Rust Programming Language

Prepared By Kent Marete

High Level Languages

Years

Issues

1. It’s difficult to write secure code

- It’s common for security exploits to leverage bugs in the way C and C++ programs handle memory,

2. It’s very difficult to write multithreaded code, which is the only way to exploit the abilities of modern machines.

C++

- C with classes

  Rust shares the ambitions Bjarne Stroustrup articulates for C++ in his paper **Abstraction and the C++ machine model**:

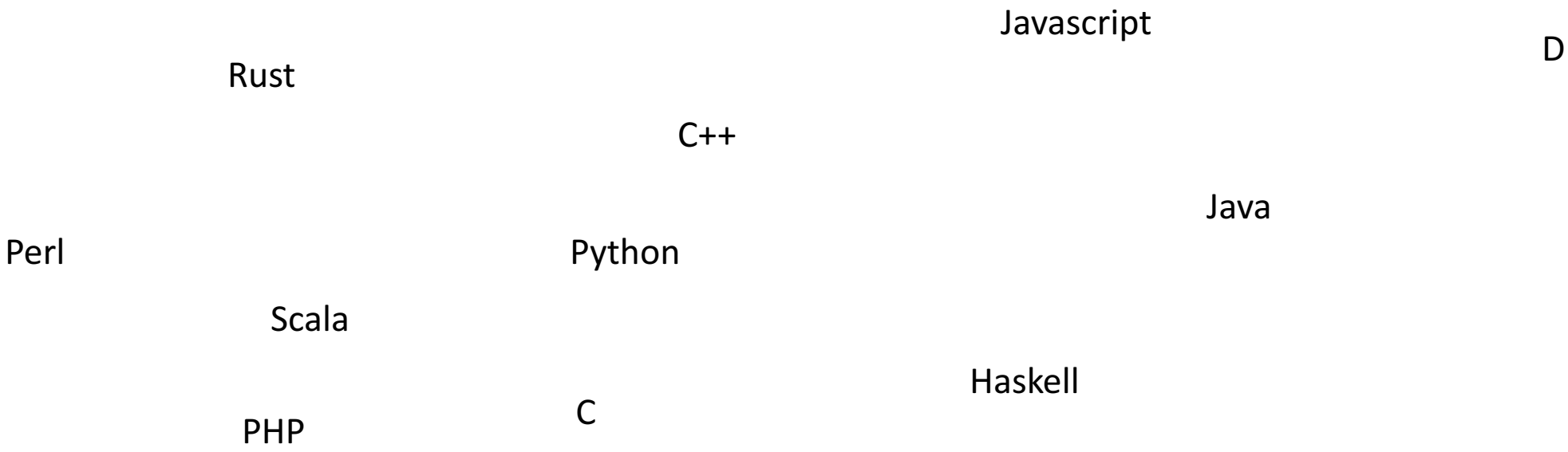
- *What you don't use, you don't pay for*

Content

- What is Rust?
- What rust has to offer?
- Why Rust?
- What is safety?
- What is control?
- What is Concurrency?
- Intro to Data structures – struct, tuples, slices, string, closures
- References…

What is Rust?

- **Rust** is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety. **(safety, concurrency, and speed)**
- It’s a programming language founded by Mozilla research.

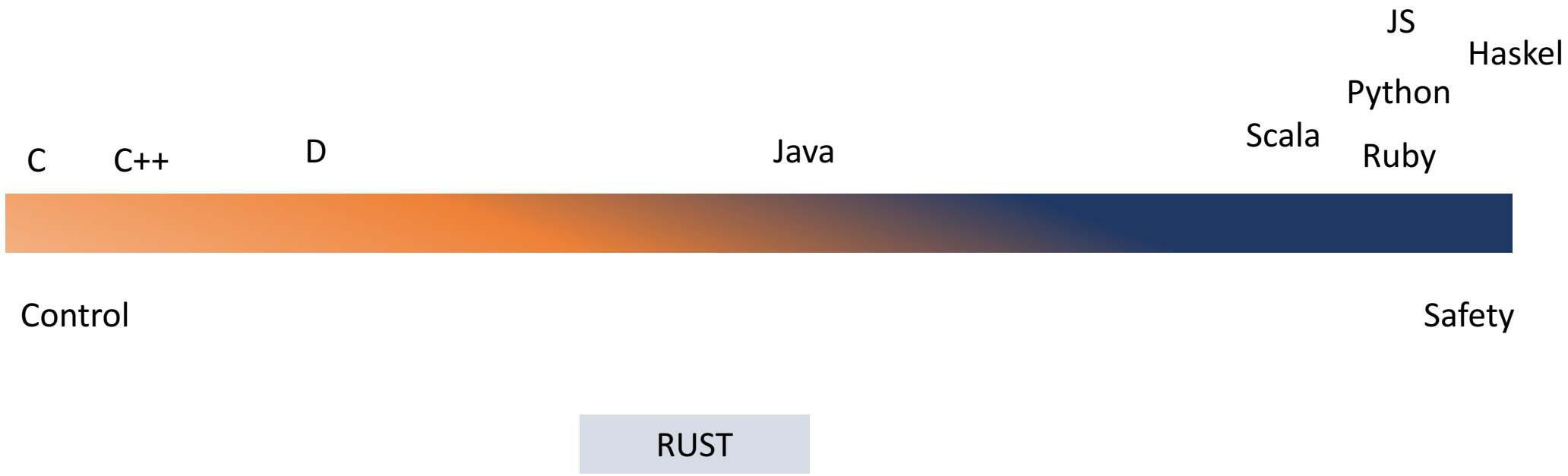


History

- **Mozilla** starts sponsoring Rust in 2009
- Post 1.0.0 (2015-)
- **> 11K** crates - libraries
- **1,852** contributors on Github.
- **Big areas**: game dev, operating systems, web development, block chain

What Rust has to offer

- We can organize these languages in a linear spectrum



Speed?

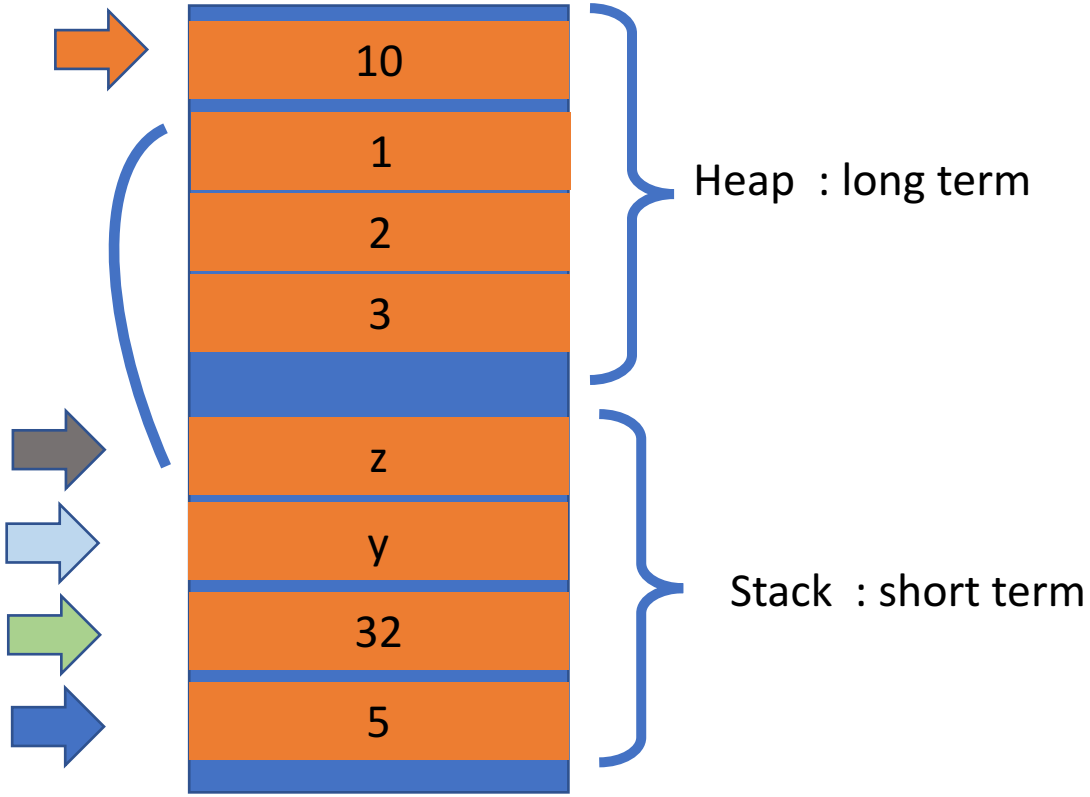
Speed?

- No Garbage Collection - Rust uses the ***Resource Acquisition Is Initialization*** (RAII) technique - object lifetime
- LLVM - is a compiler infrastructure
- Zero Cost Abstractions - *What you don't use, you don't pay for*
- Minimal Runtime – No GC, can compile without stdlib

What is Control?

Rust gives the developer fine control over the use of memory

Control? - Stack and heap



Use box construct
let y = Box::new(10);

println!("y = {}", *y);

let z = vec![1,2,3];

let x = 5; //i32

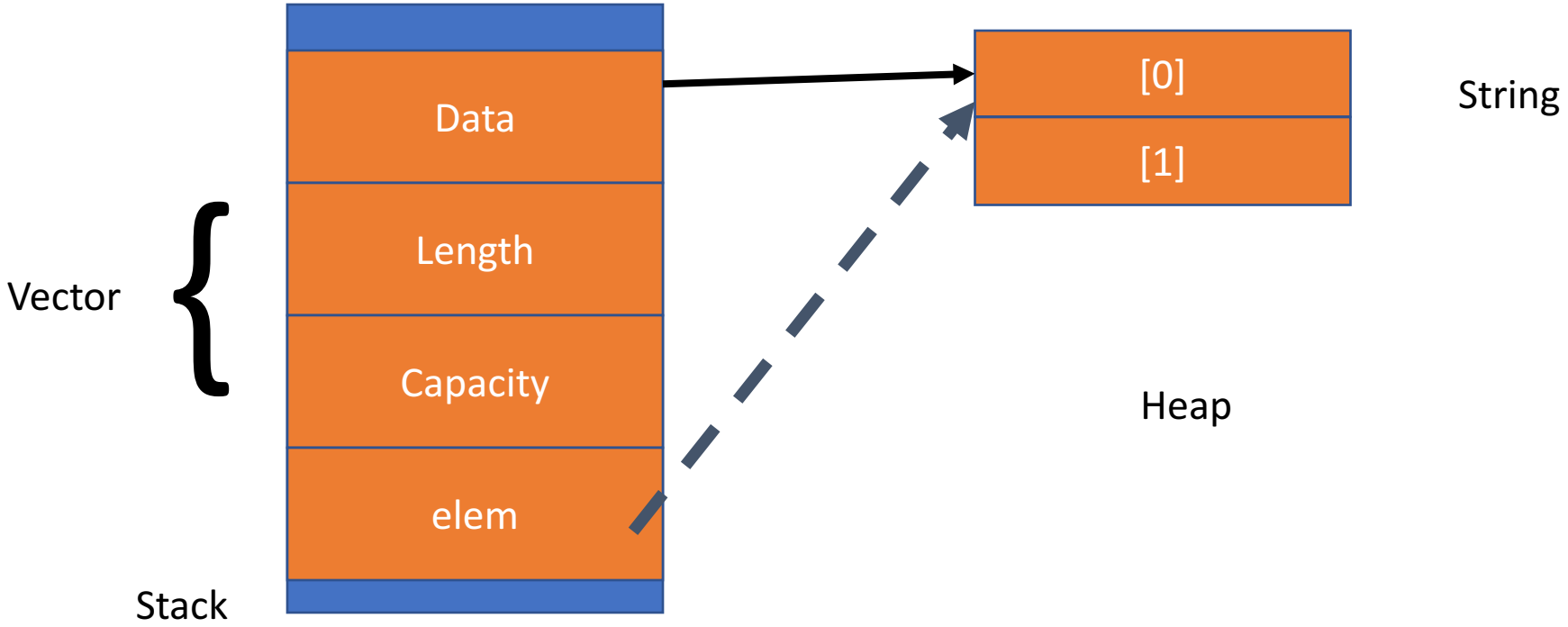
fn inc(x:i32){x+1}

inc(32);

[Playground link]

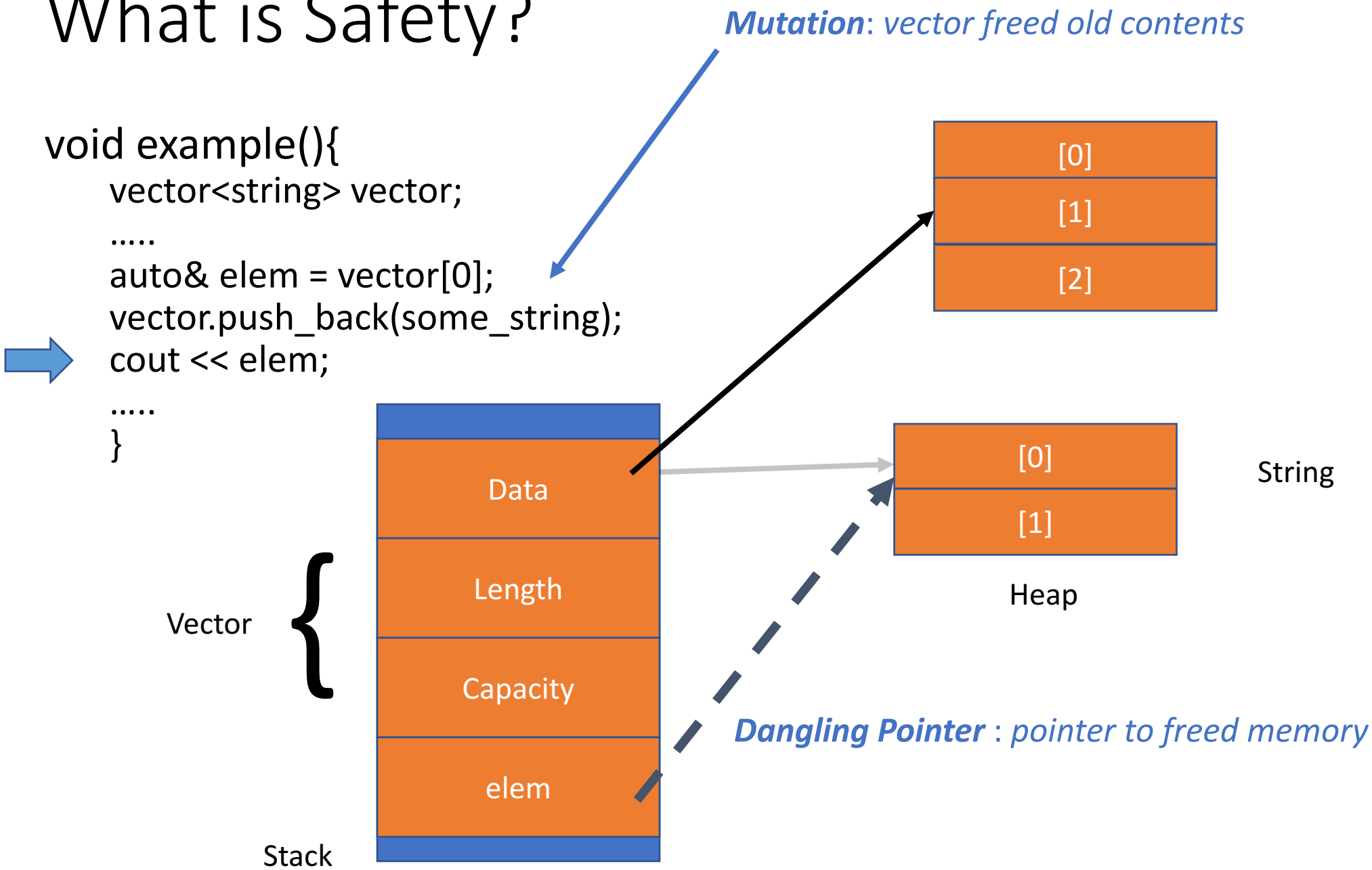
What is Safety?

```
void example(){
    vector<string> vector;
    …..
    auto& elem = vector[0];
    vector.push_back(some_string);
    …..
    }
```



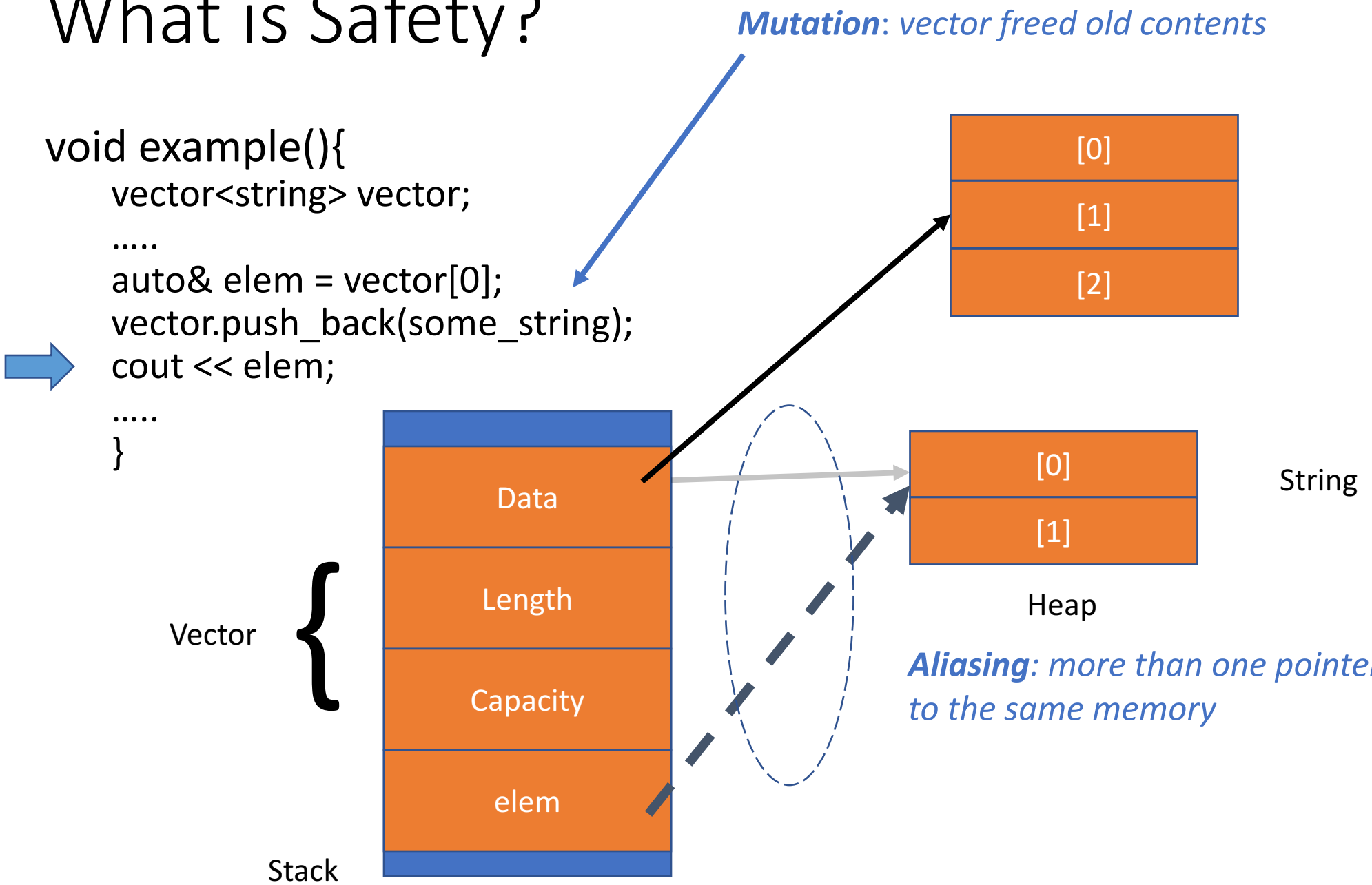
What is Safety?

```
void example(){
    vector<string> vector;
    …..
    auto& elem = vector[0];
    vector.push_back(some_string);
    cout << elem;
    …..
    }
```



What is Safety?

```
void example(){
    vector<string> vector;
    …..
    auto& elem = vector[0];
    vector.push_back(some_string);
    cout << elem;
    …..
    }
```

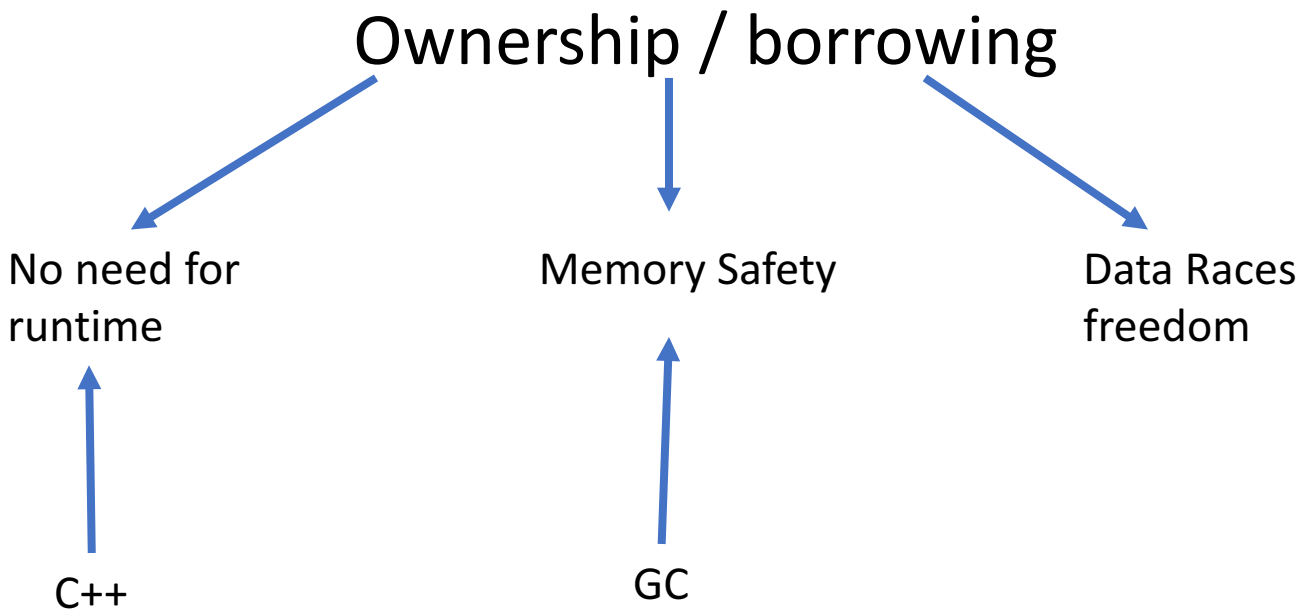


Garbage Collection?

Downside:

- No Low level control
- GC pauses -- suspension time
- requires runtime

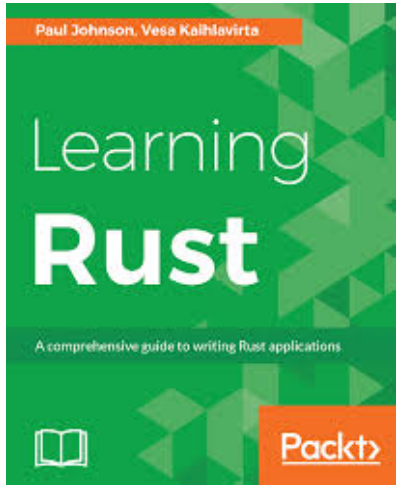
Rust Solution



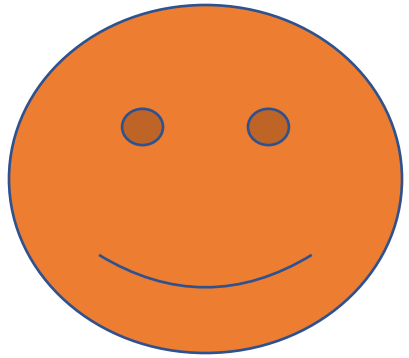
Ownership

~~Aliasing~~ + Mutation

Owner

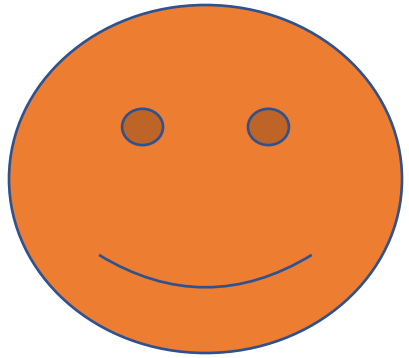


Give the book



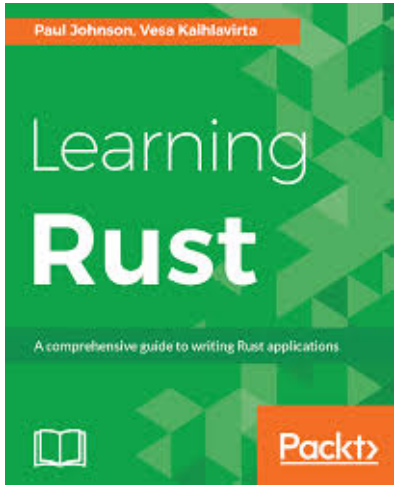
Ownership

Owner



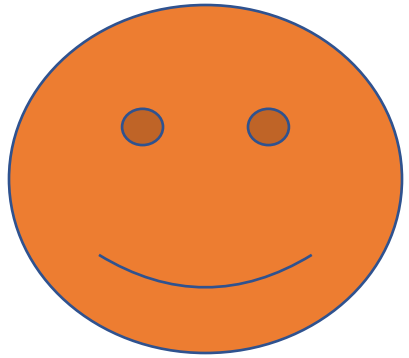
New Owner

Take the book

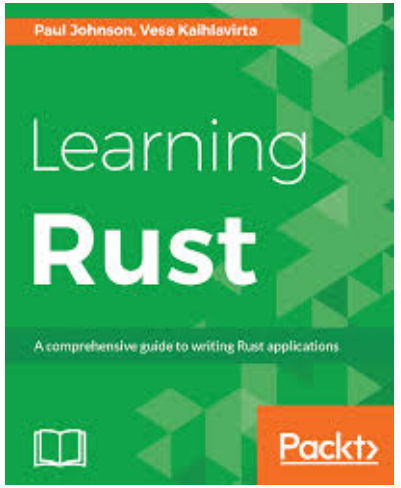


Ownership

The owner decides to go away



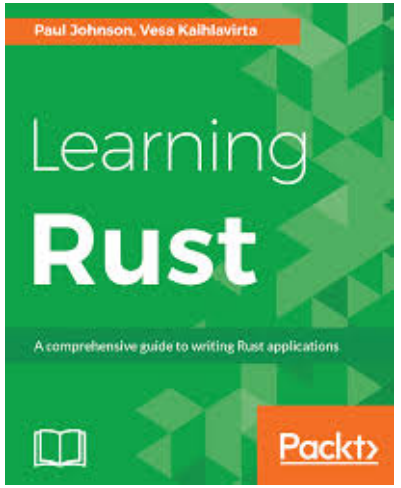
Take the book



Ownership

The new owner goes away

Destroy the book



Compiler enforces ownership

```
fn give(){
    let mut vec = Vec::new();
    vec.push(1);
    vec.push(2);
    take(vec);
    vec.push(3);
}
```

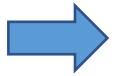
```
fn take(vec: Vec<i32>) {
    println!("{:?}", vec);
}
```

error[E0382]: use of moved value: `vec`

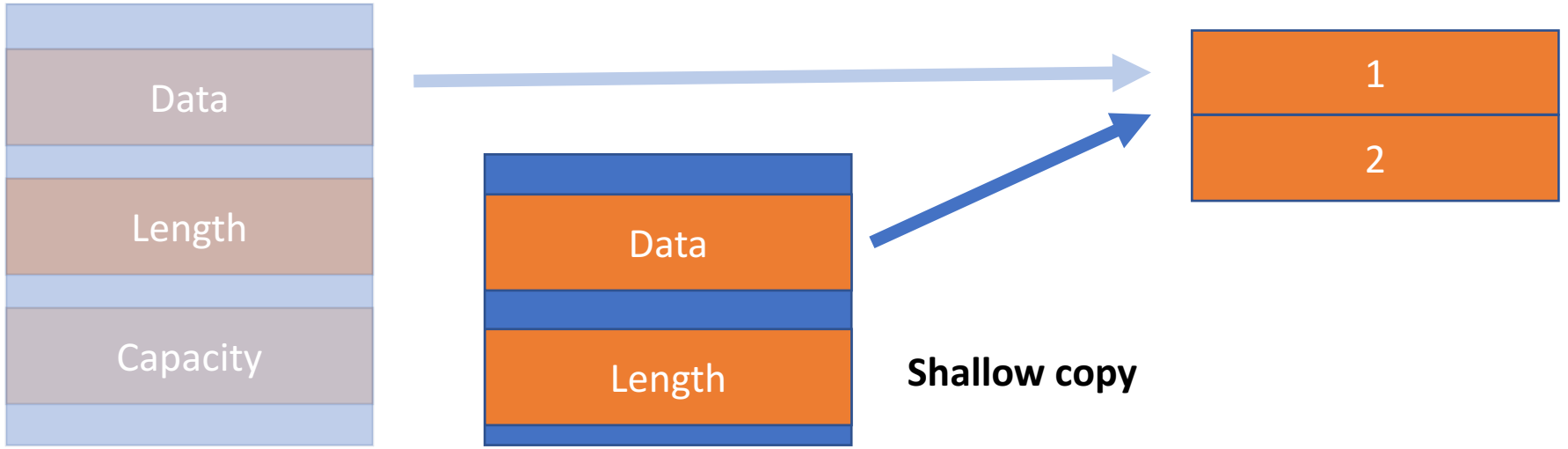
Playground link

Compiler enforces ownership

```
fn give(){
        let mut vec = Vec::new();
        vec.push(1);
        vec.push(2);
        take(vec);
}
```



```
fn take(vec: vec<i32>) {
        println!("{:?}", vec);
}
```



Rules of Ownership

- Each value has its ***owner***.

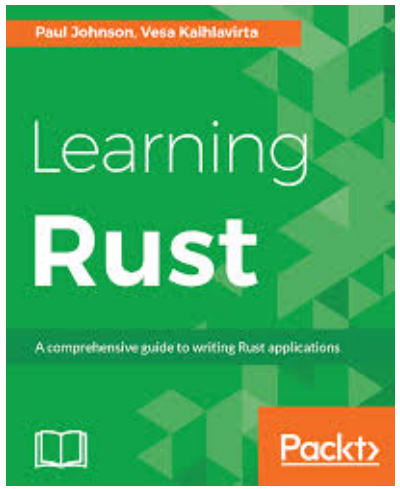
- There can only be one **owner** at a time.

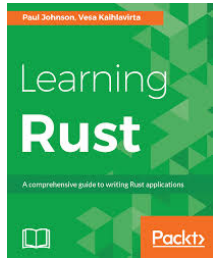
- When the owner goes out of scope, the value will be dropped.

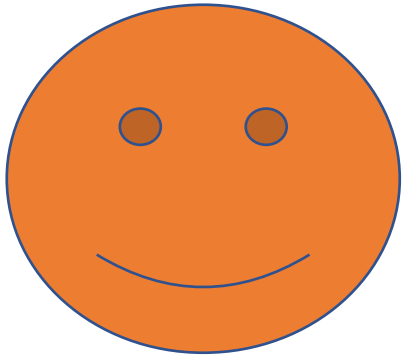
Borrowing (Shared Borrowing(**&T**))

Aliasing + ~~Mutation~~

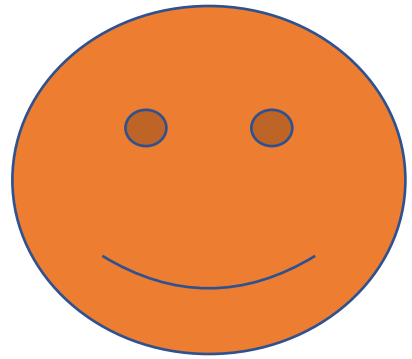
Owner





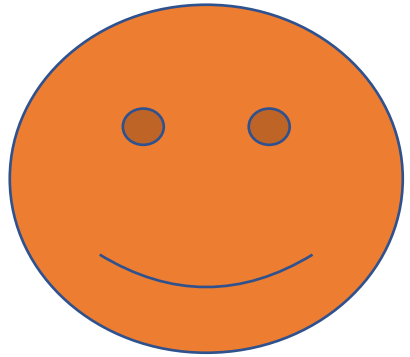


Give the book

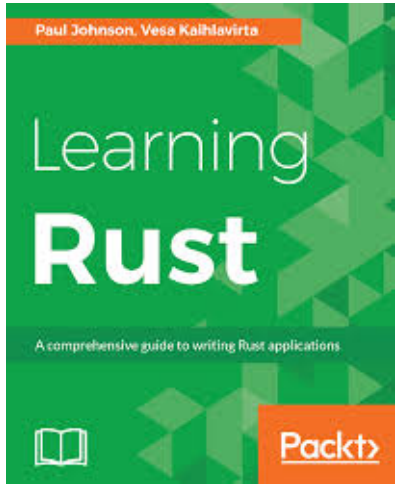


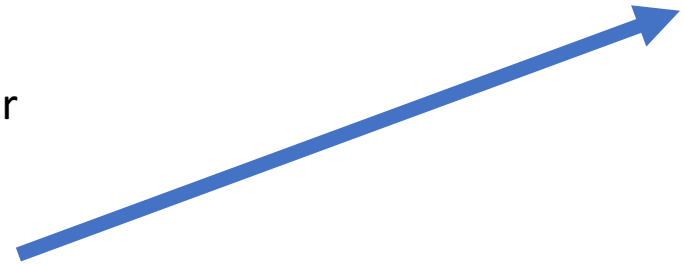
Borrowing (Mutable Borrowing(**&mut T**))

~~Aliasing~~ + Mutation

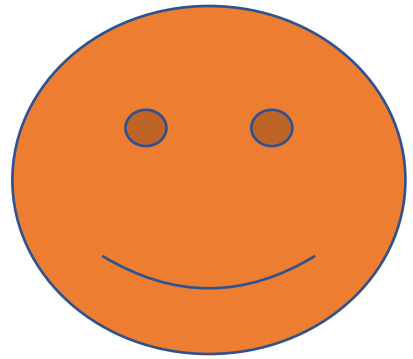


Owner



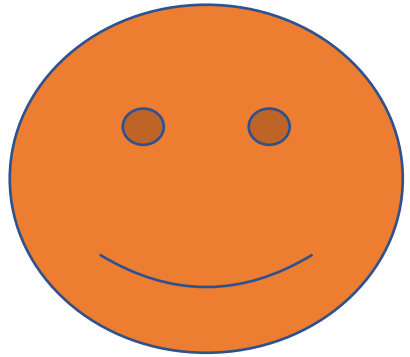


Give the book

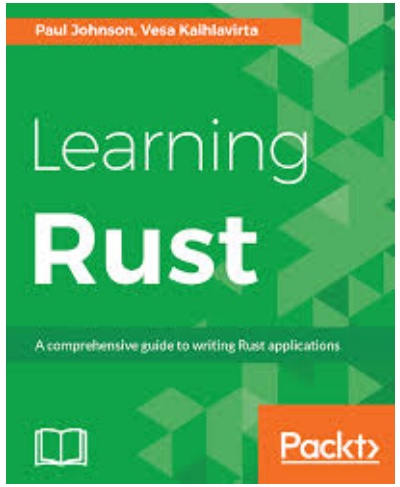


Borrowing (Mutable Borrowing(**&mut** T))

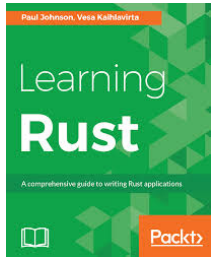
~~Aliasing~~ + Mutation

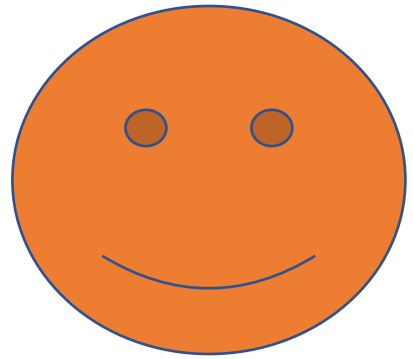


Owner



Give the book





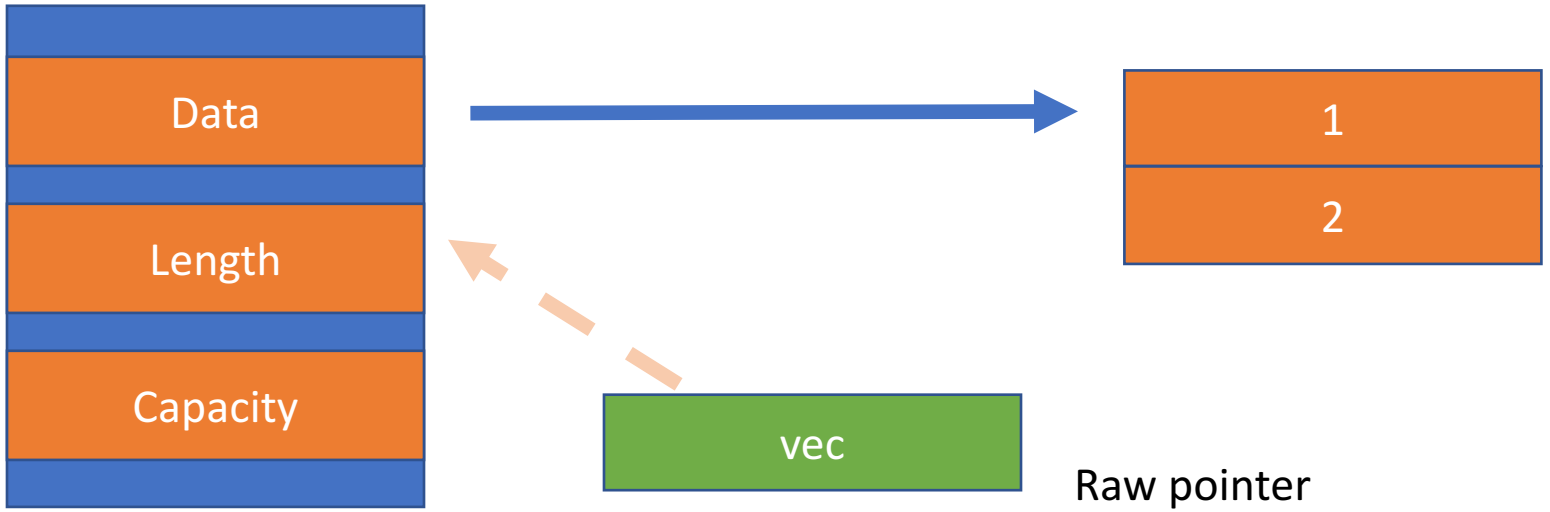
Shared borrow (&T)

Shared reference to &vec<i32>

```
fn  lender(){
        let  mut  vec  =  Vec::new();
        vec.push(1);
        vec.push(2);
        user(&vec);}
```

```
fn user(vec: &vec<i32>) {
        println!("{:?}",  vec);
}
```

Loan out the vec



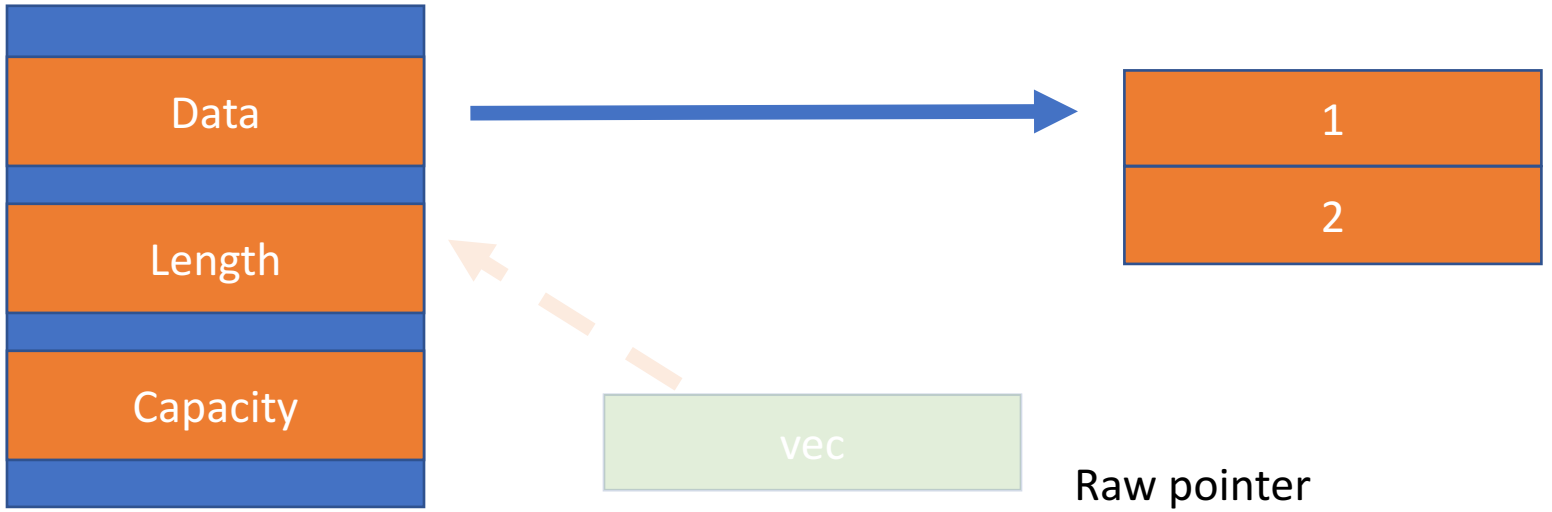
Playground link

Shared borrow

```
fn  lender(){
        let  mut  vec  =  Vec::new();
        vec.push(1);
        vec.push(2);
        user(&vec);}
```

```
fn user(vec: &vec<i32>) {
        println!("{:?}",  vec);
}
```

End forget about the vec



[Playground link]

Shared reference are immutable

```
fn user(vec: &vec<i32>) {
        vec.push(1);
        vec.push(2);
}
```

Aliasing + ~~Mutation~~

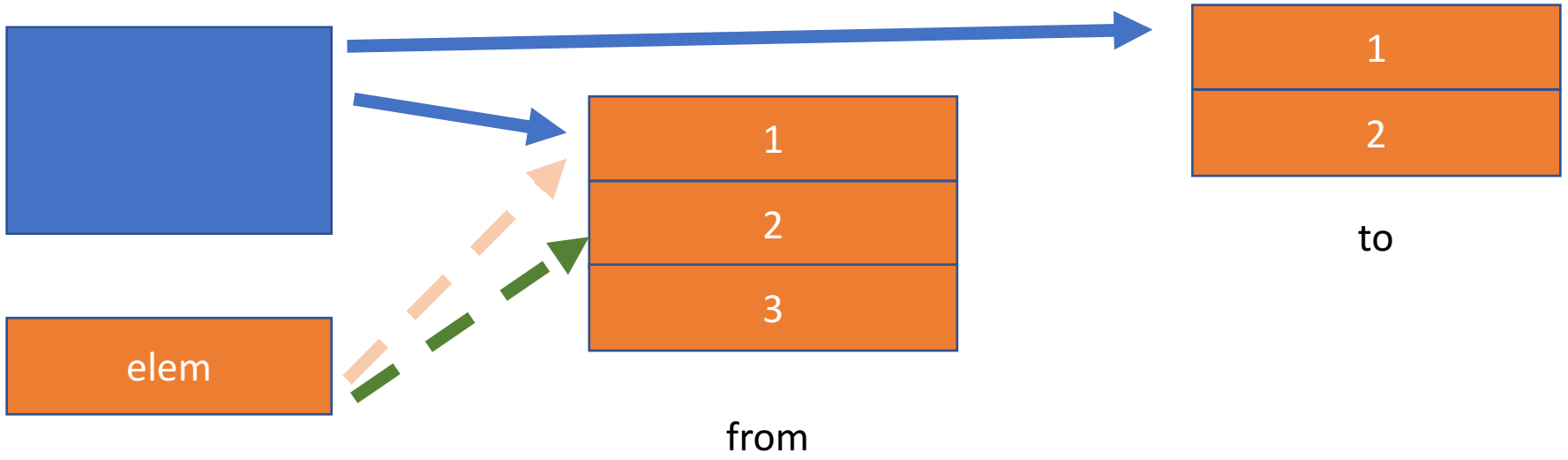
Error : cannot mutate shared references

Mutable references --- Iteration

```
fn send_message(from: &vec<i32>, to: &mut vec<i32>) {
    for elem in from.iter(){
        to.push(*elem);
    }
}
```

Shared references

Mutable references



Lessons

- Mutable reference is the only way to access the memory it points at
- Cannot have both shared and mutable references at the same time.

example

```
fn example(){
    let mut vec = Vec::new();

    for i in 0..vec.len(){
        let elem: &i32 = &vec[i];
        vec.push(1);
        println!("{:?}",elem);
    }

    vec.push(1);


    println!("{:?}", vec);
}
```

FINE TO DO : Loan expired

Playground Link

Concurrency?

Concurrency?

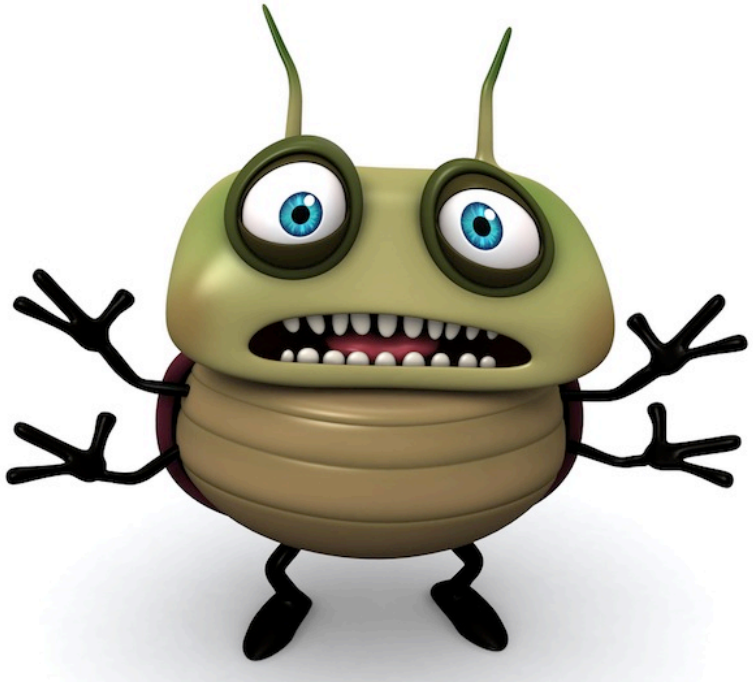
- Handling concurrent programming safely and efficiently
- Concurrency addresses how to create threads to run multiple pieces of code at the same time

Concurrency?

- Concurrency addresses how to create threads to run multiple pieces of code at the same time

Concurrency?

- Memory safety bugs and concurrency bugs



ownership

Rust's compiler checks statically for you

Who is using Rust?

Friend of Rust

https://www.rust-lang.org/en-US/friends.html

End