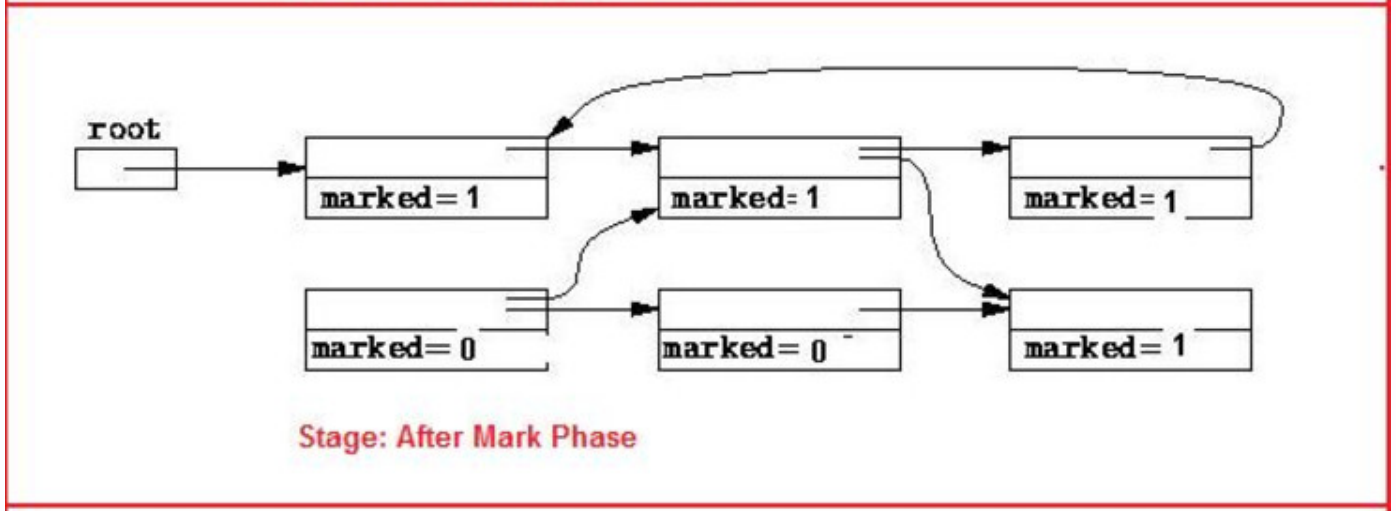
```c
#include <stdio.h>
static int VALUE = 42;

void world(char *st, int num) {
    printf("%s(%d)\n", st, num);
}


void hello(int num) {
    char *st = "hello world";
    int v = VALUE+num;
    world(st, v);
}

int main() {
    hello(2);
}
```



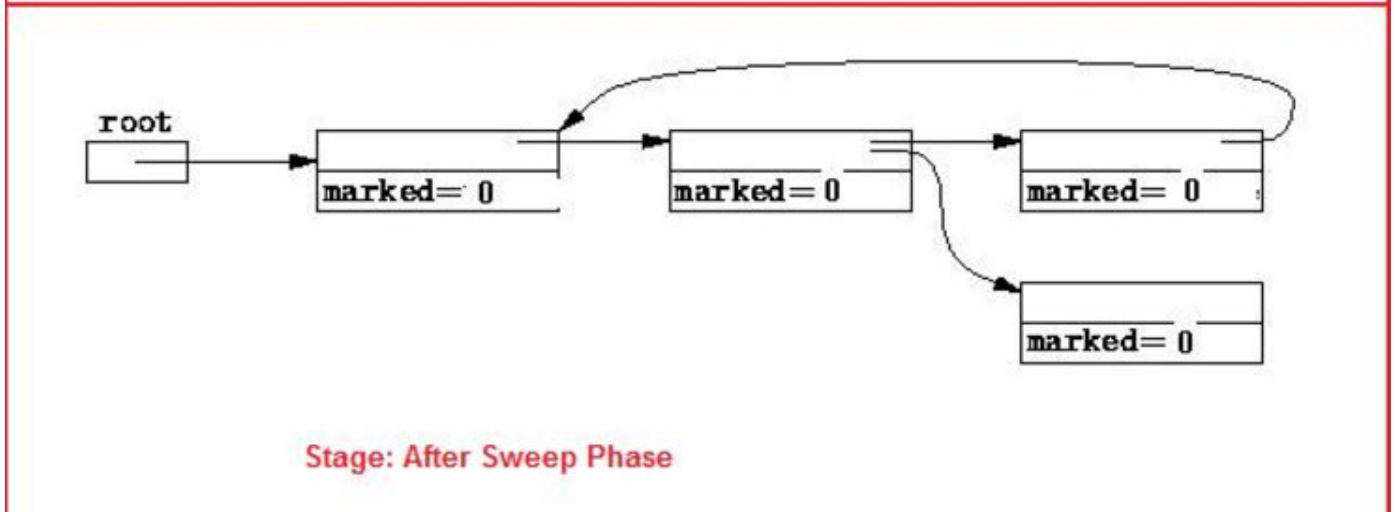
Lifetime: Heap memory (tracing GC)



Stage :- Before Garbage Collection

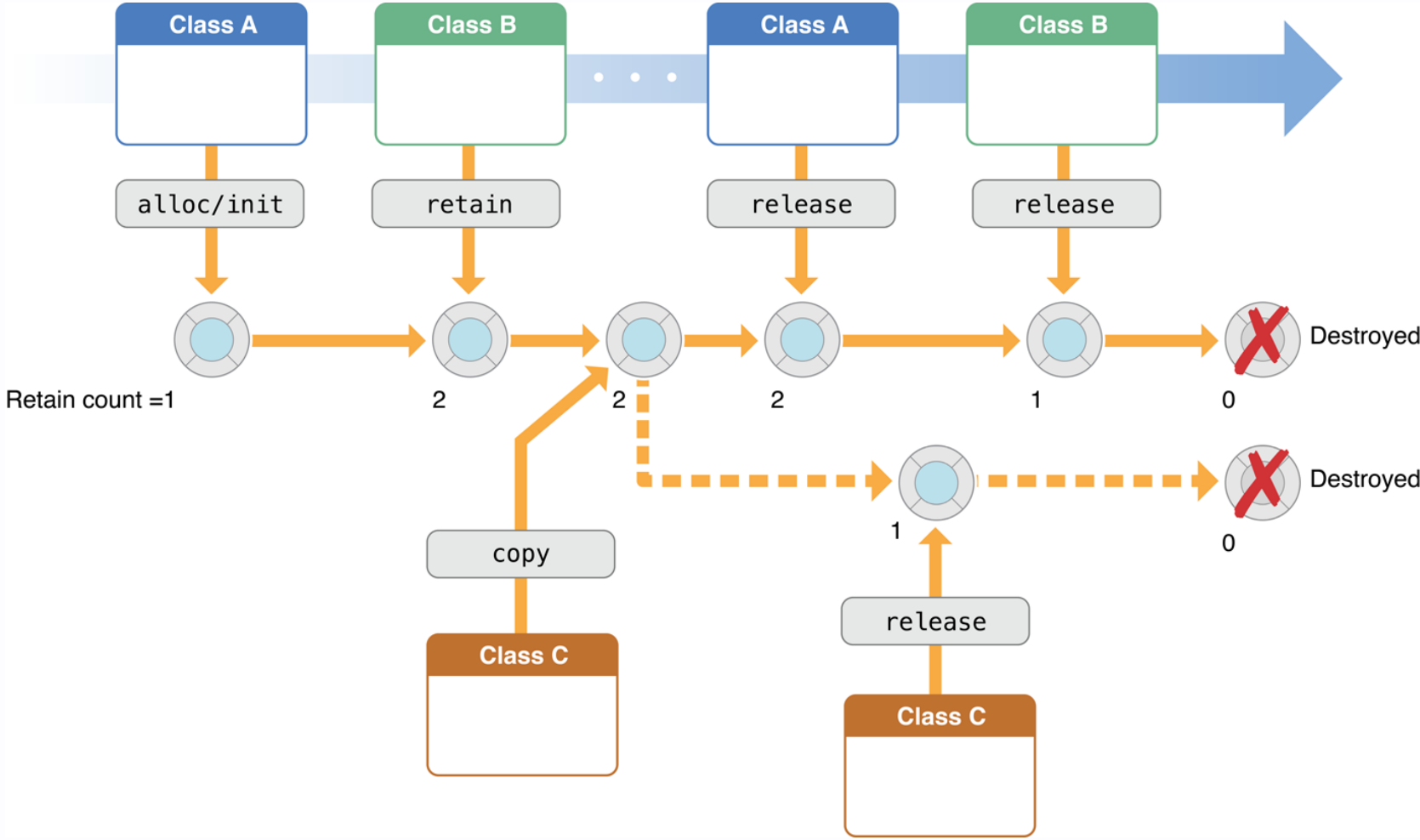


Stage: After Mark Phase



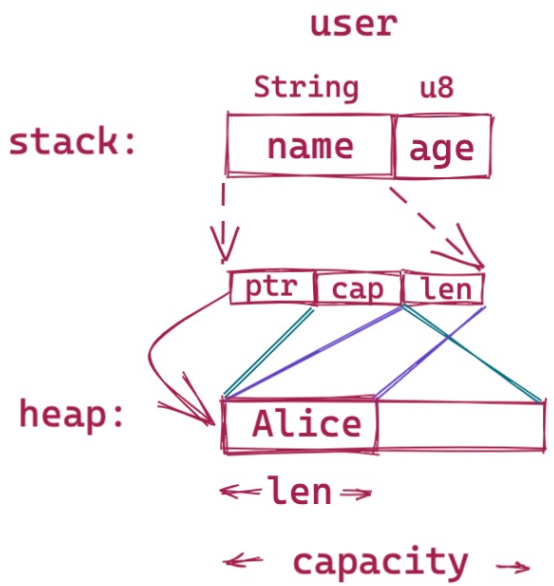
Stage: After Sweep Phase

Lifetime: Heap memory (ARC)

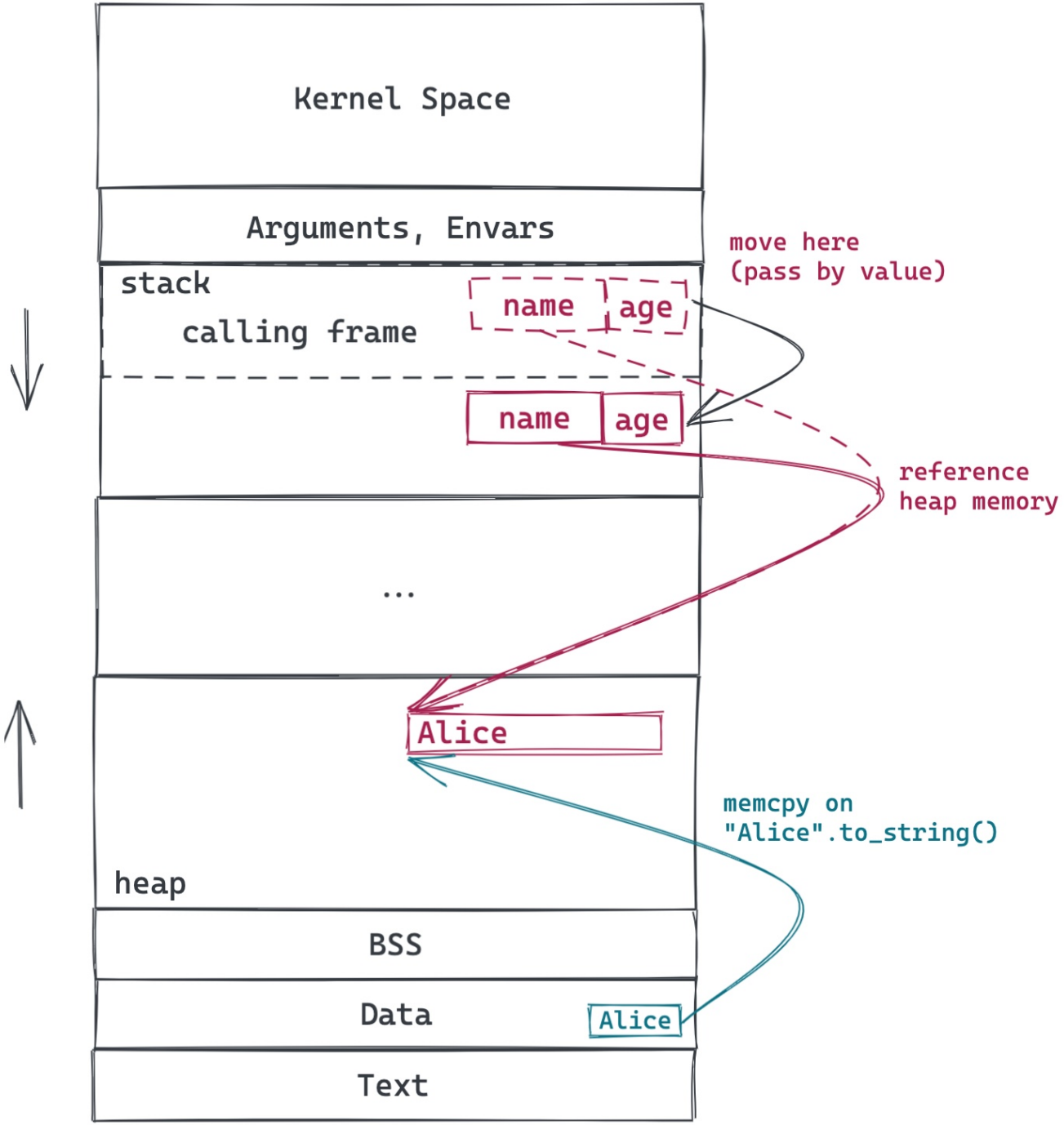


How Rust handles lifetime?

Move semantics

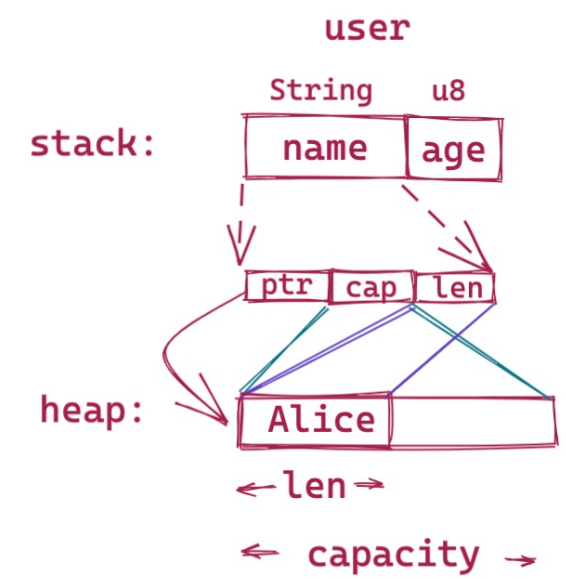



```
{
  let user = User {
    name: "Alice".to_string(),
    age: 20,
  };
  let result = insert(user);
   ...
}
```

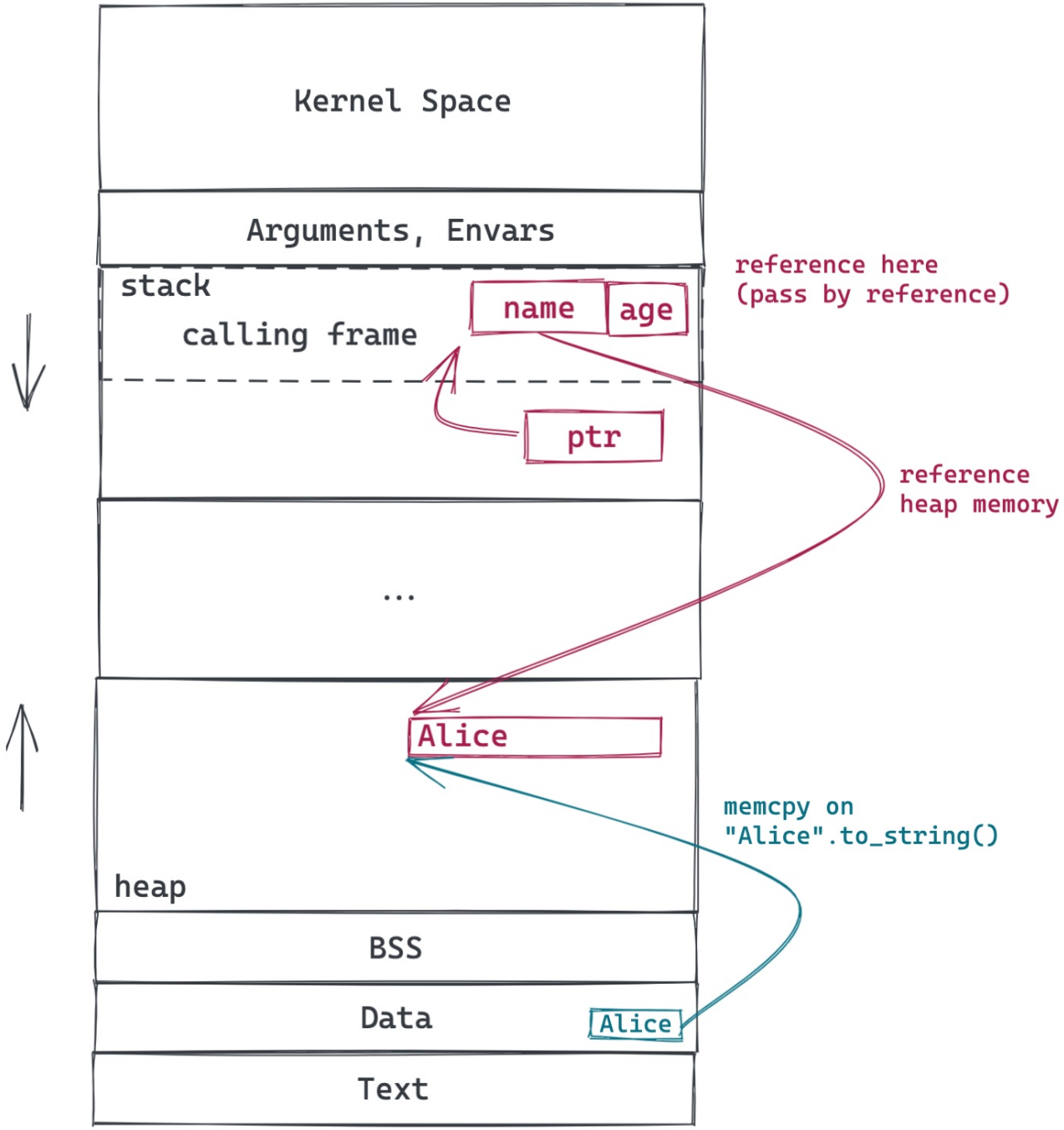


What a bout Borrow?

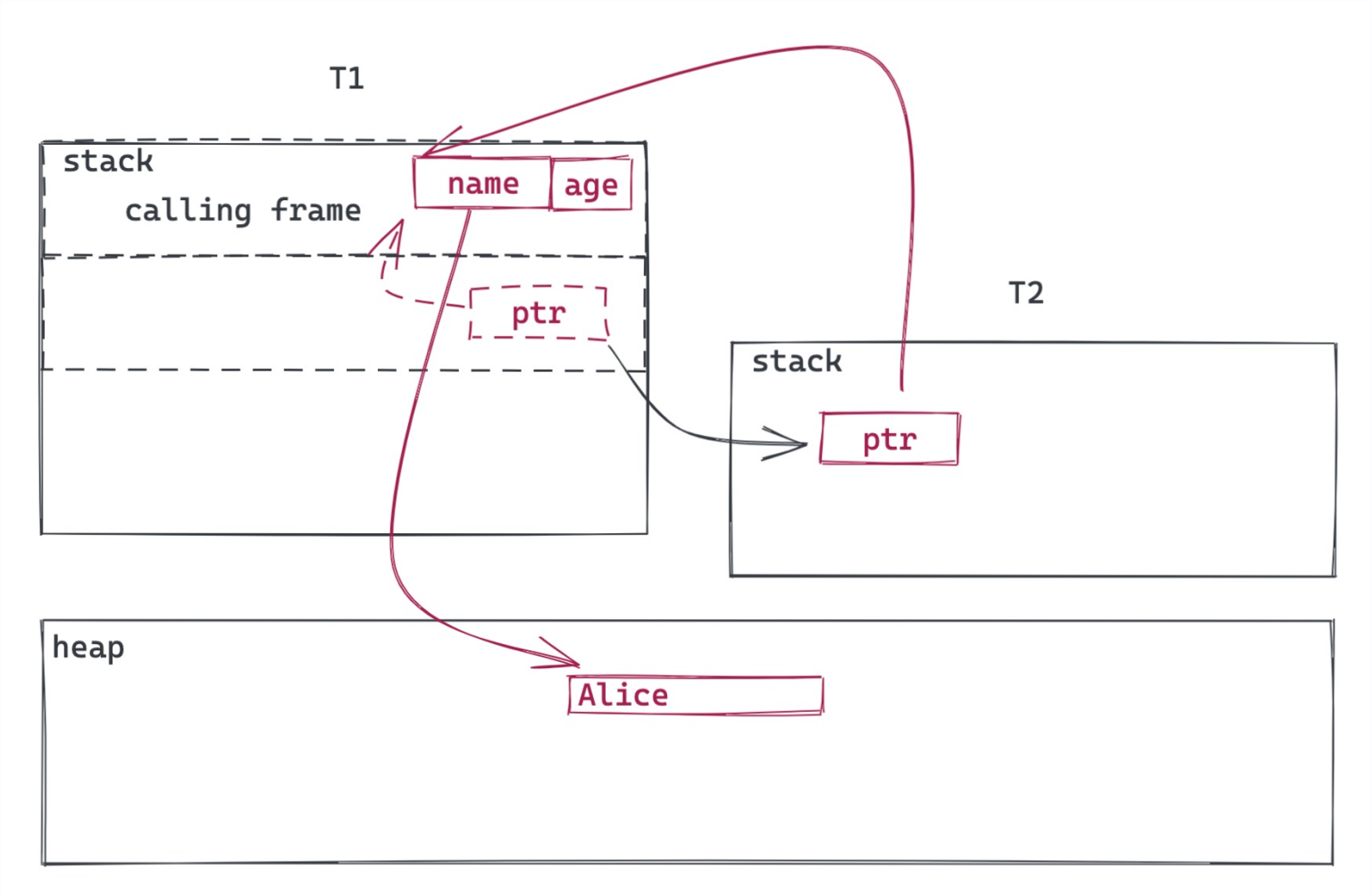
Think about: why does this work in Rust, but not C/C++?




```
{
  let user = User {
    name: "Alice".to_string(),
    age: 20,
  };
  let result = insert(&user);
   ...
}
```



Rust lifetime checker prevents this...



Benefit of lifetime-constrained borrow

- can borrow anything (stack object, heap object)
- safety can be guaranteed at compile-time (no runtime lifetime bookkeeping)
- Rust borrow checker is mostly a lifetime checker

Lifetime Annotation

- similar as generics, but in lowercase starting with `'`
- only need to put annotation when there's conflicts

```
// need explicit lifetime
struct User<'a> {
    name: &'a str,
    ...
}
fn process<T, 'a, 'b>(item1: &'a T, item2: &'b T) {}

// &'a User could be written as &User since on confliction
fn lifetime_example(user: &User) { // --+ Lifetime 'a
    if user.is_authed() {          //   |--+ Lifetime 'b
        let role = user.roles();   //   |  |
                                   //   |  |--+ Lifetime 'c
        verify(&role);             //   |  |  |
                                   //   |  |--+
    }                              //   |--+
}                                  // --+

fn verify(x: &Role) { /*...*/ }
```

Static Lifetime

- 'static
- data included in bss / data / text section
  - constants / static variables
  - string literals
  - functions
- if used as trait bound:
  - the type does not contain any non-static references
  - owned data always passes a `'static` lifetime bound, but reference to the owned data does not

Thread spawn

```
pub fn spawn<F, T>(f: F) -> JoinHandle<T>
where
    F: FnOnce() -> T,
    F: Send + 'static,
    T: Send + 'static,
{
    Builder::new().spawn(f).expect("failed to spawn thread")
}
```

The 'static constraint is a way of saying, roughly, that no borrowed data is permitted in the closure.

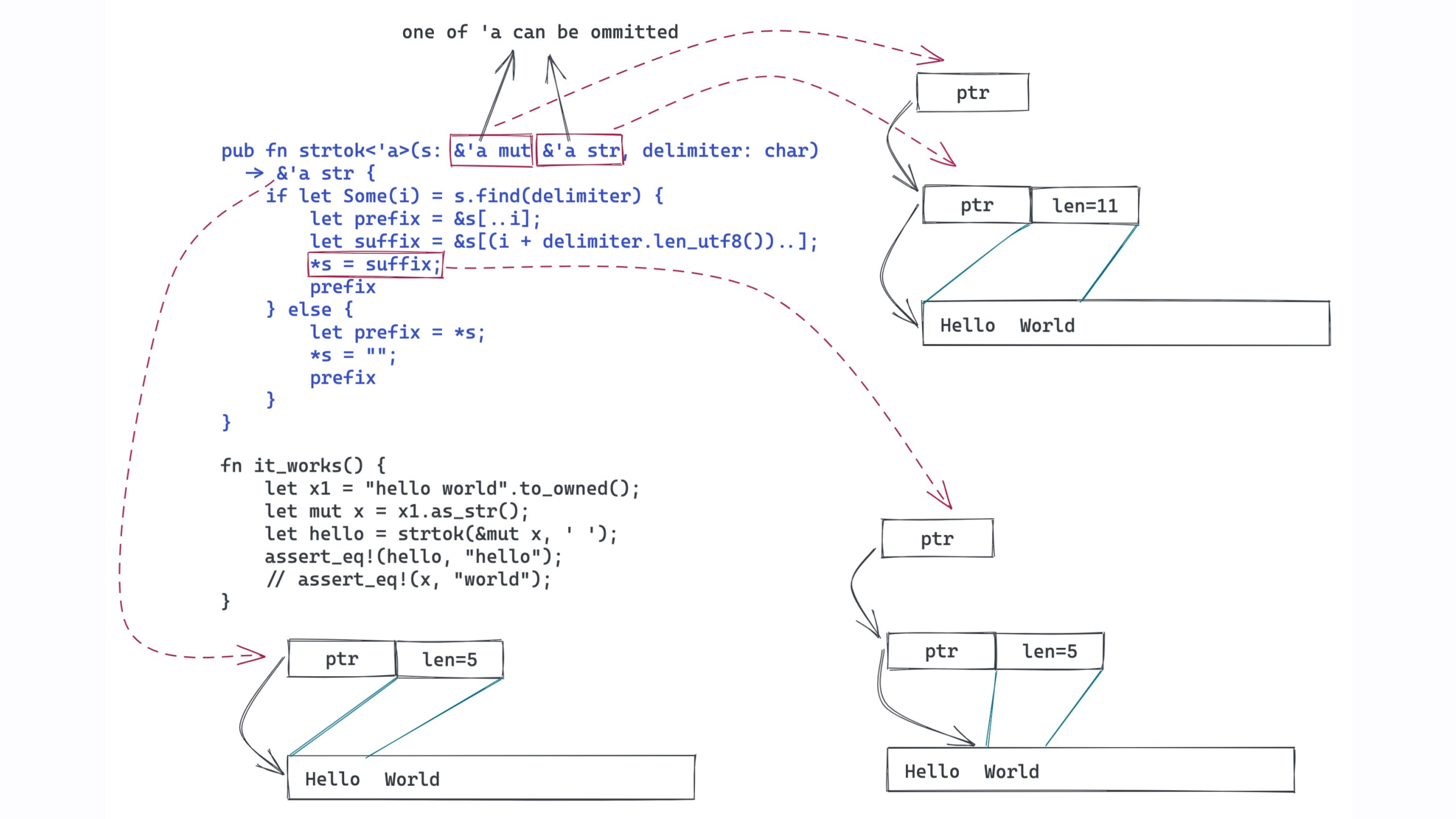
RAII (Resource Acquisition Is Initialization)

- initializing the object will also make sure resource is initialized
- releasing the object will also make sure resource is released

Drop Trait

- memory
- file
- socket
- lock
- any other OS resources

demo


one of 'a can be ommitted
pub fn strtok<'a>(s: &'a mut &'a str, delimiter: char)
-> &'a str {
if let Some(i) = s.find(delimiter) {
let prefix = &s[..i];
let suffix = &s[(i + delimiter.len_utf8())..];
*s = suffix;
prefix
} else {
let prefix = *s;
*s = "";
prefix
}
}
fn it_works() {
let x1 = "hello world".to_owned();
let mut x = x1.as_str();
let hello = strtok(&mut x, ' ');
assert_eq!(hello, "hello");
// assert_eq!(x, "world");
}
ptr
ptr
len=11
Hello World
ptr
ptr
len=5
Hello World
ptr
len=5
Hello World

Mental model

- write the code and defer the complexity about ensuring the code is safe/correct
- comfront the most of the safety/correctness problems upfront
- Mutate can only happen when you own the data, or you have a mutable reference
  - either way, the thread is guaranteed to be the only one with access at a time
- Fearless Refactoring
- reinforce properties well-behaved software exhibits
- sometimes too strict: rust isn't dogmatic about enforcing it

References

- Mark-And-Sweep (Garbage Collection Algorithm)
- Tracing garbage collection
- Swift: Avoiding Memory Leaks by Examples
- Reference counting
- Fearless concurrency with Rust
- Rust means never having to close a socket
- Programming Rust: ownership
- Crust of Rust: lifetime annotation (recommended)

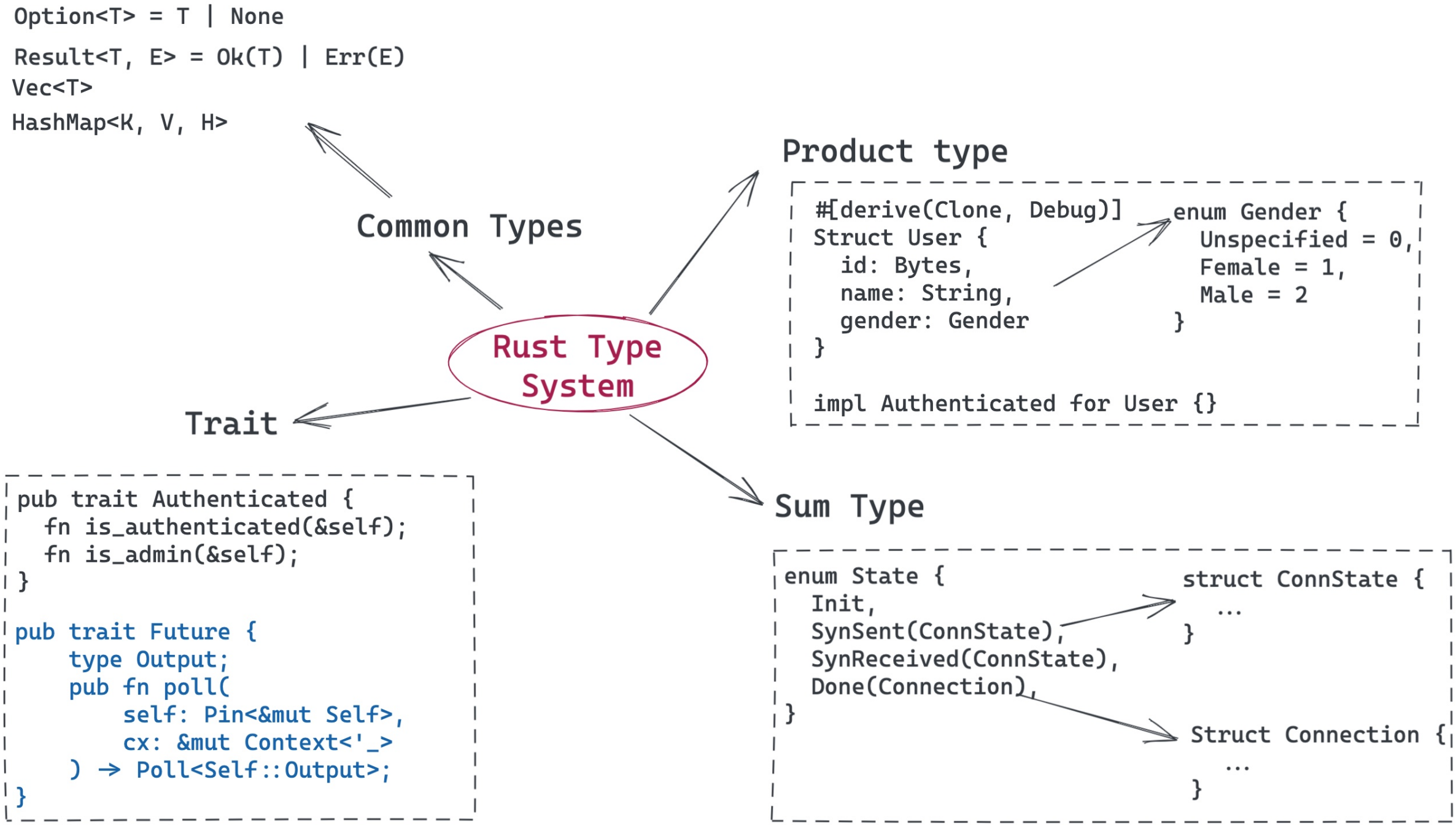
Cost of defects

- Don't introduce defect (this is impossible because humans are fallible).
- Detect and correct defect as soon as the bad key press occurs (within reason: you don't want the programmer to lose too much flow) (milliseconds later).
- At next build / test time (seconds or minutes later).
- When code is shared with others (maybe you push a branch and CI tells you something is wrong) (minutes to days later).
- During code review (minutes to days later).
- When code is integrated (e.g. merged) (minutes to days later).
- When code is deployed (minutes to days or even months later).
- When a bug is reported long after the code has been deployed (weeks to years later).

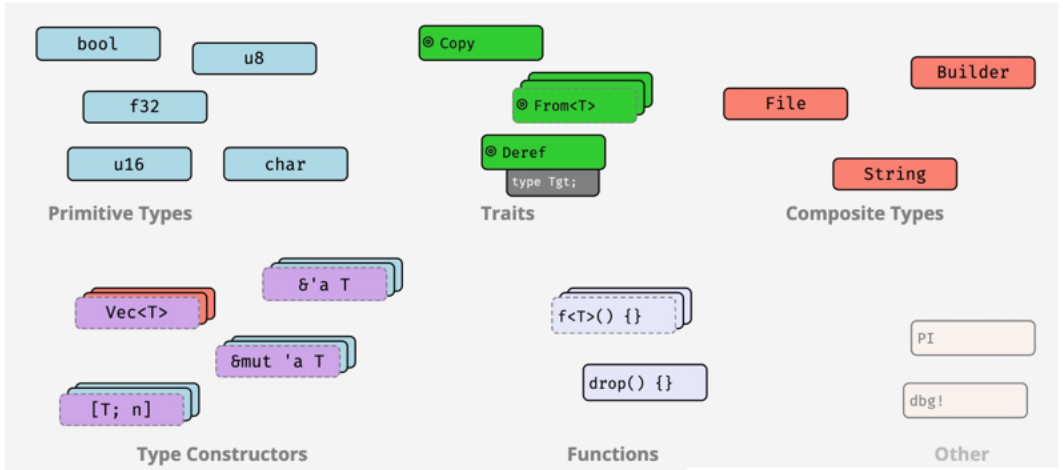
Ownership and Borrow rules

- Use after free: no more (reference can't point to dropped value)
- Buffer underruns, overflows, illegal memory access: no more (reference must be valid and point to owned value)
- memory level data races: no more (single writer or multiple readers)

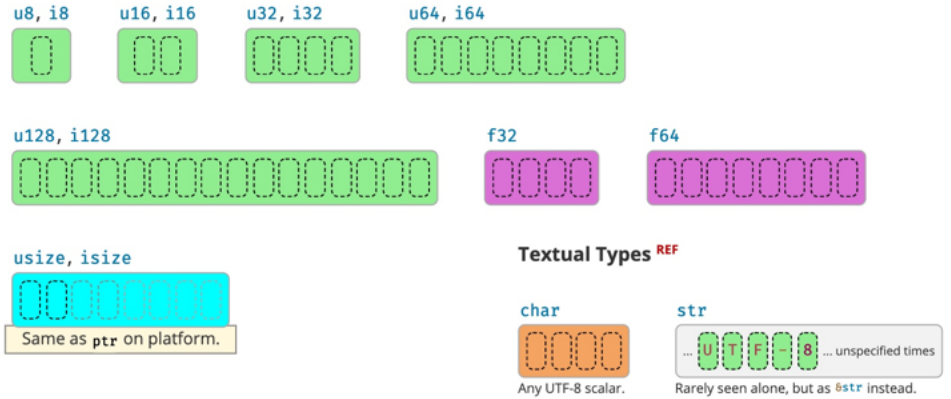
Typesystem and data structures


Option<T> = T | None
Result<T, E> = Ok(T) | Err(E)
Vec<T>
HashMap<K, V, H>
Common Types
Rust Type System
Trait
pub trait Authenticated {
fn is_authenticated(&self);
fn is_admin(&self);
}
pub trait Future {
type Output;
pub fn poll(
self: Pin<&mut Self>,
cx: &mut Context<'_>
) -> Poll<Self::Output>;
}
Product type
#[derive(Clone, Debug)]
Struct User {
id: Bytes,
name: String,
gender: Gender
}
impl Authenticated for User {}
enum Gender {
Unspecified = 0,
Female = 1,
Male = 2
}
Sum Type
enum State {
Init,
SynSent(ConnState),
SynReceived(ConnState),
Done(Connection),
}
struct ConnState {
...
}
Struct Connection {
...
}

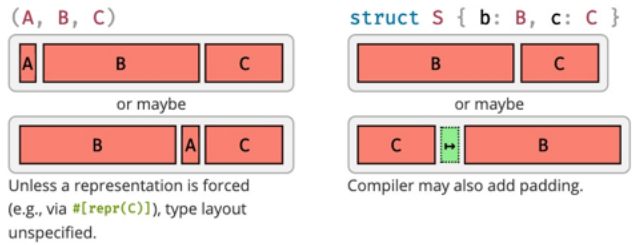
Memory layout



Numeric Types REF

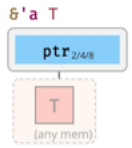


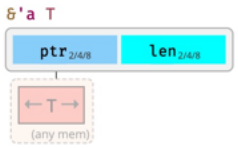
Textual Types REF

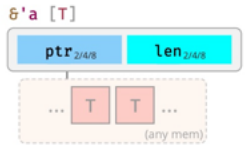


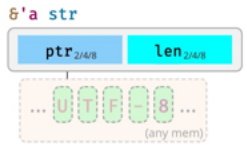
Pointer Meta

Many reference and pointer types can carry an extra field, **pointer metadata**. [STD] It can be the element- or byte-length of the target, or a pointer to a *vtable*. Pointers with meta are called **fat**, otherwise **thin**.

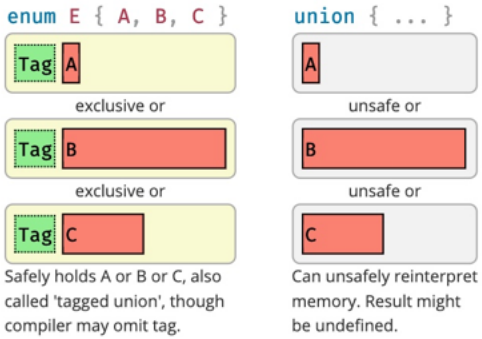


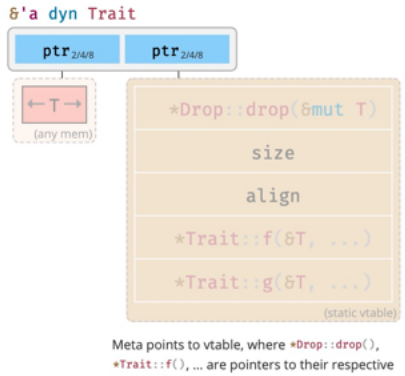






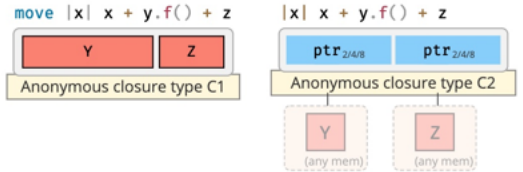
These **sum types** hold a value of one of their sub types:

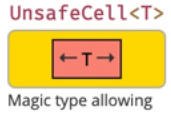


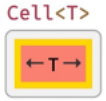


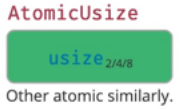
Closures

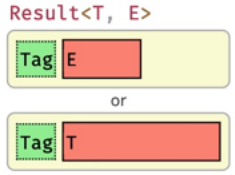
Ad-hoc functions with an automatically managed data block **capturing** [REF] environment where closure was defined. For example:

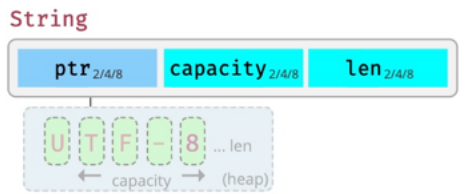


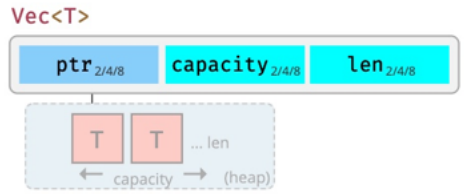




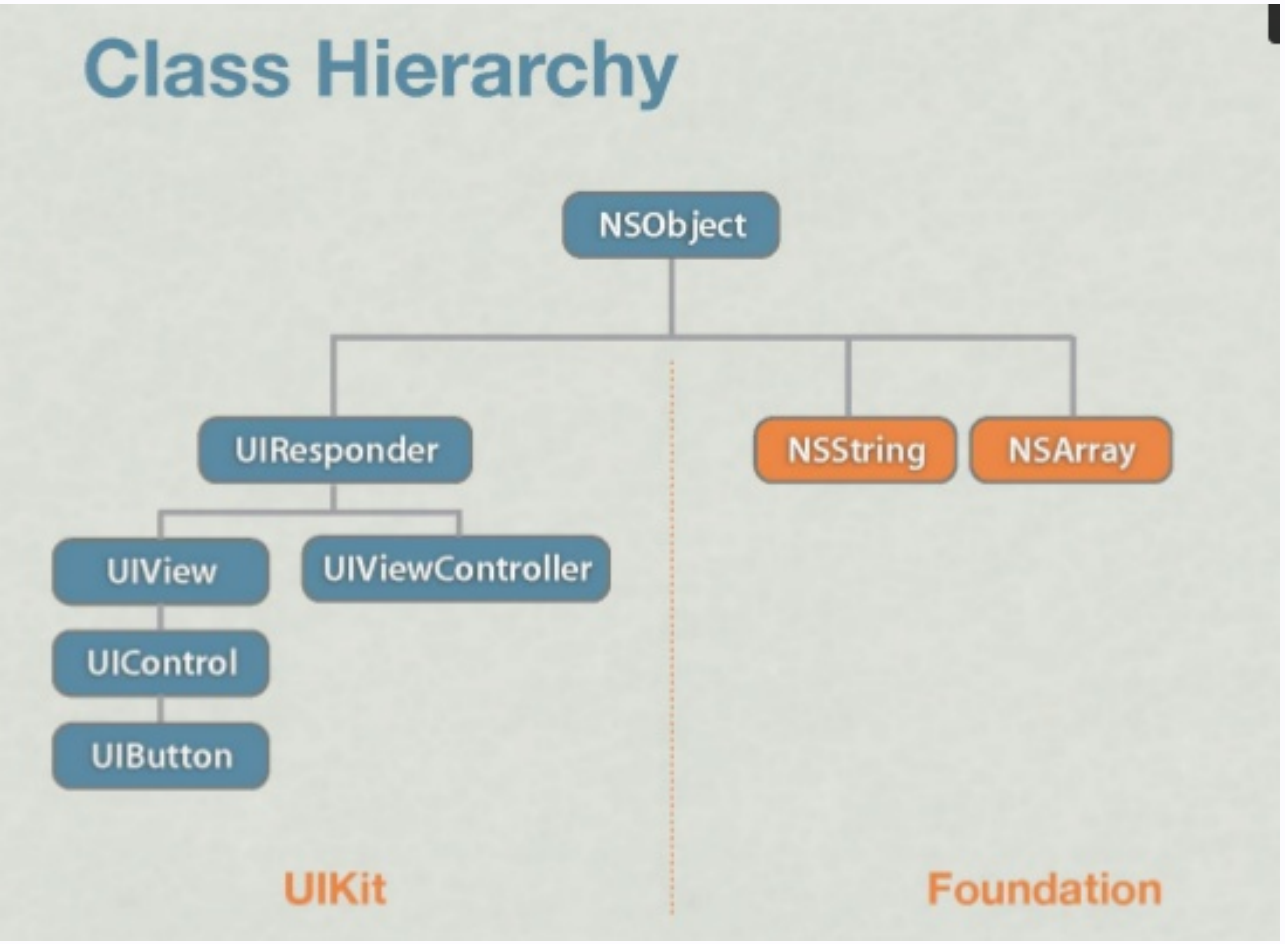








Why not object oriented?



Class is awesome

- Encapsulation
- Access control
- Abstraction
- Namespace
- Extensibility

Class has problems

- inheritance is pretty limited - choose superclass well!
- know what/how to override (and when not to)
- superclass may have properties
  - you have to accept it
  - initialization burden
  - don't break assumptions of superclass
- hard to reuse outside the hierachy (composition over inheritance)

Solution: Trait (Typeclass)

refer to the actual data

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }

    ...
}
```

methods must be implemented

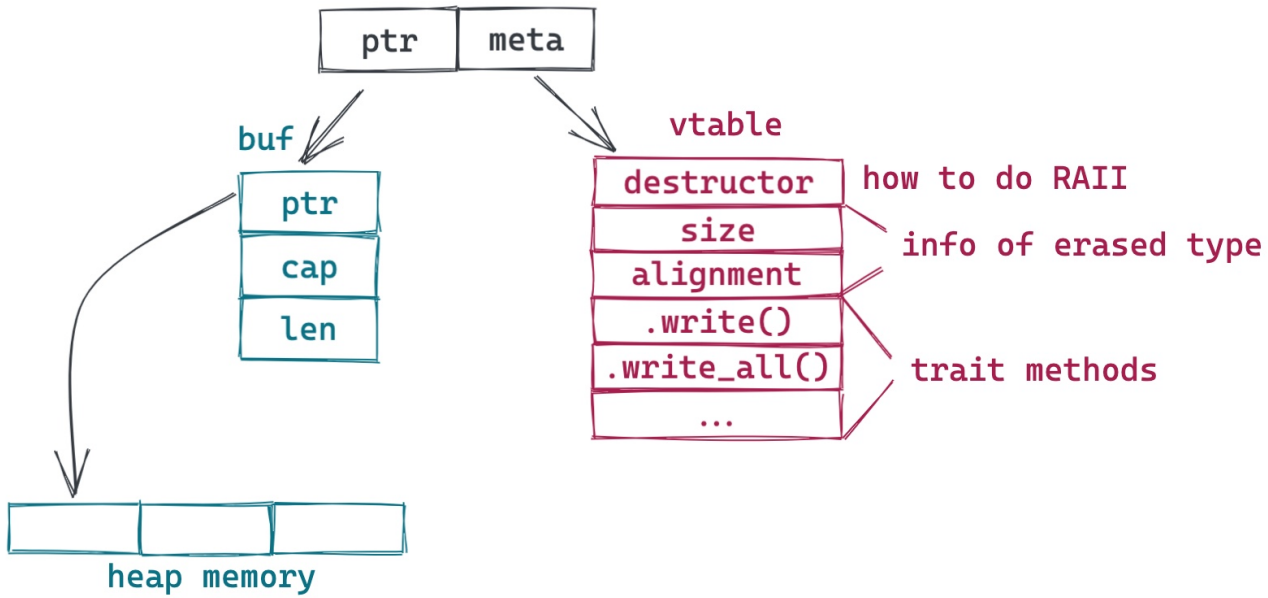
methods have a default impl

```
struct Sink {
    total: usize,
}

impl Write for Sink {
    fn write(&mut self, buf: &[u8]) -> Result<usize> {
        self.total += buf.len();
        Ok(buf.len())
    }

    fn flush(&mut self) -> Result<()> {
        Ok(())
    }
}
```

Coherence rule: either trait or type must be new in the current crate.

when implementing trait methods, you can access the actual data

```rust
use std::io::Write;

let mut buf: Vec<u8> = vec![];
let writer: Write = buf; // `Write` does not have a constant size

let writer: &mut Write = &mut buf; // ok. Trait object
```



Trait Object

- Unlike java, you can't assign a value to a trait (no implicit reference!!!)
- trait object is a fat pointer (automatically converted)
  - normal pointer reference to the value
  - vtable (vtable pointer)
    - unlike C++, it is not a ptr in struct
- dynamic dispatch

```rust
pub trait Formatter {
    fn format(&self, input: &mut str) -> bool;
}

struct MarkdownFormatter;
impl Formatter for MarkdownFormatter {
    fn format(&self, input: &mut str) -> bool { todo!() }
}

struct RustFormatter;
impl Formatter for RustFormatter {
    fn format(&self, input: &mut str) -> bool { todo!() }
}

struct HtmlFormatter;
impl Formatter for HtmlFormatter {
    fn format(&self, input: &mut str) -> bool { todo!() }
}

pub fn format(input: &mut str, formatters: Vec<Box<dyn Formatter>>) {
    for formatter in formatters {
        formatter.format(input);
    }
}
```

```rust
pub trait Iterator {
    type Item;
    pub fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

associate type

```rust
pub trait From<T> {
    pub fn from(T) -> Self;
}
```

generic type

```rust
trait Person {
    fn name(&self) -> String;
}

trait Student: Person {
    fn university(&self) -> String;
}

trait Programmer {
    fn fav_language(&self) -> String;
}

trait CompSciStudent: Programmer + Student {
    fn git_username(&self) -> String;
}
```

super trait

trait composition

More about trait

- associated type
- generics
- supertrait
- trait composition

Generics

References

- All about trait objects

Concurrency - primitives

Concurrency - async/await

Networking and security

FFI with C/Elixir/Swift/Java

WASM/WASI

Rust for real-world problems

May the Rust be with you