# 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

- 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

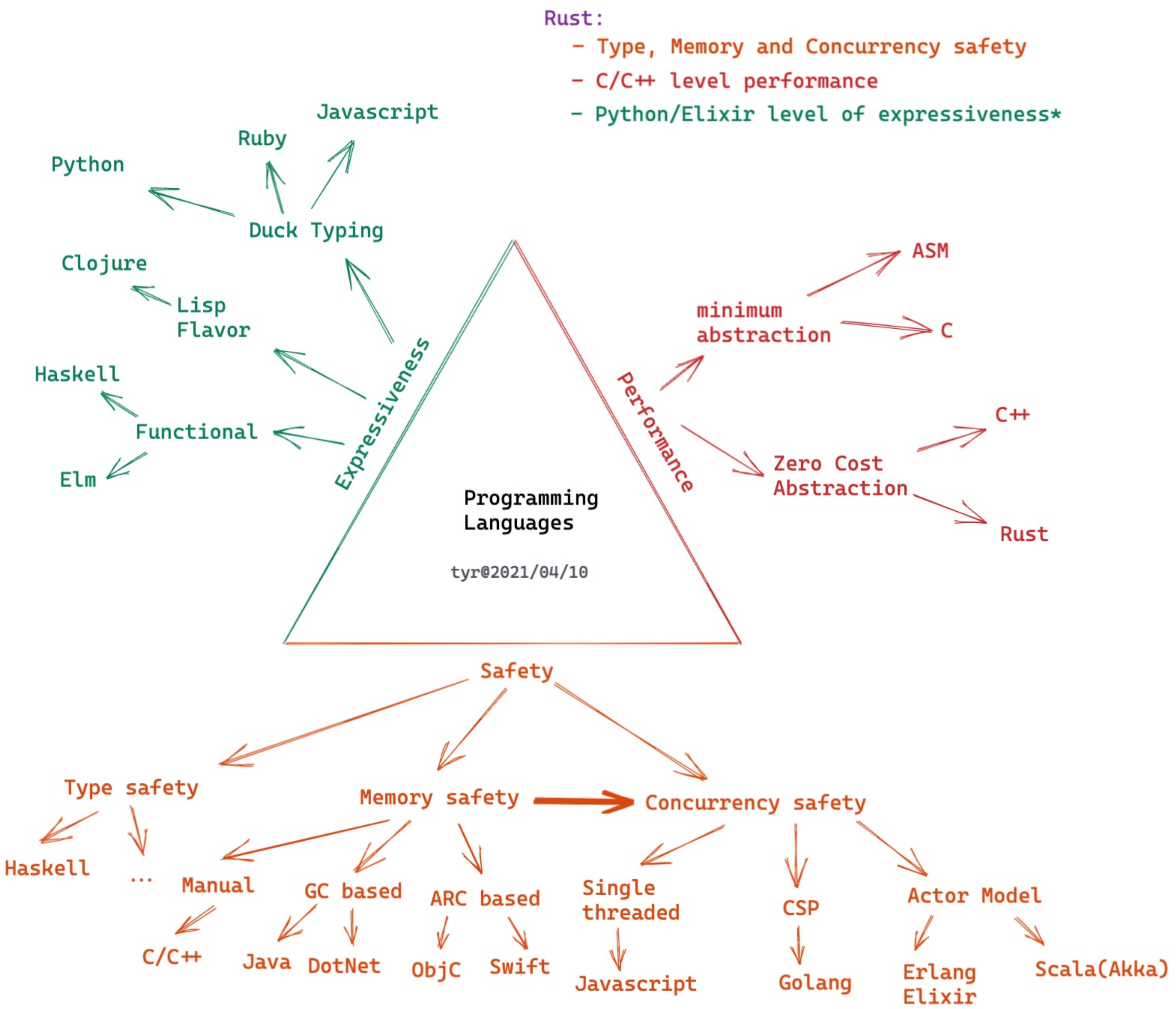
- Approachability
- Availability
- Compatibility
- Composability
- Debuggability
- Expressiveness
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- Performance
- Portability
- Productivity
- Resilliency
- Rigor
- Safety
- Security
- Simplicity
- Stability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

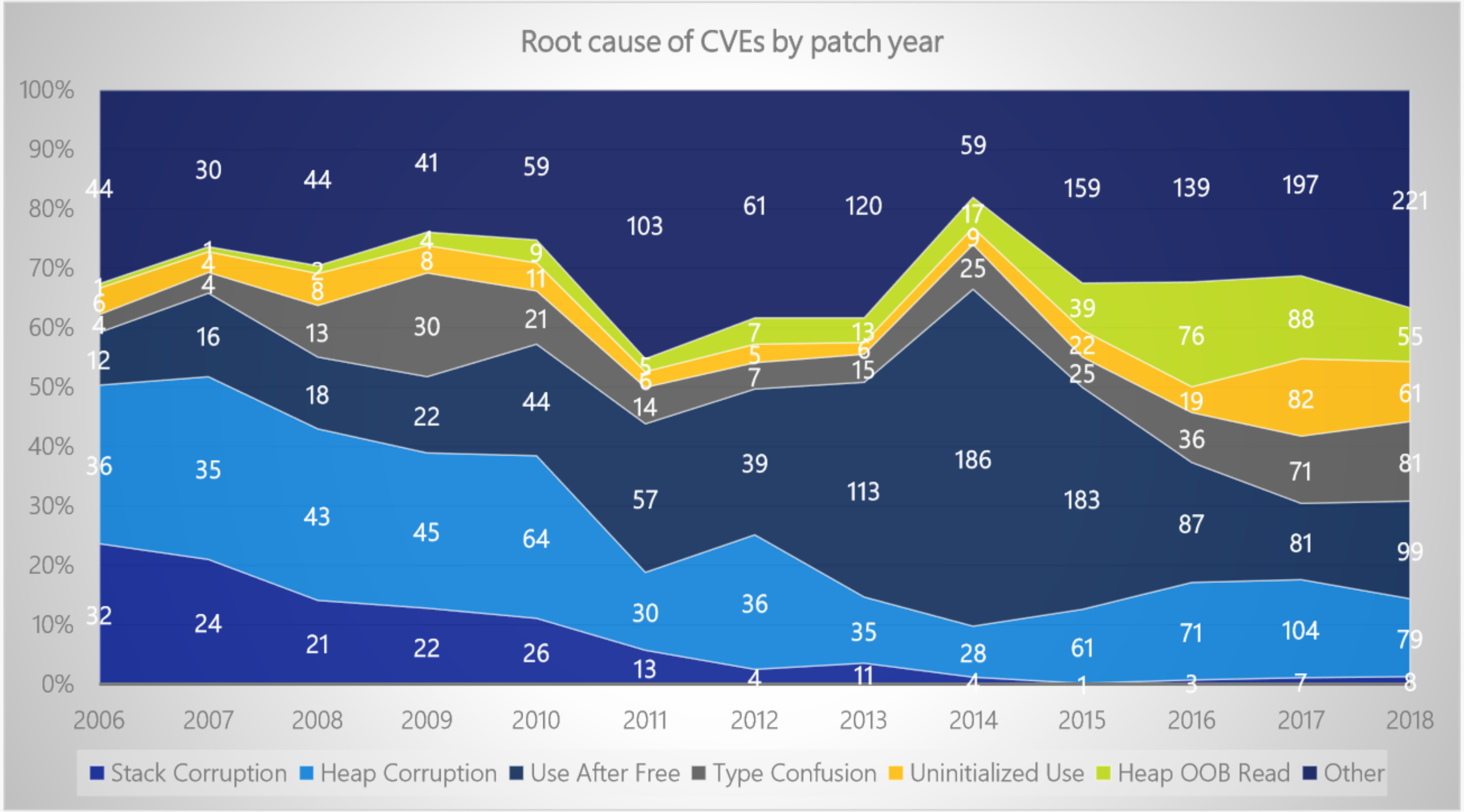
# Rust

- 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*
Programming Languages
tyr@2021/04/10
Expressiveness
Duck Typing
Python
Ruby
Javascript
Lisp Flavor
Clojure
Functional
Haskell
Elm
Performance
minimum abstraction
ASM
C
Zero Cost Abstraction
C++
Rust
Safety
Type safety
Haskell
...
Memory safety
Manual
C/C++
GC based
Java
DotNet
ARC based
ObjC
Swift
Concurrency safety
Single threaded
Javascript
CSP
Golang
Actor Model
Erlang
Elixir
Scala(Akka)

# 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

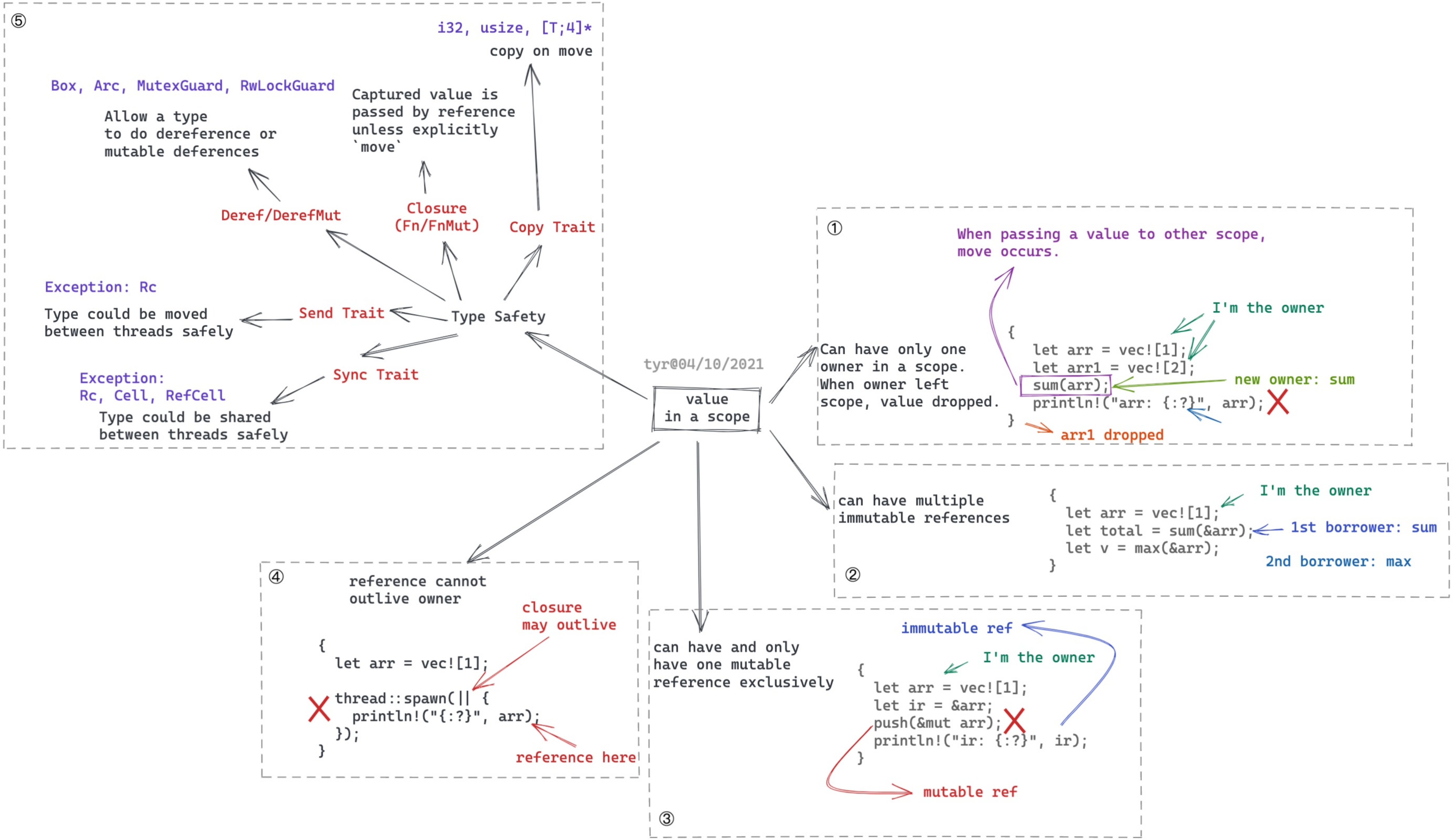
# Concurrency safety

- 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?

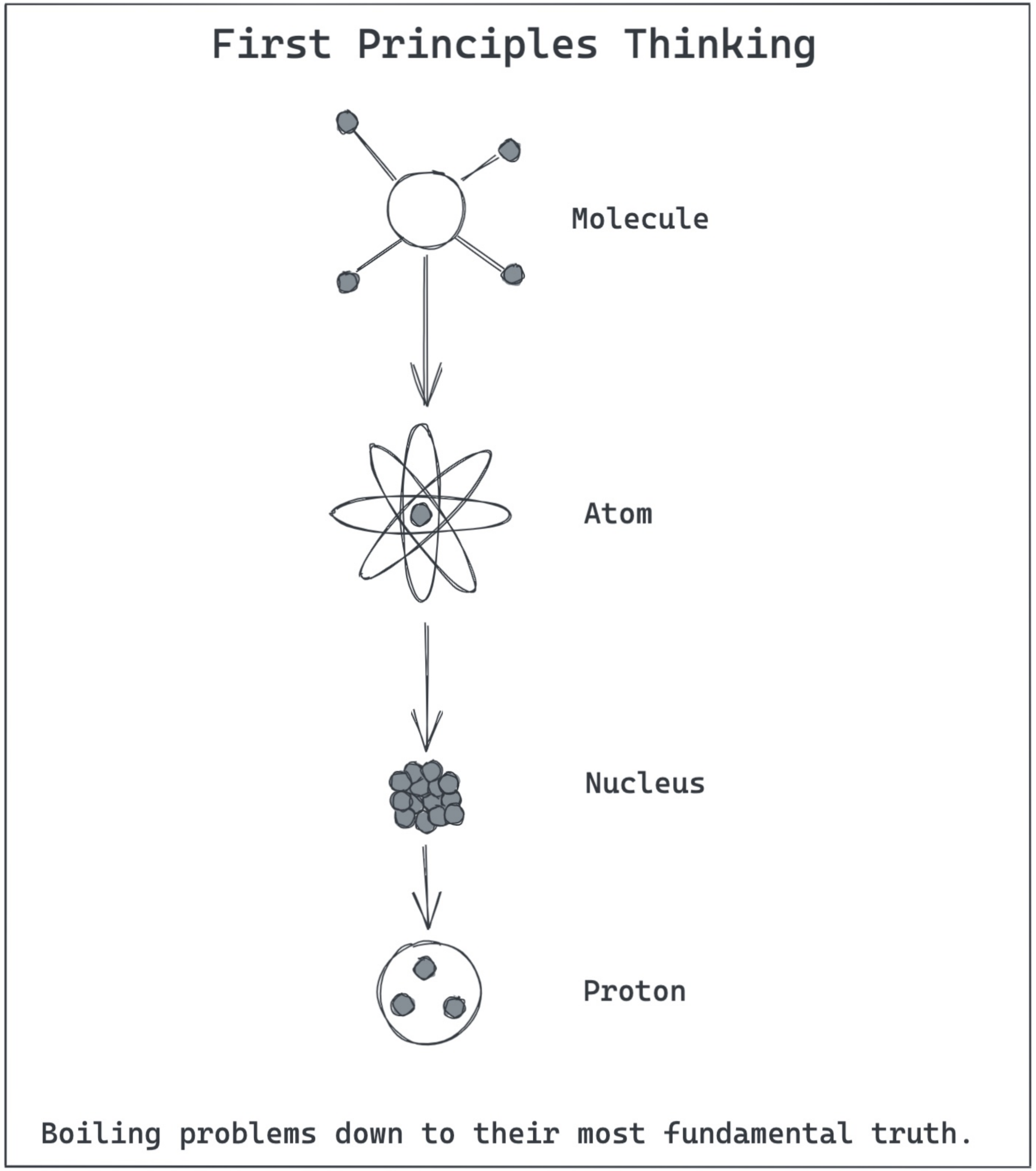


tyr@04/10/2021
value in a scope
①
When passing a value to other scope, move occurs.
Can have only one owner in a scope. When owner left scope, value dropped.
I'm the owner
new owner: sum
arr1 dropped
{
let arr = vec![1];
let arr1 = vec![2];
sum(arr);
println!("arr: {:?}", arr);
}
②
can have multiple immutable references
I'm the owner
1st borrower: sum
2nd borrower: max
{
let arr = vec![1];
let total = sum(&arr);
let v = max(&arr);
}
③
can have and only have one mutable reference exclusively
immutable ref
I'm the owner
mutable ref
{
let arr = vec![1];
let ir = &arr;
push(&mut arr);
println!("ir: {:?}", ir);
}
④
reference cannot outlive owner
closure may outlive
reference here
{
let arr = vec![1];
thread::spawn(|| {
println!("{:?}", arr);
});
}
⑤
Type Safety
Copy Trait
i32, usize, [T;4]*
copy on move
Closure (Fn/FnMut)
Captured value is passed by reference unless explicitly `move`
Deref/DerefMut
Box, Arc, MutexGuard, RwLockGuard
Allow a type to do dereference or mutable deferences
Send Trait
Exception: Rc
Type could be moved between threads safely
Sync Trait
Exception: Rc, Cell, RefCell
Type could be shared between threads safely

# 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
}
```

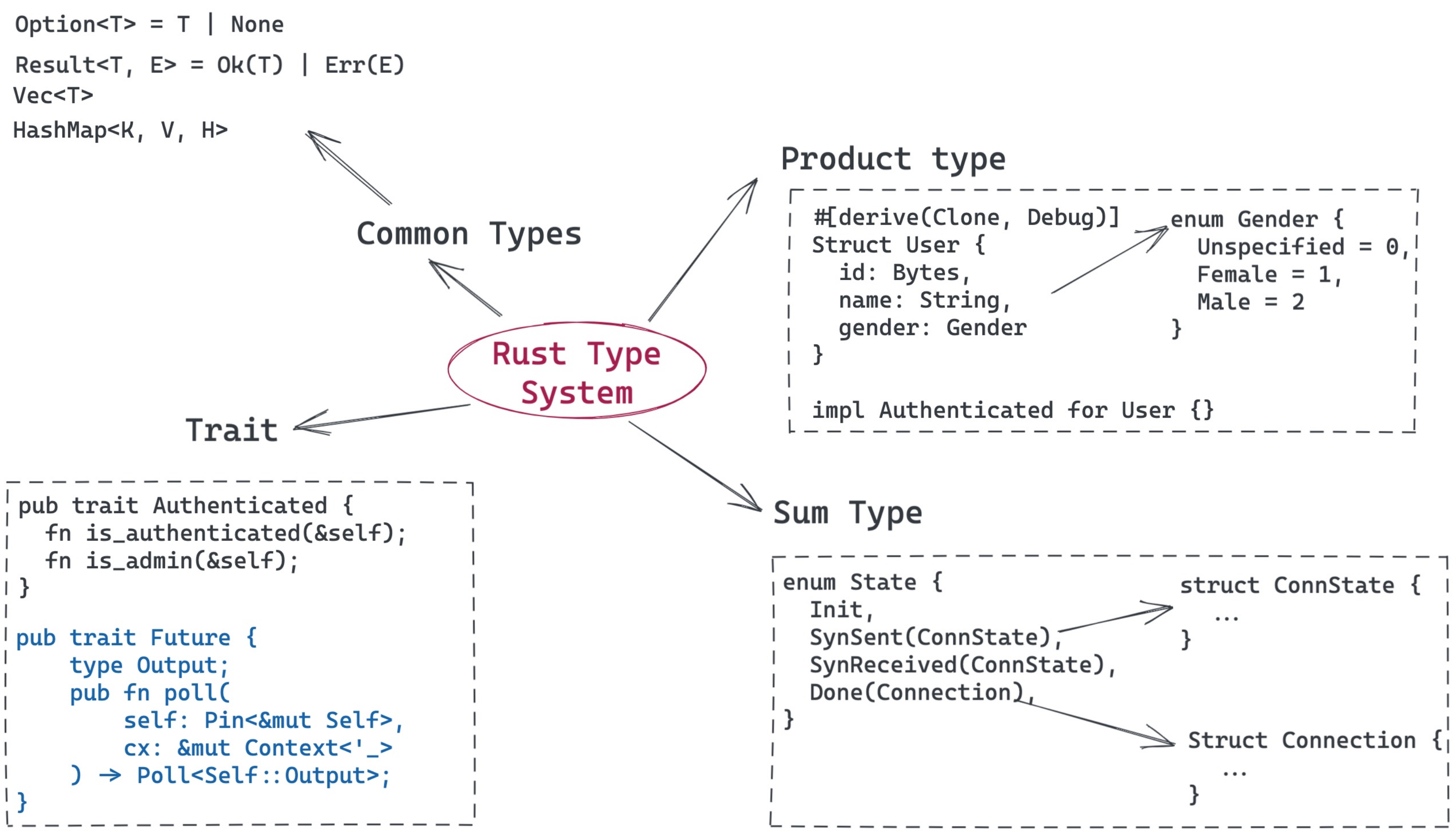



# Recap

- 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?

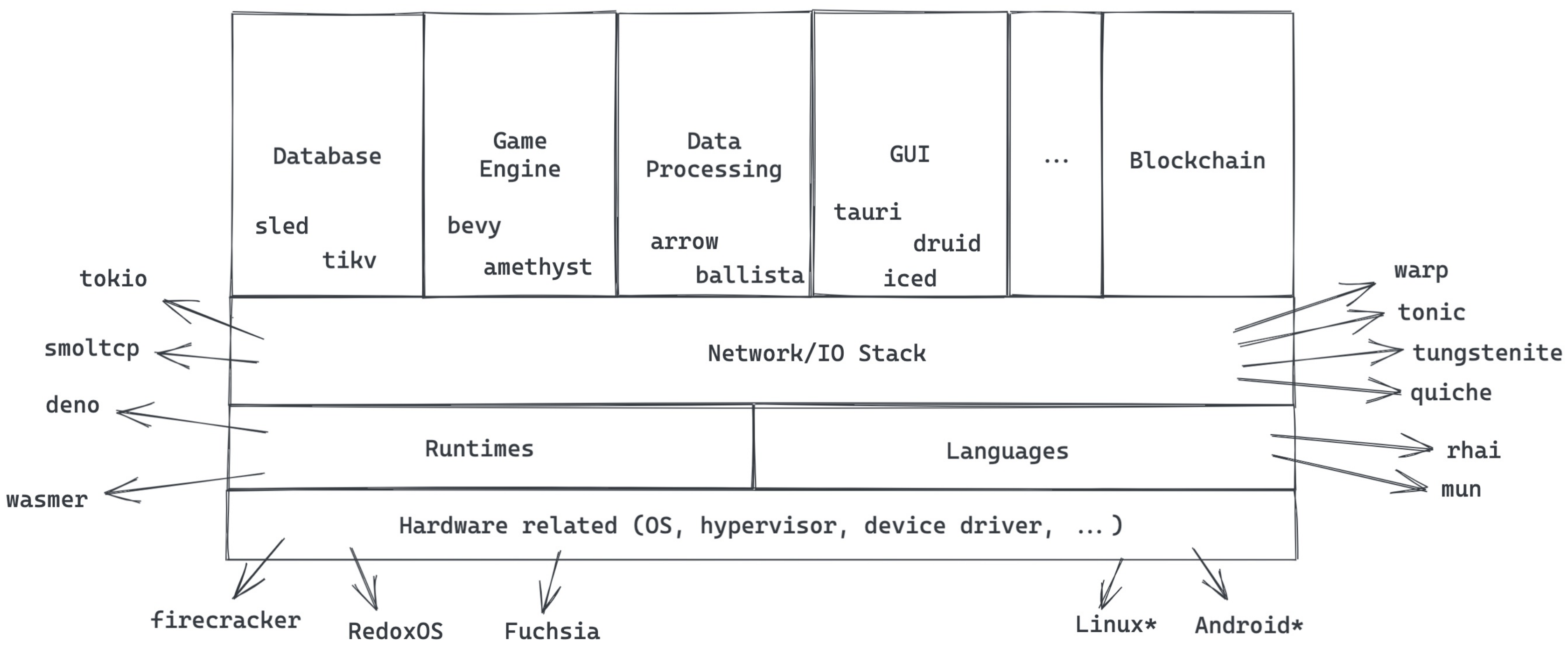
```toml
# client configuration

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



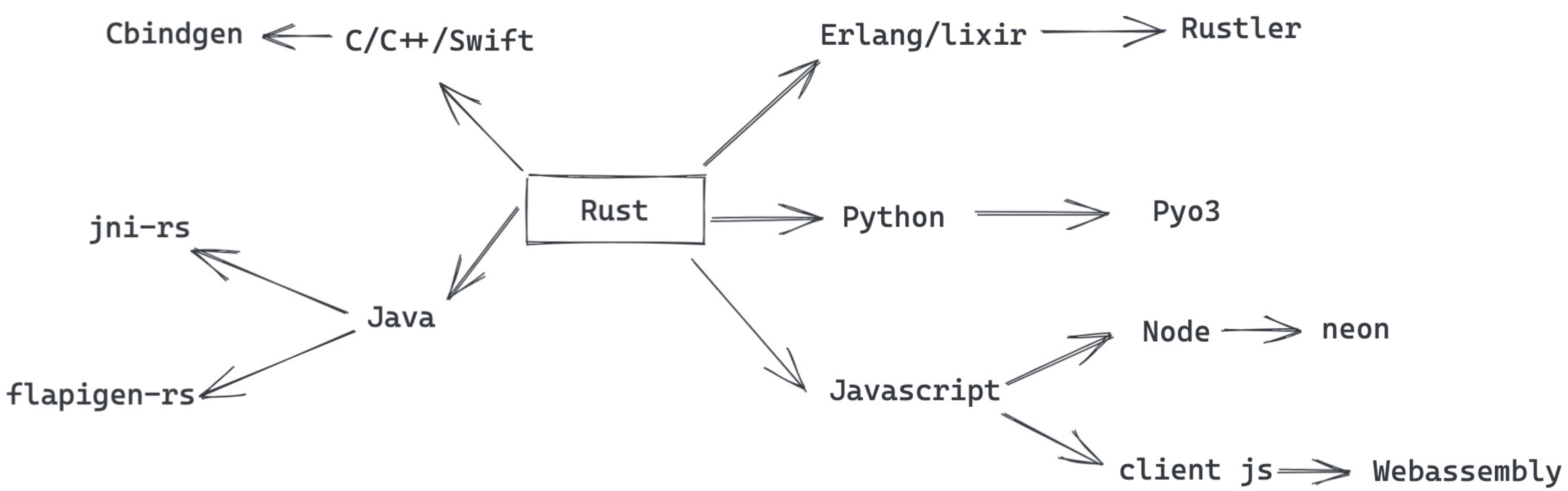
# Should I use Rust?

- 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

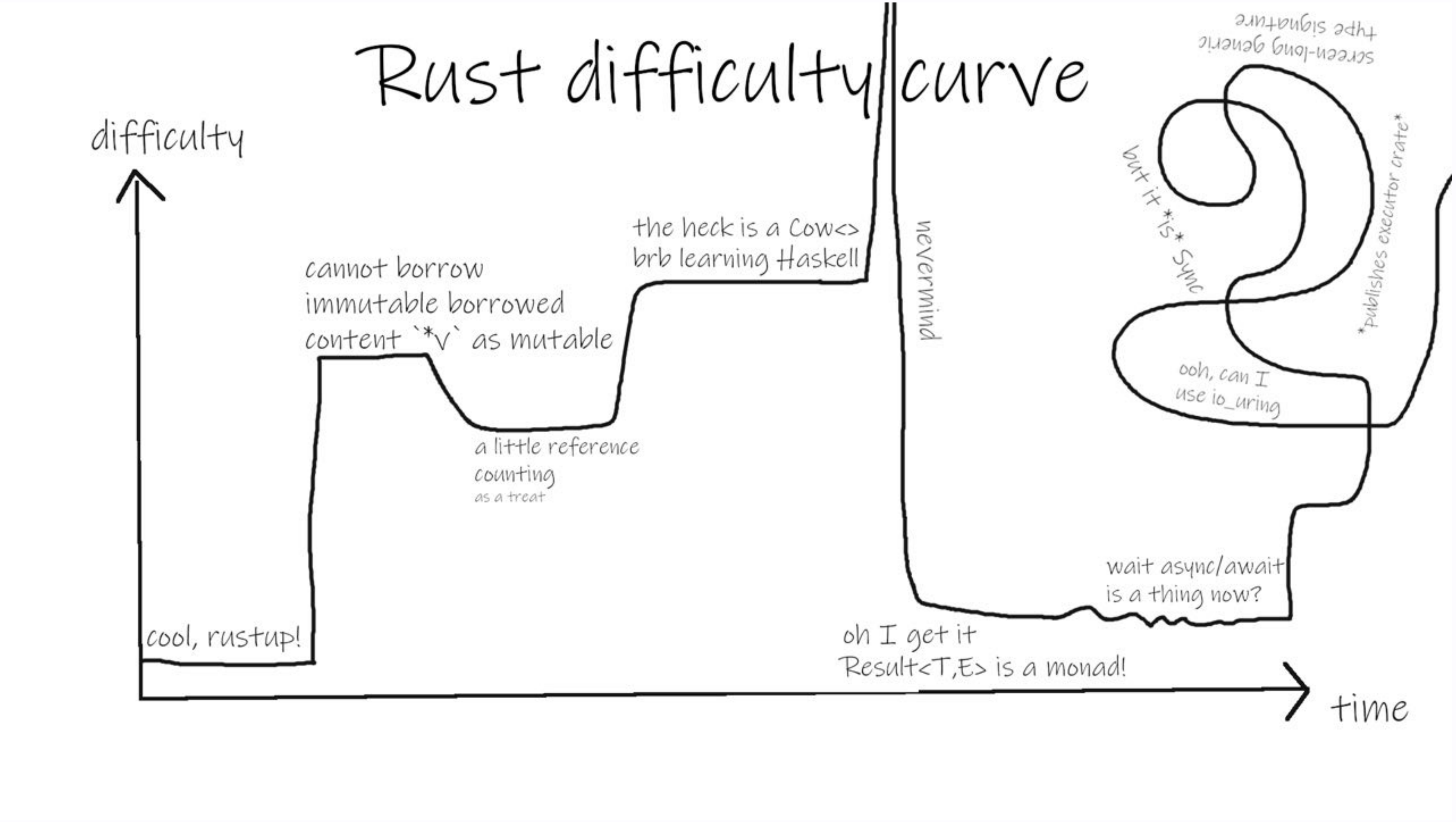

Cbindgen
C/C++/Swift
Erlang/lixir
Rustler
jni-rs
Rust
Python
Pyo3
Java
Node
neon
flapigen-rs
Javascript
client js
Webassembly

# 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

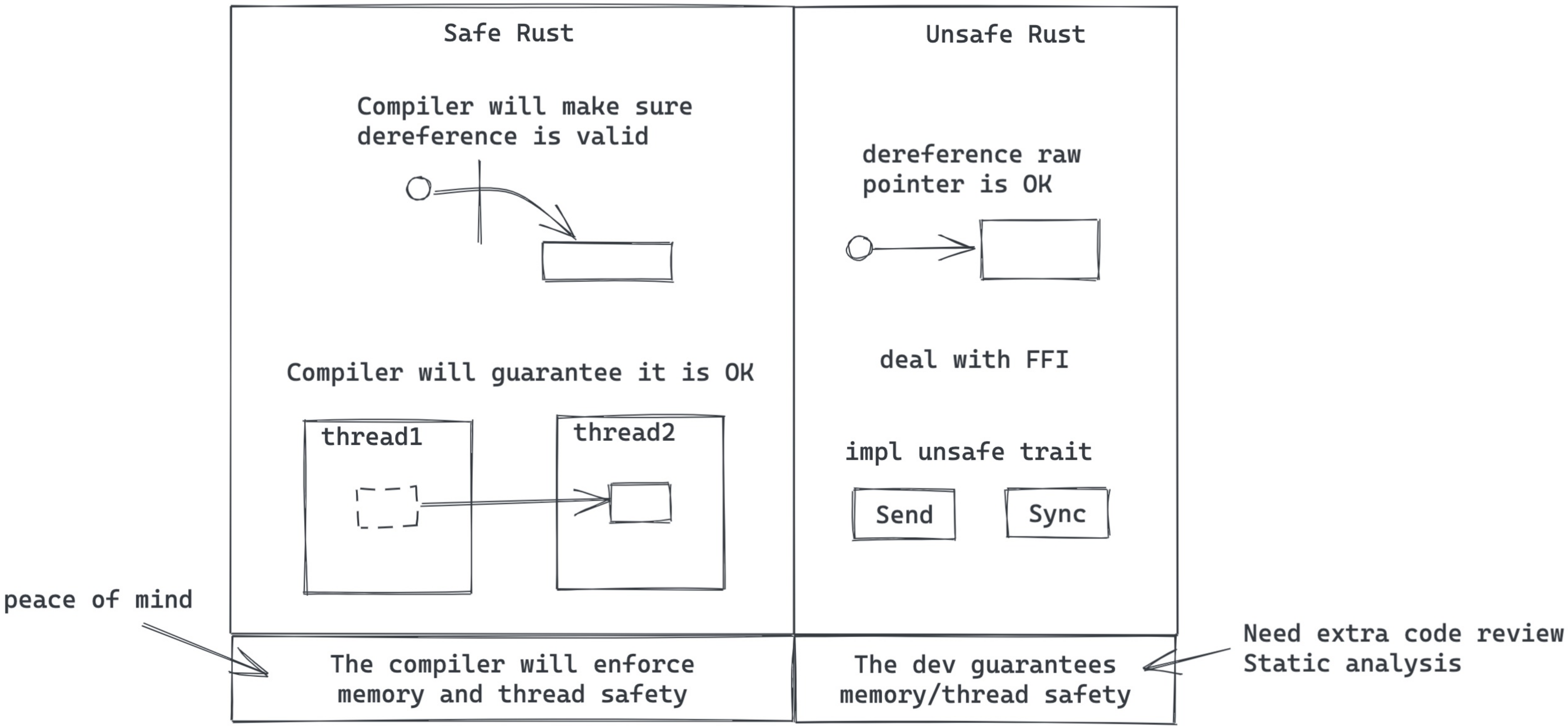
# 1. Rust is super hard to learn...



# 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...

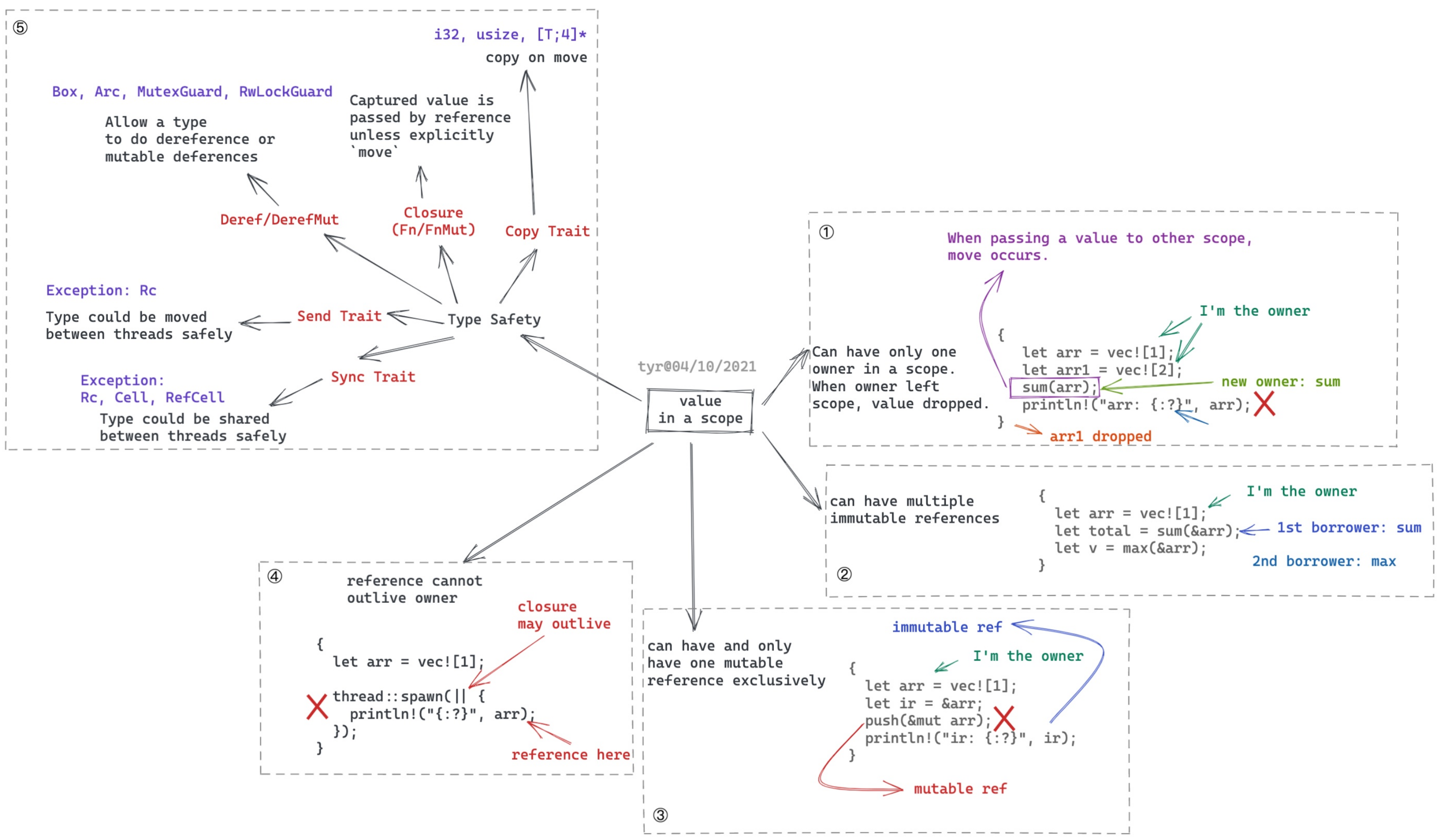


# References

- 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

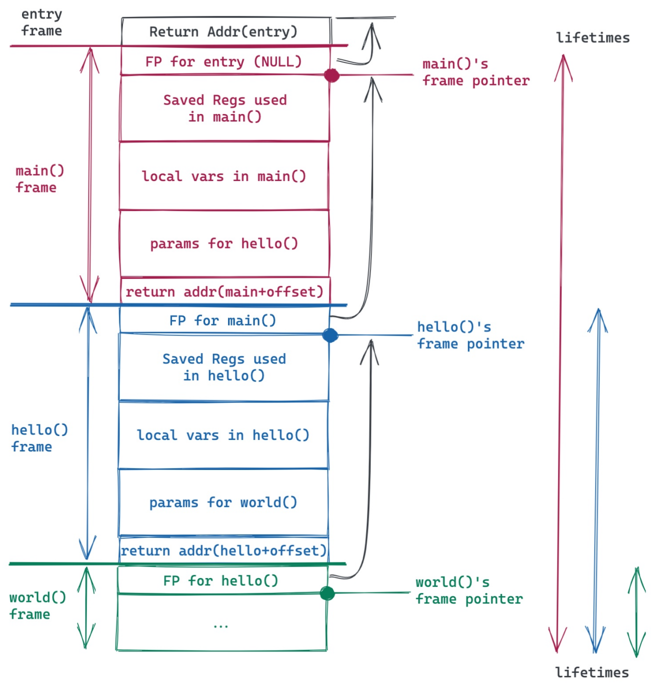
# Ownership/Borrow Rules Review



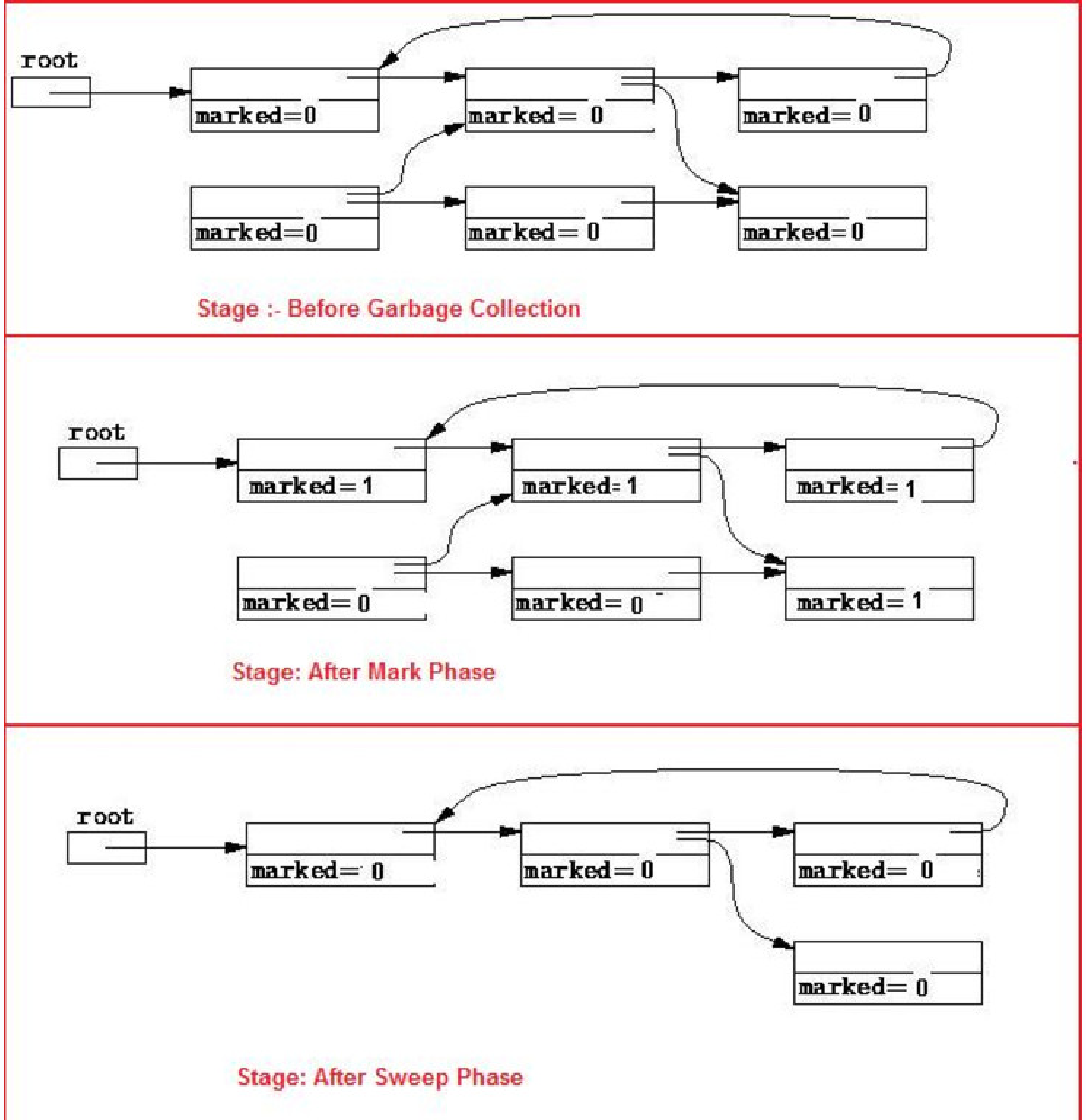
# Lifetime, not a new idea

# Lifetime: Stack memory

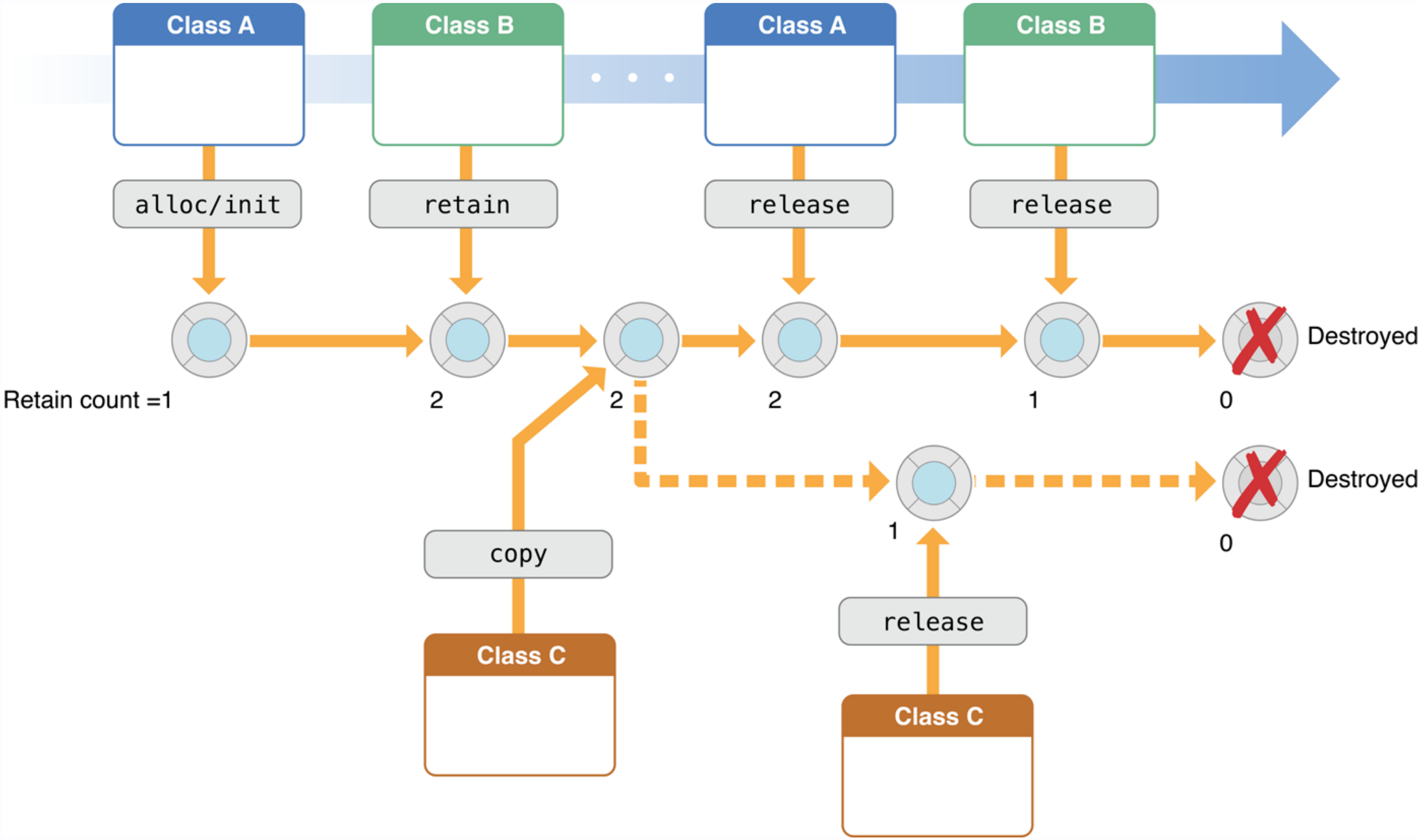
```
❯ cat stack.c
#include <stdio.h>
static int VALUE = 42;

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);
}

/tmp
❯ arm-none-eabi-objdump -d stack.o

stack.o:     file format elf32-littlearm


Disassembly of section .text:

00000000 <world>:
   0:   e92d4800        push    {fp, lr}
   4:   e28db004        add     fp, sp, #4
   8:   e24dd008        sub     sp, sp, #8
   c:   e50b0008        str     r0, [fp, #-8]
  10:   e50b100c        str     r1, [fp, #-12]
  14:   e51b200c        ldr     r2, [fp, #-12]
  18:   e51b1008        ldr     r1, [fp, #-8]
  1c:   e59f0010        ldr     r0, [pc, #16]   ; 34 <world+0x34>
  20:   ebfffffe        bl      0 <printf>
  24:   e1a00000        nop                     ; (mov r0, r0)
  28:   e24bd004        sub     sp, fp, #4
  2c:   e8bd4800        pop     {fp, lr}
  30:   e12fff1e        bx      lr
  34:   00000000        .word   0x00000000

00000038 <hello>:
  38:   e92d4800        push    {fp, lr}
  3c:   e28db004        add     fp, sp, #4
  40:   e24dd010        sub     sp, sp, #16
  44:   e50b0010        str     r0, [fp, #-16]
  48:   e59f3030        ldr     r3, [pc, #48]   ; 80 <hello+0x48>
  4c:   e50b3008        str     r3, [fp, #-8]
  50:   e59f302c        ldr     r3, [pc, #44]   ; 84 <hello+0x4c>
  54:   e5933000        ldr     r3, [r3]
  58:   e51b2010        ldr     r2, [fp, #-16]
  5c:   e0823003        add     r3, r2, r3
  60:   e50b300c        str     r3, [fp, #-12]
  64:   e51b100c        ldr     r1, [fp, #-12]
  68:   e51b0008        ldr     r0, [fp, #-8]
  6c:   ebfffffe        bl      0 <world>
  70:   e1a00000        nop                     ; (mov r0, r0)
  74:   e24bd004        sub     sp, fp, #4
  78:   e8bd4800        pop     {fp, lr}
  7c:   e12fff1e        bx      lr
  80:   00000008        .word   0x00000008
  84:   00000000        .word   0x00000000

00000088 <main>:
  88:   e92d4800        push    {fp, lr}
  8c:   e28db004        add     fp, sp, #4
  90:   e3a00002        mov     r0, #2
  94:   ebfffffe        bl      38 <hello>
  98:   e3a03000        mov     r3, #0
  9c:   e1a00003        mov     r0, r3
  a0:   e24bd004        sub     sp, fp, #4
  a4:   e8bd4800        pop     {fp, lr}
  a8:   e12fff1e        bx      lr
```



# Lifetime: Heap memory (tracing GC)

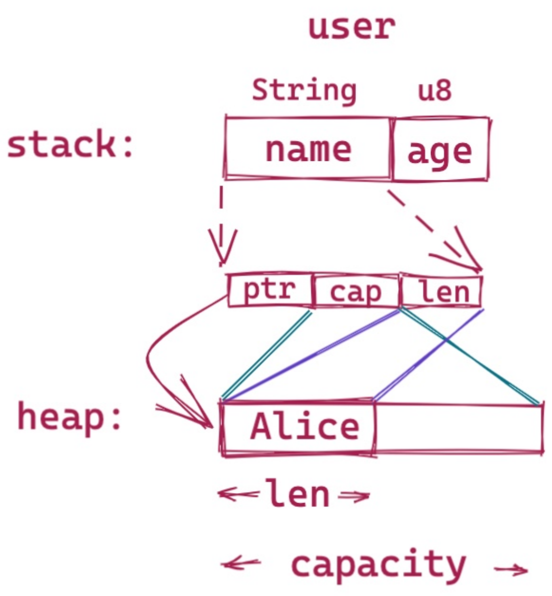


# Lifetime: Heap memory (ARC)

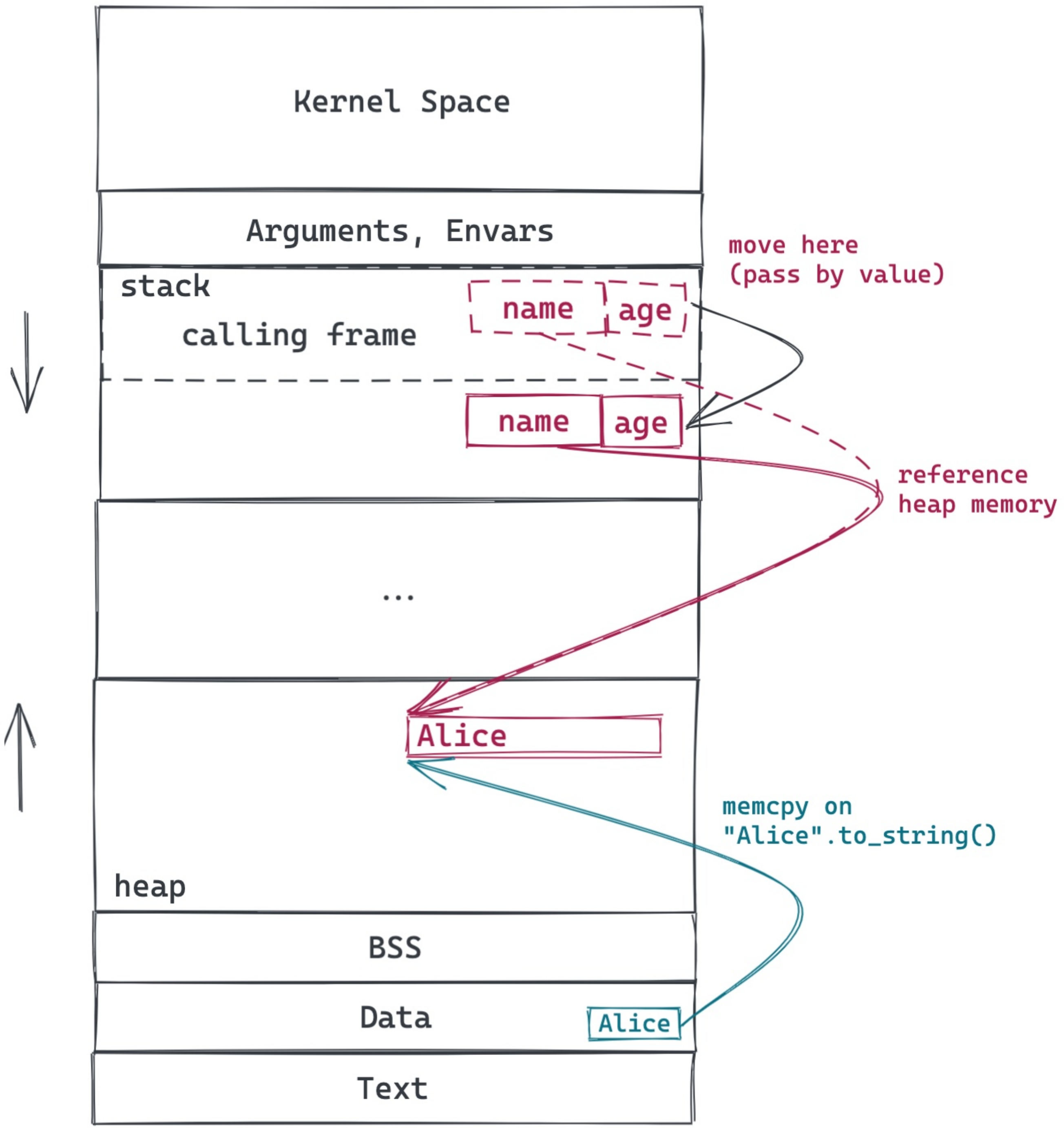


# How Rust handles lifecycle?

# Move semantics

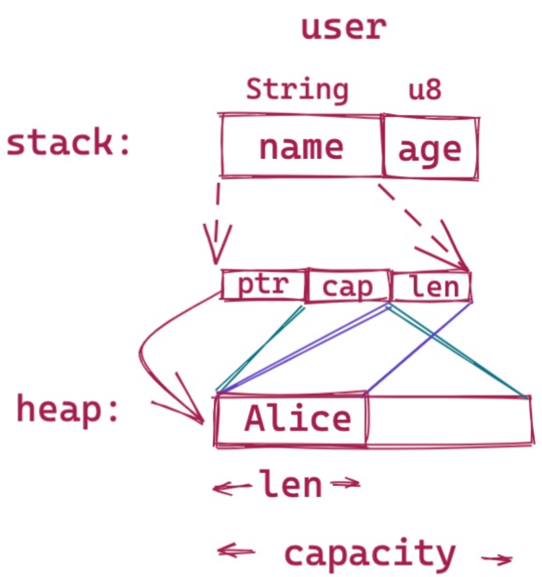



```
{
  let user = User {
    name: "Alice".to_string(),
    age: 20,
  };
  let result = insert(user);
  ...
}
```

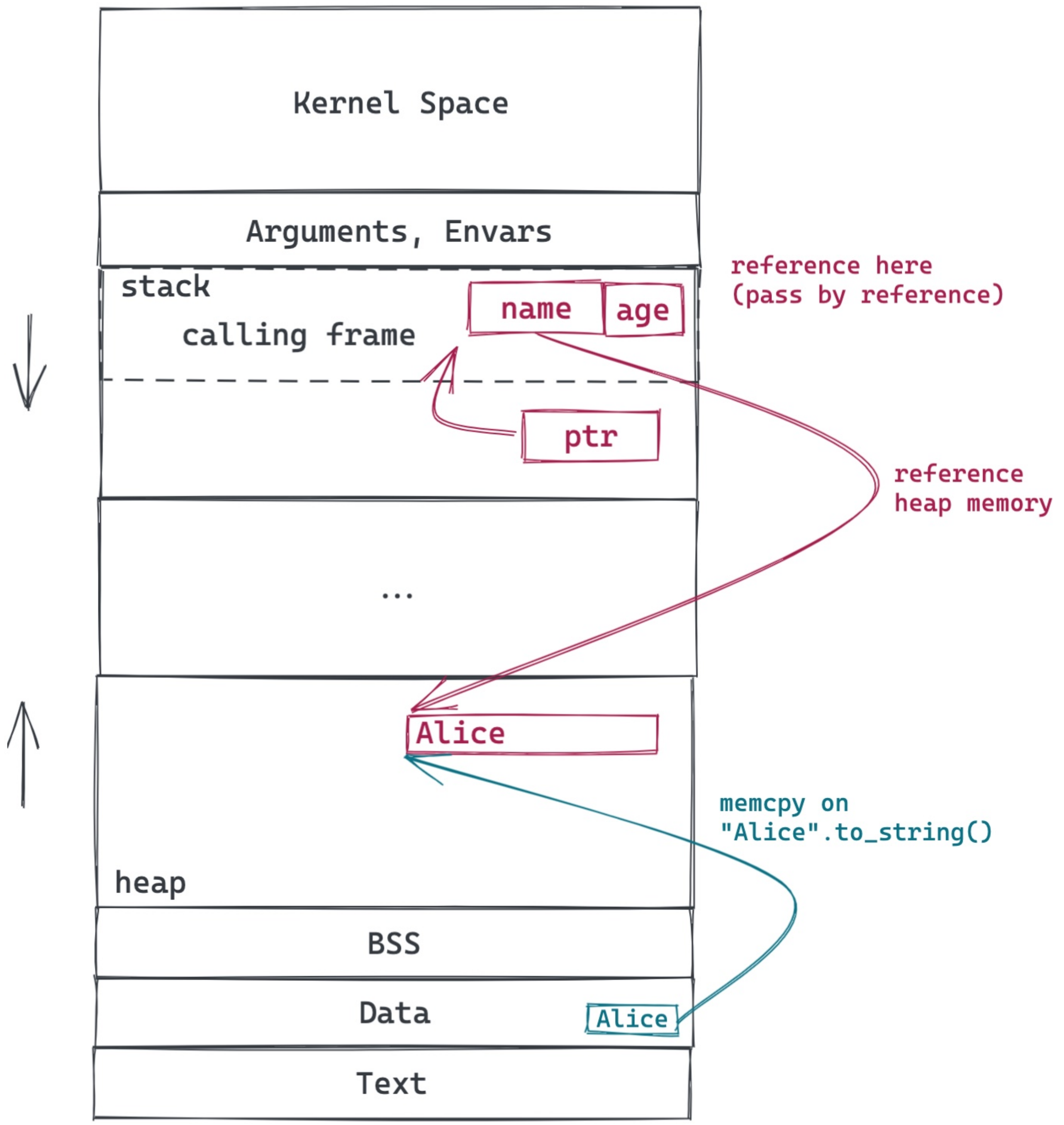


# Immutable Borrow

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);
  ...
}
```



# Benefit of lifetime-contrained 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

```rust
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 resources

# 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

# 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)

# 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

# Typesystem and data structures

- The type system is relatively strong and prevents many classes of bugs.
- The borrow checker and the rules it enforces prevent safety issues at compile time. Some of these violations can be detected by other languages' compilers. However, in many cases sufficient auditing (like {address, memory, thread} sanitizers) is run much less frequently, often only in CI tests, which can be hours or days later.
- Invariants can be encoded and enforced in the type system through features like enums being algebraic data types.
- Variables are immutable by default and must be explicitly annotated as mutable. This forces you to think about where and how data mutation occurs, enabling you to spot issues sooner.
- Option<T> significantly curtails the billion dollar mistake.
- Result<T, E> forces you to reckon about handling errors.

# 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