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

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

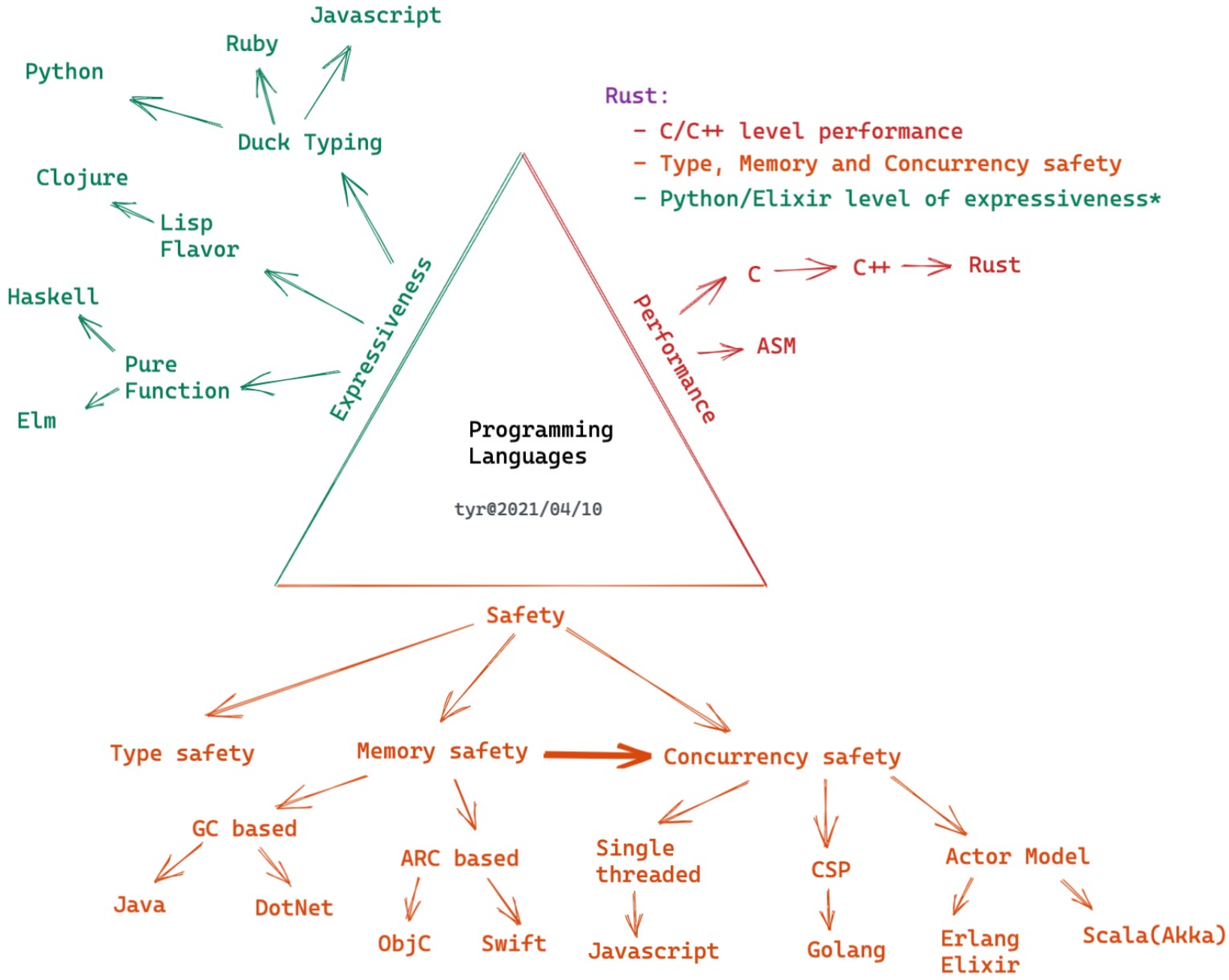
- **Approachability**
- Availability
- Compatibility
- Composability
- Debuggability
- **Expressiveness**
- Extensibility
- Interoerability
- Integrity
- Maintainabiity
- Measurability
- Operability
- Performance
- Portability
- **Productivity**
- Resilliency
- Rigor
- Safety
- Security
- **Simplicity**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

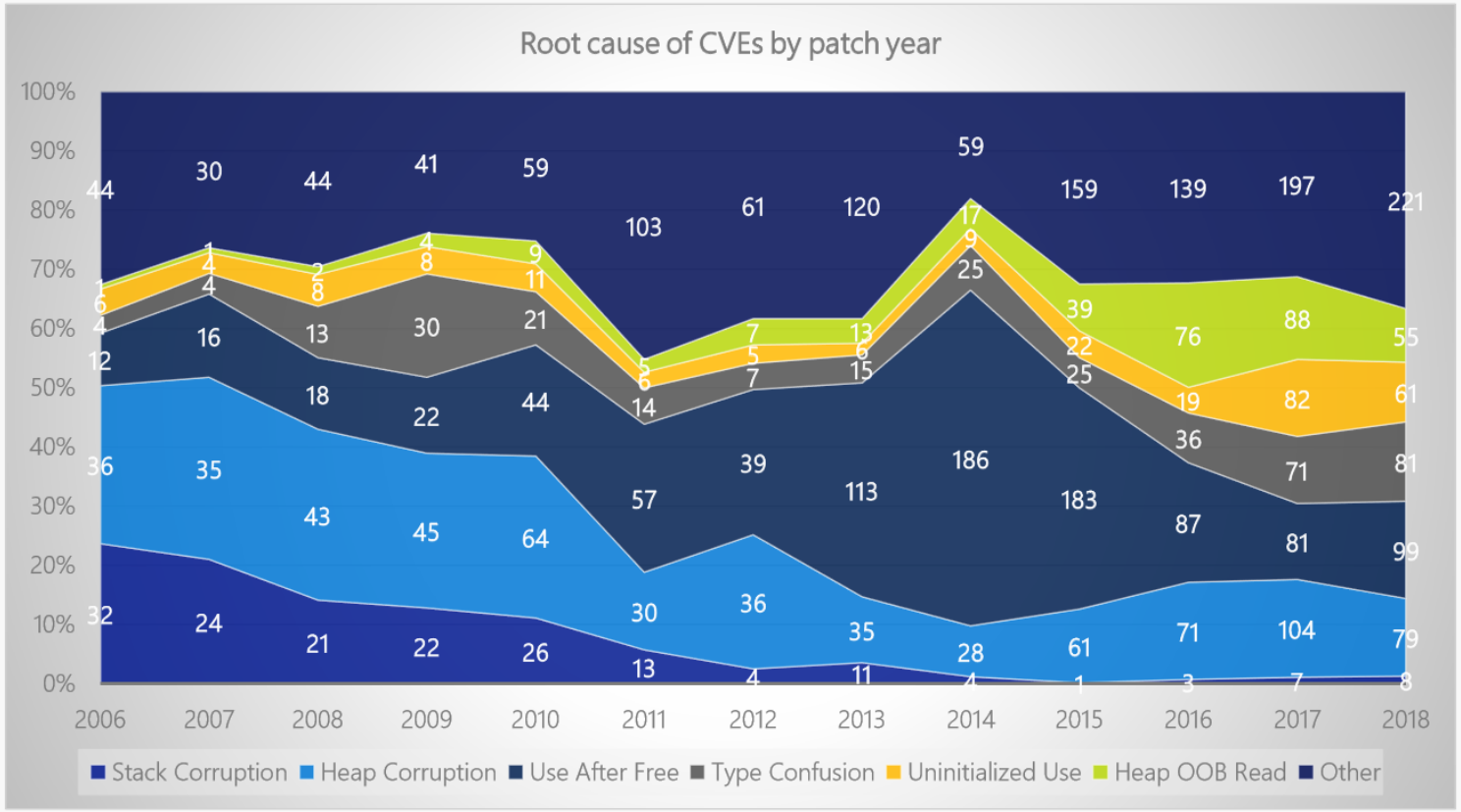
Rust

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


Javascript
Ruby
Python
Duck Typing
Clojure
Lisp
Flavor
Haskell
Pure
Function
Elm
Expressiveness
Rust:
- C/C++ level performance
- Type, Memory and Concurrency safety
- Python/Elixir level of expressiveness*
C
C++
Rust
ASM
Performance
Programming
Languages
tyr@2021/04/10
Safety
Type safety
Memory safety
Concurrency safety
GC based
Java
DotNet
ARC based
ObjC
Swift
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**)

Why 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

How ownership works?

- Question to ask - Passing value: by ref or by value?
- Answers from Rust:
  - A value can only have one owner in a scope
  - When an owned value is out of scope, it is dropped
  - A value can have multiple immutable references
  - A value can have only one mutable reference which is mutual exclsive
  - A reference cannot outlive its owner

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 actually achieves memory safety and conccurency safety?

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);
  });
}
```

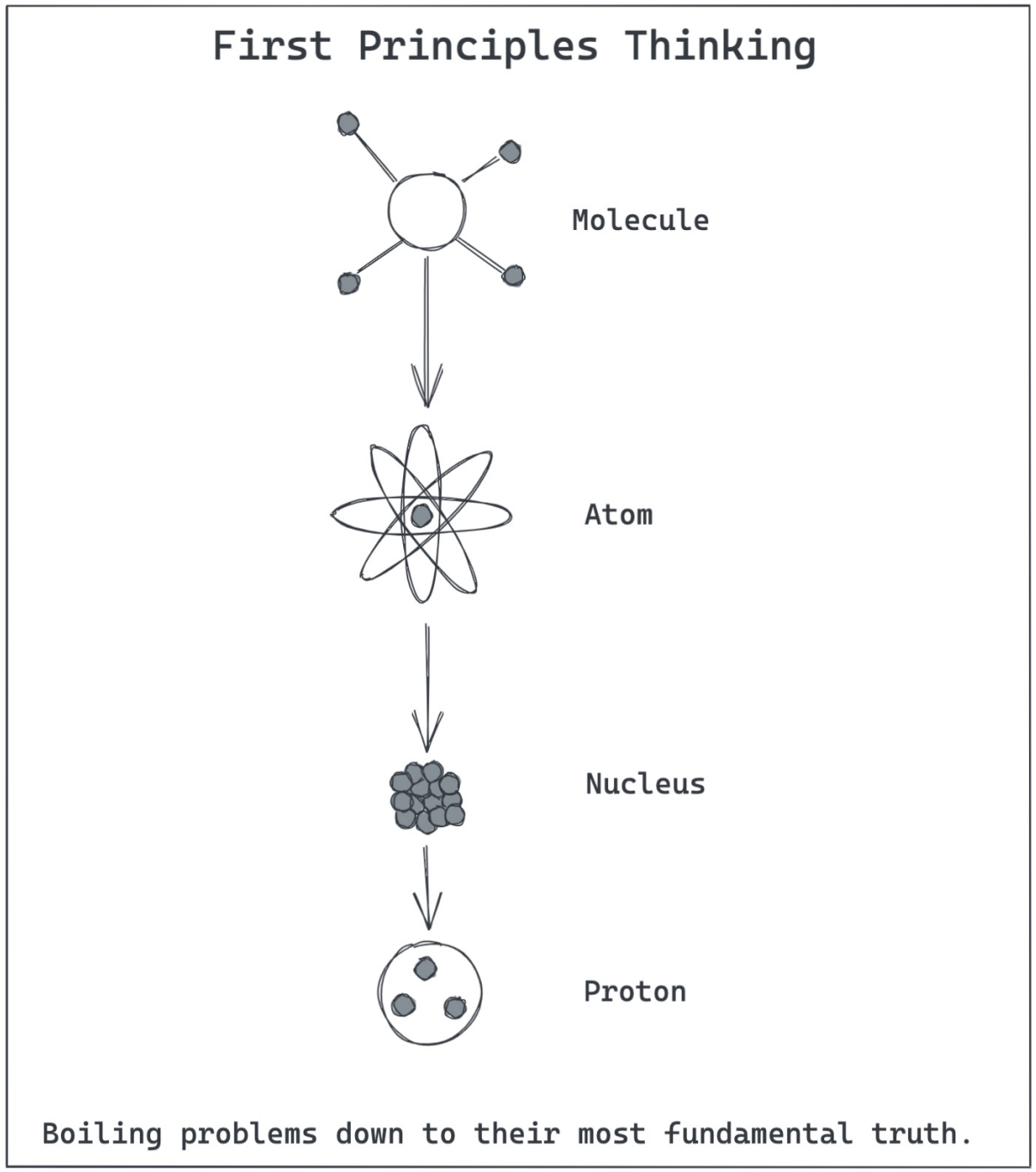
closure may outlive

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

Zero Cost Abstraction

Rust way of solving problems

Let's go to basics about types

- can be used any number of times
  - Other languages: this is how we works
  - Rust: Copy / Clone
- can't be used more than once
  - Other lanugages: ??
  - Rust: move semantics
- must be used at least once
  - Other lanugages: linter will detect that, hopefully
  - `unused_variables`, `unused_assignments`, `unused_must_use`
- must be used exactly once

References

- The pain of real linear types in Rust
- Substructural type system

About memory safety

- C/C++

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

May the Rust be with you