Lesson 2 Outline

- Recap Lesson 1
- Ownership
- Borrowing
- str & String
- Practice & Examples

https://github.com/RustRome/corso-rust

Recap Lesson 1

Variables

```
let a = 5;
let b = "Rust Evangelism Strike Force";
let b: &str = "Rust Evangelism Strike Force";
let a = 34usize;
let c = 12_usize;
let c = 100_000;
```

With **mut** keyword for mutability

Primitive Data Types

Ints: **i8, i16, i32, i64, u8, u16, u32, u64**

Floats: **f32, f64**

And: **bool, char, &str, array, tuple**

Functions

```
fn add(a: i32, b: i32) → i32 {
  a + b
  // or
  // return a + b;
}
```

C vs Rust

C

```c
void main(){
    const int n = 4;
    //n++;
    printf("%d\n",n);
    int* p_n = &n;
    (*p_n)++;
    printf("%d\n",n);
}
```

Rust const

```
fn main() {

    const N : i32 = 1;
    let v = &mut N;
    println!("{}", v);
    *v = *v+1;
    println!("{}", N);
    println!("{}", v);

}
```

Rust static

```rust
fn main() {

    static N : i32 = 1;
    let v = &mut N;
    println!("{}", v);
    *v = *v+1;
    println!("{}", N);
    println!("{}", v);

}
```

```
error[E0596]: cannot borrow immutable static item `N` as mutable
 --> src/main.rs:7:13
  |
7 |     let v = &mut N;
  |             ^^^^^^ cannot borrow as mutable
```

Rust static unsafe

```
fn main() {

    static mut N : i32 = 1;

    unsafe {
        N +=1;
        println!("{}", N);
    }

}
```

Key Concepts

Key concepts

Rust is born with the aim to balance **control** and **security**.

That is, in other words:

operate at low level with high-level constructs.

What safety means?

Problem with safety happens when we have a resource that **at the same time**:

- has **alias**: more references to the resource
- is **mutable**: someone can modify the resource

That is (almost) the definition of **data race.**

What safety means?

Problem with safety happens when we have a resource that **at the same time**:

- has **alias**: more references to the resource
- is **mutable**: someone can modify the resource

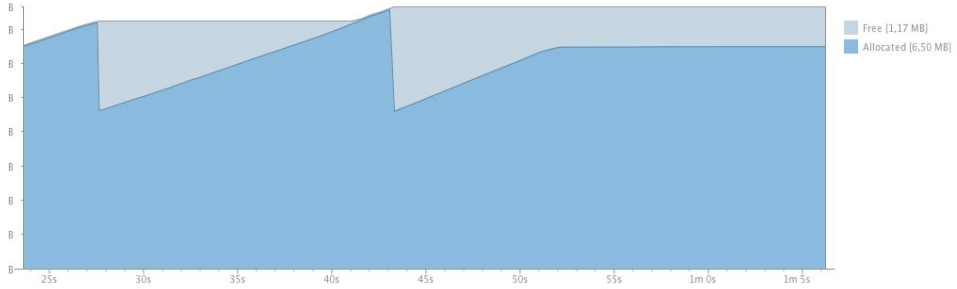
That is (almost) the definition of **data race.**

alias + mutable = 💀

What about the garbage collector?

With the garbage collector:

- we lose control
- requires a **runtime**!

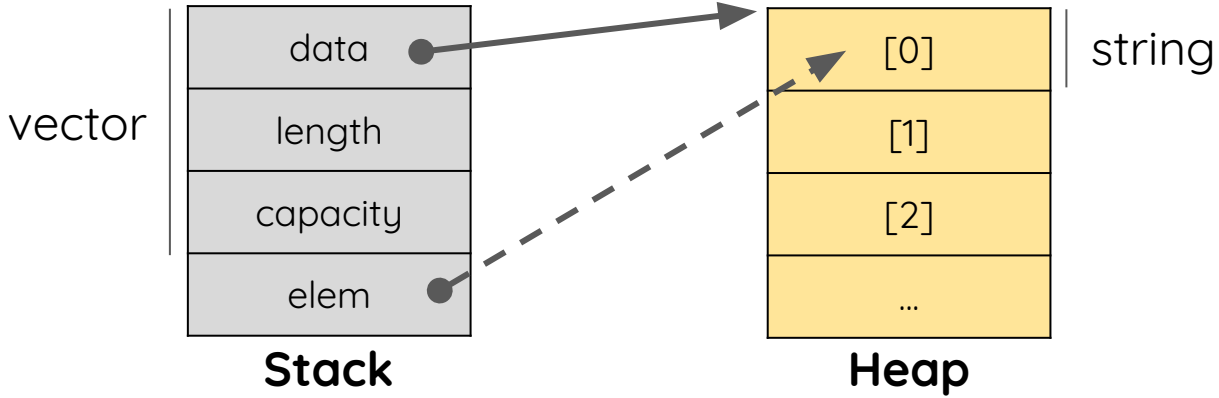


Anyway, it is **insufficient** to prevent data race or iterator invalidation.

What control means?

C++

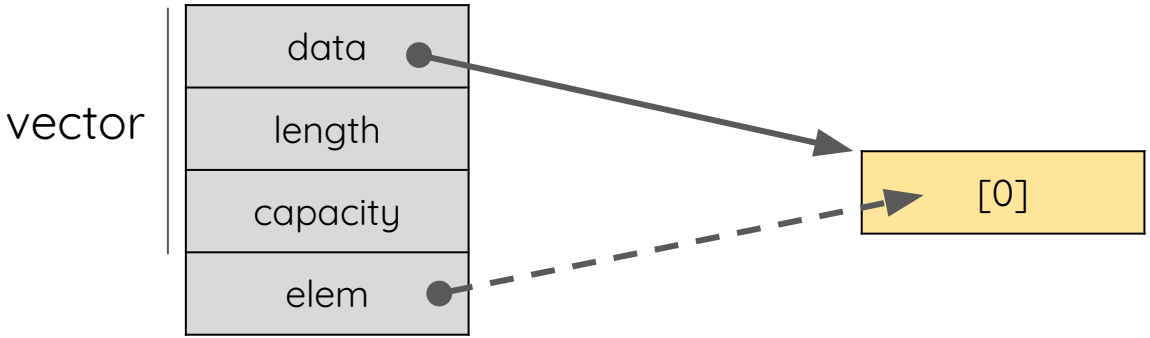
```
void example() {
  vector<string> vector;
  …
  auto& elem = vector[0];
}
```



What safety means?

C++

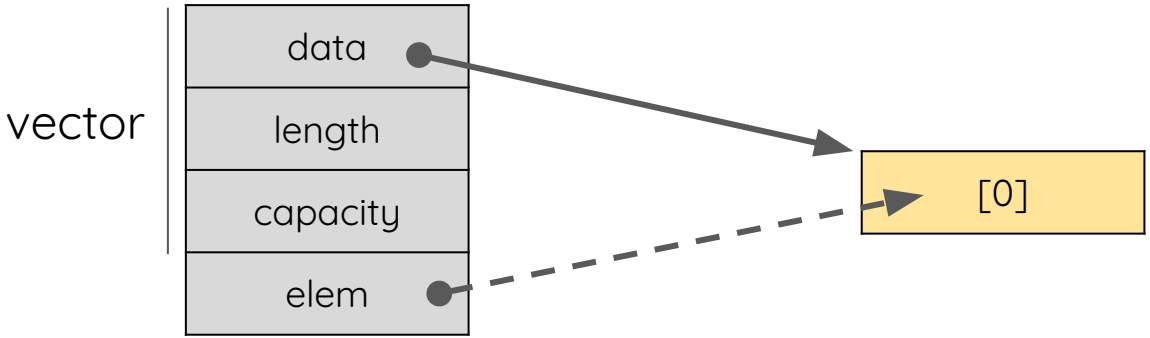
```
void example() {
  vector<string> vector;
  …
⇨ auto& elem = vector[0];
  vector.push_back(some_string);
  cout << elem;
}
```



What safety means?

C++

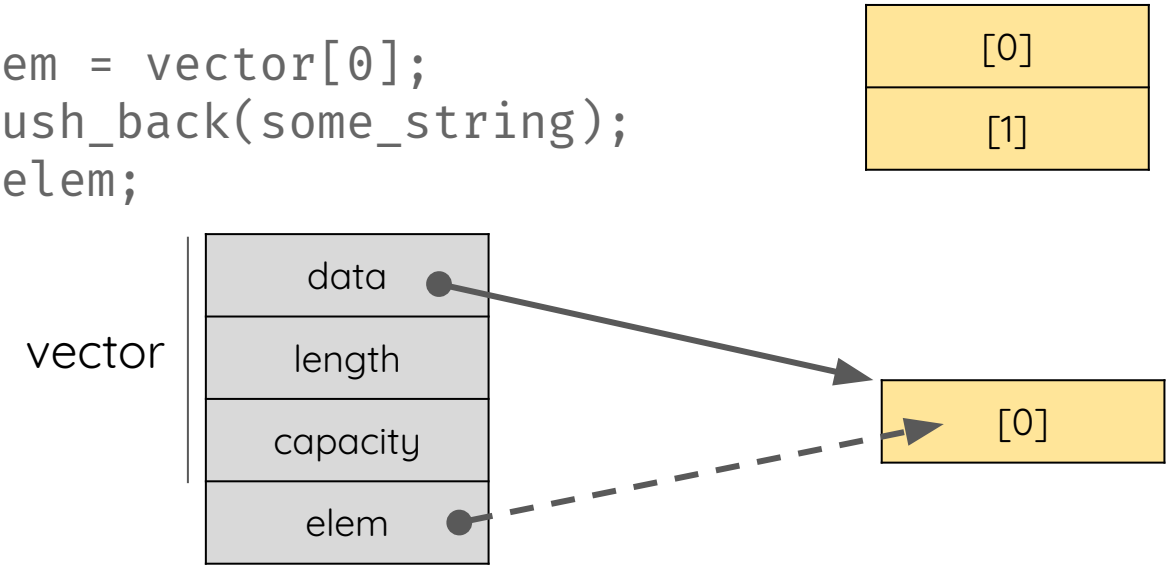
```
void example() {
  vector<string> vector;
  …
  auto& elem = vector[0];
⇨ vector.push_back(some_string);
  cout << elem;
}
```



What safety means?

C++

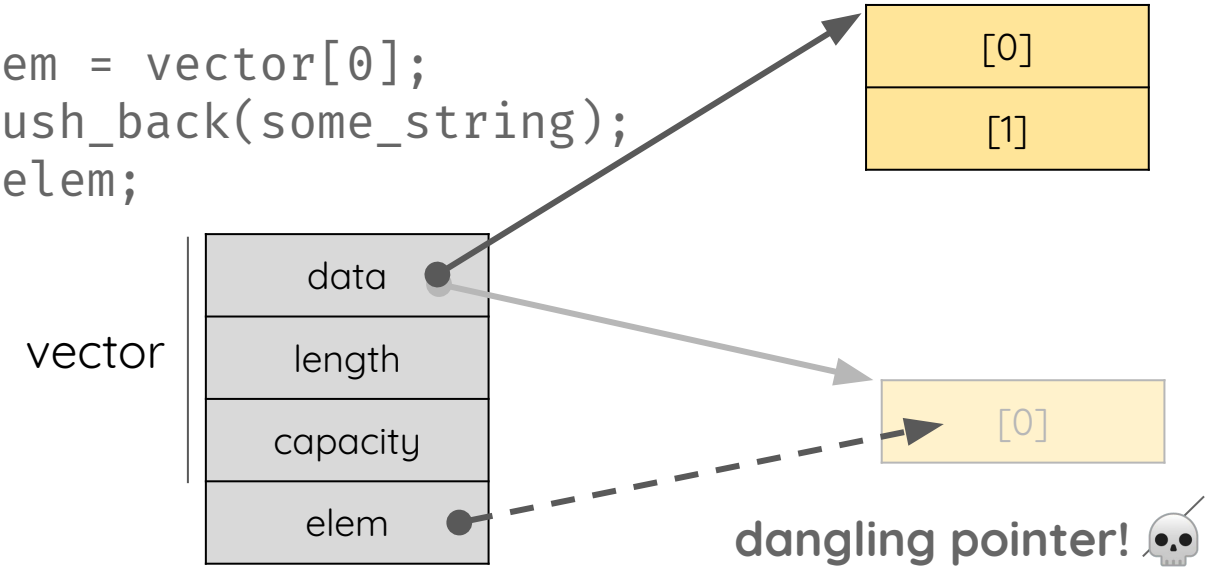
```
void example() {
  vector<string> vector;
  …
  auto& elem = vector[0];
⇨ vector.push_back(some_string);
  cout << elem;
}
```



What safety means?

C++

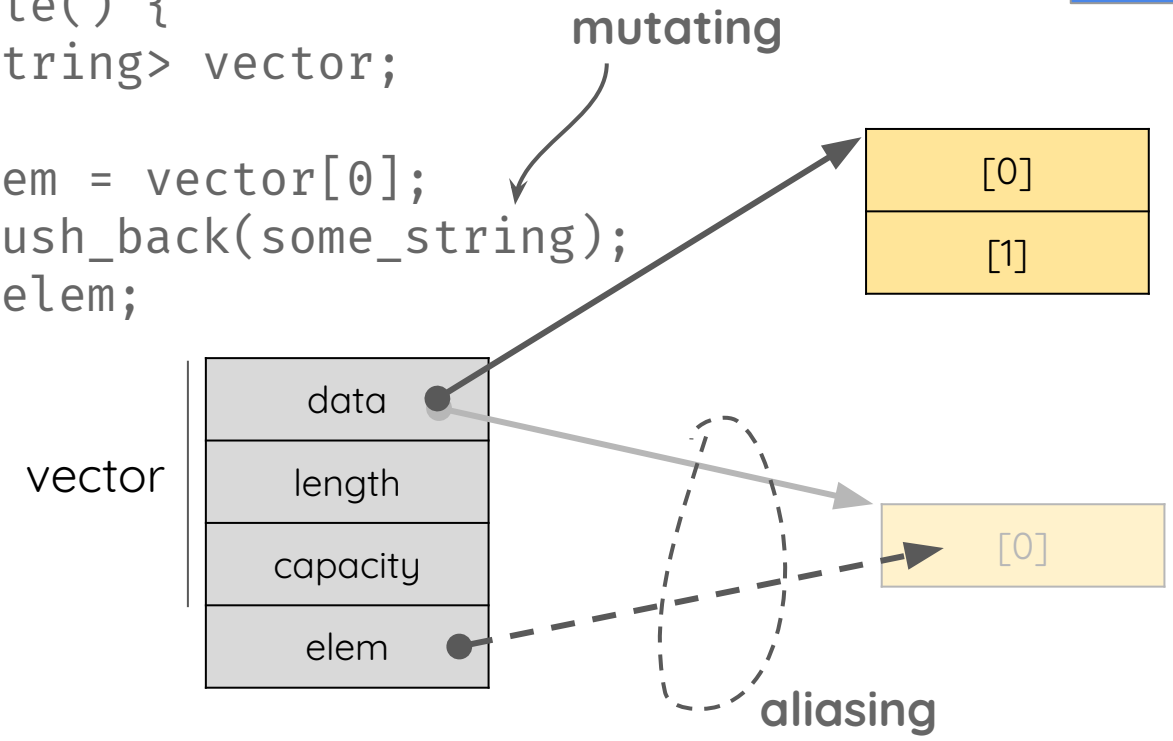
```
void example() {
  vector<string> vector;
  …
  auto& elem = vector[0];
⇨ vector.push_back(some_string);
  cout << elem;
}
```



What safety means?

C++

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



The Rust Way

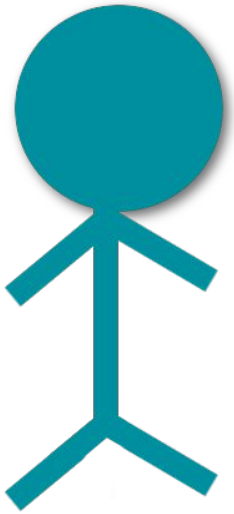
Rust solution to achieve both control and safety is to push as much as possible checks at compile time.

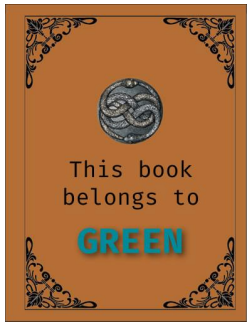
This is achieved mainly through the concepts of

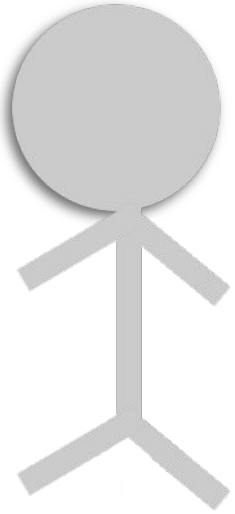
- **Ownership**
- **Borrowing**
- **Lifetimes**

Ownership

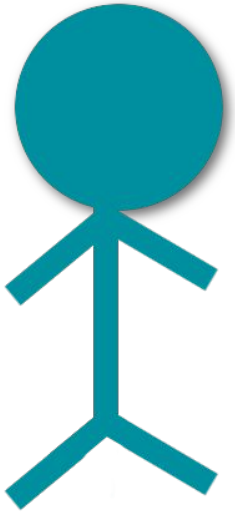
Ownership

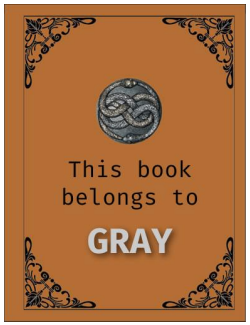


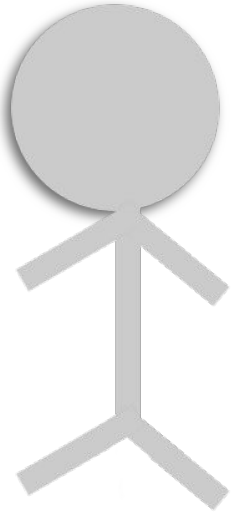




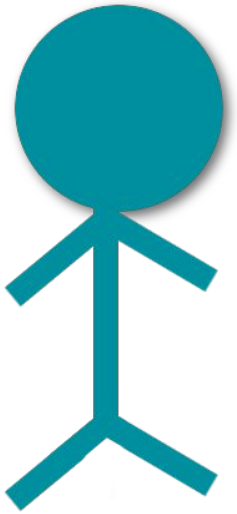
Ownership







Ownership



Green **doesn't own** the book anymore and **cannot use it**

Ownership

Each value has a variable called **owner**, which

can be only **one at time.**

The value is **dropped** when the owner goes out of scope.

Ownership

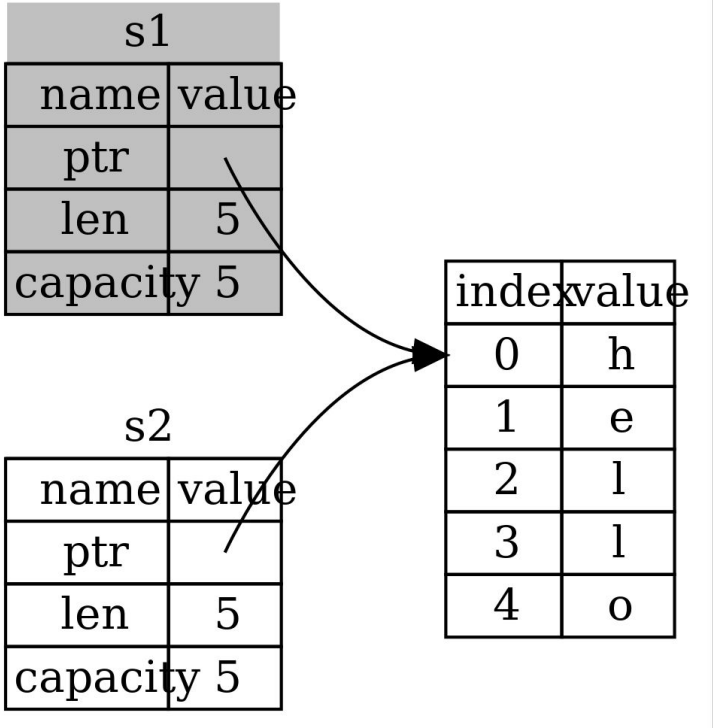
```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

Ownership (Error)

```
error[E0382]: use of moved value: `s1`
 ⟶ src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value used here after move
  |
  = note: move occurs because `s1` has type `std::string::String`, which does
  not implement the `Copy` trait
```

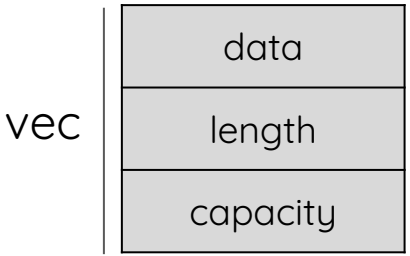
Ownership (Memory)



Ownership

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

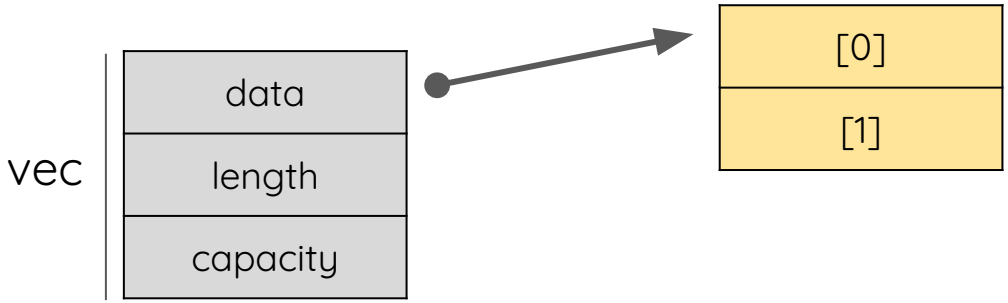
```
fn take(vec: Vec<i32>) {
    //…
}
```



Ownership

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

```
fn take(vec: Vec<i32>) {
    //…
}
```

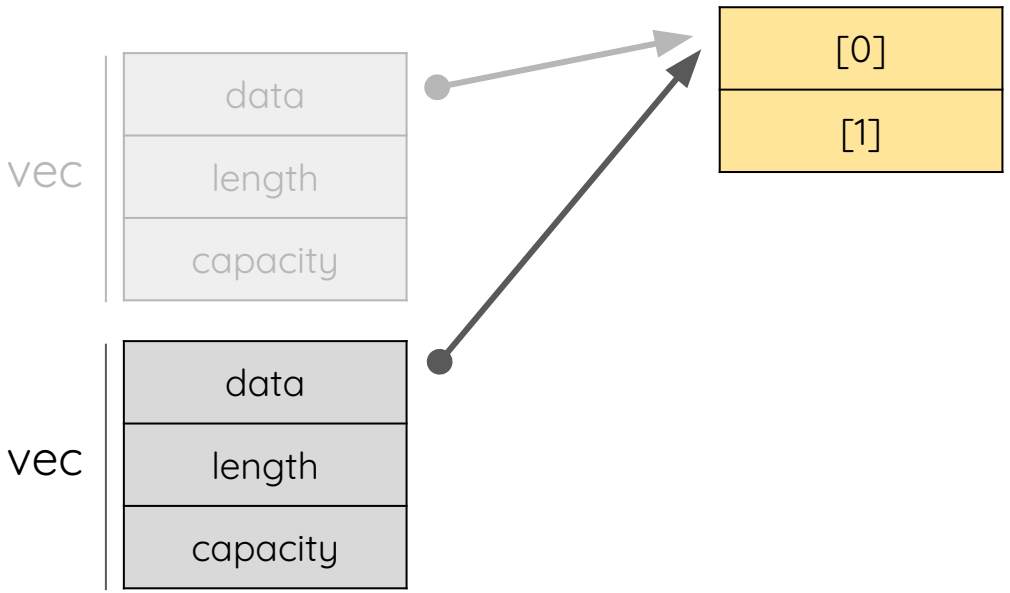


Ownership

take ownership

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

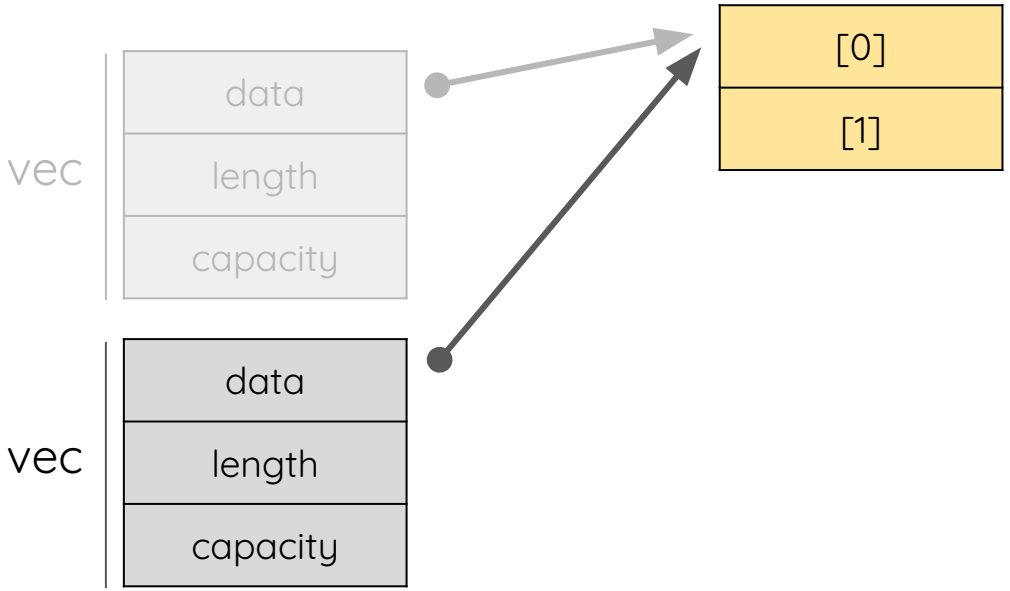
```
fn take(vec: Vec<i32>) {
⇨   //…
}
```



Ownership

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

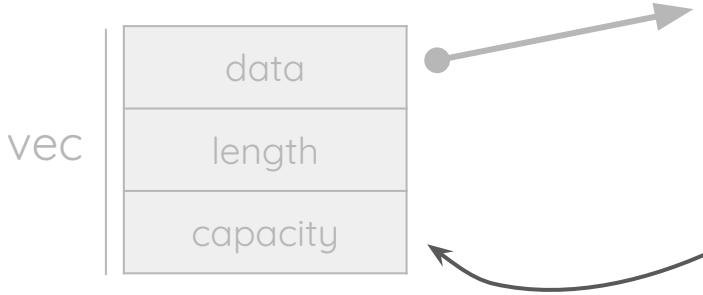
```
fn take(vec: Vec<i32>) {
    //…
}
```



Ownership

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

```
fn take(vec: Vec<i32>) {
    //…
}
```



cannot be used because data is no longer available 📗

Ownership - Error

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

```
fn take(vec: Vec<i32>) {
    //…
}
```

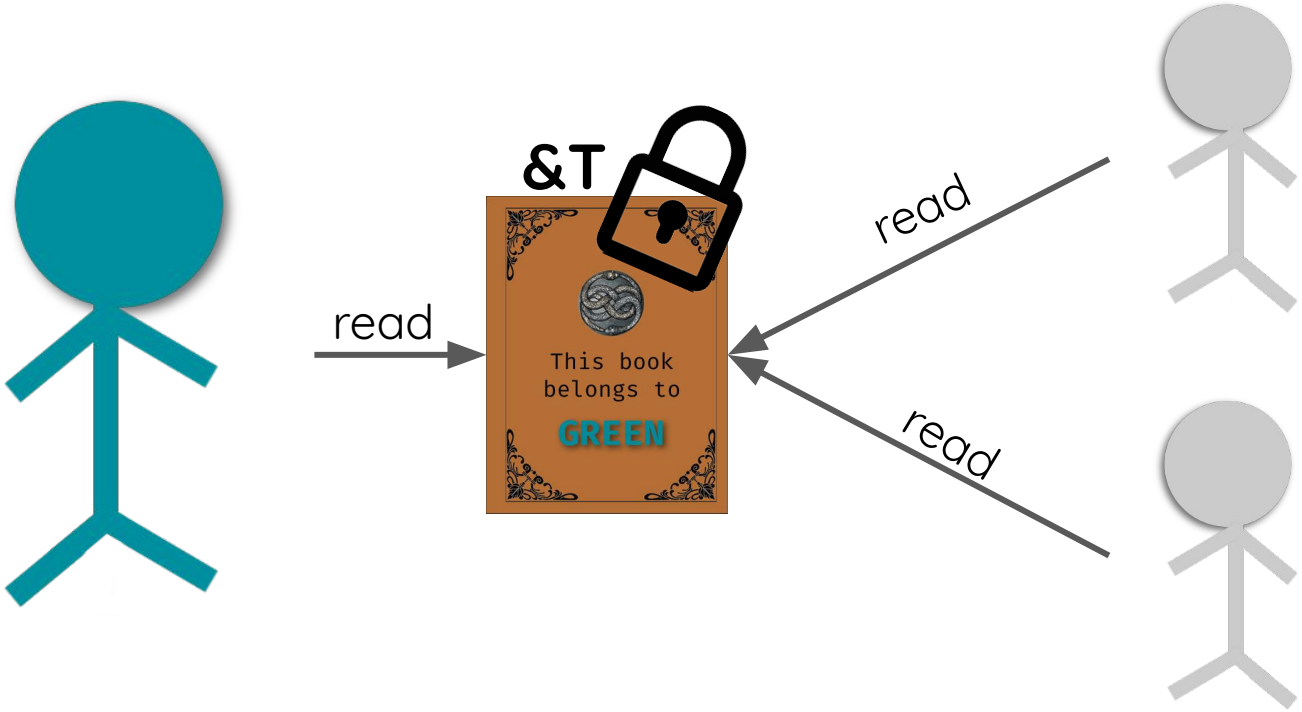
```
error[E0382]: use of moved value: `vec`
  --> src/main.rs:6:5
   |
5  |     take(vec);
   |          --- value moved here
6  |     vec.push(3);
   |     ^^^ value used here after move
   |
   = note: move occurs because `vec` has type `std::vec::Vec<i32>`, which does not implement the `Copy` trait
```

Borrowing

Borrowing with **&T**

one or more references to a resource

Borrowing with **&T**



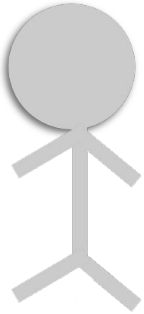
Borrowing with **&T**

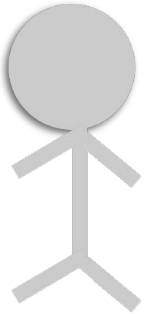
Green can use **its** book again

Borrowing with **&mut T**

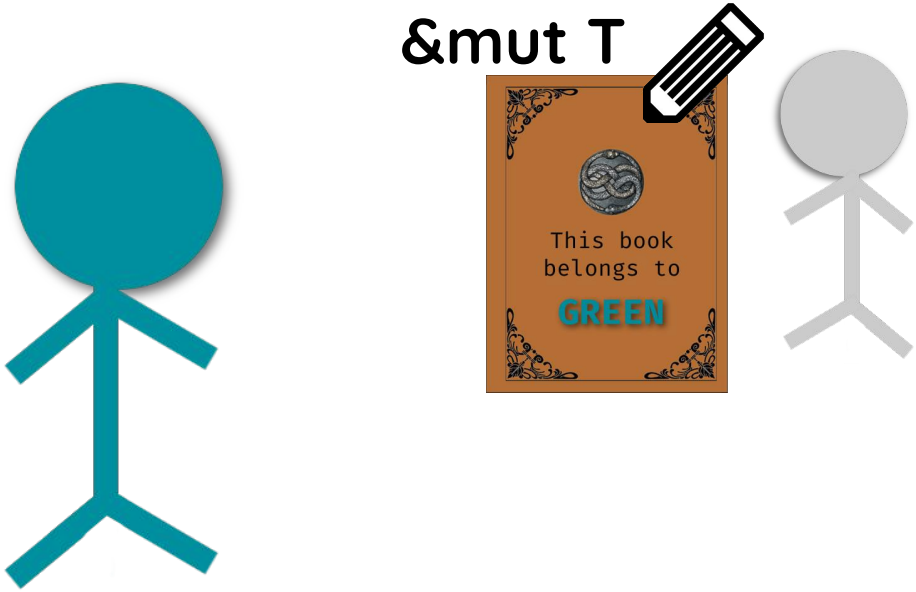
exactly one mutable reference

Borrowing with **&mut T**

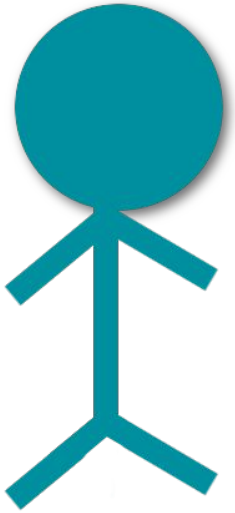


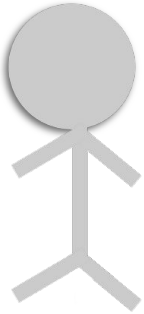


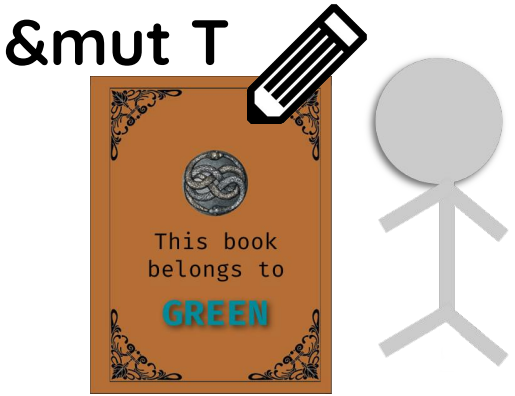
Borrowing with **&mut T**



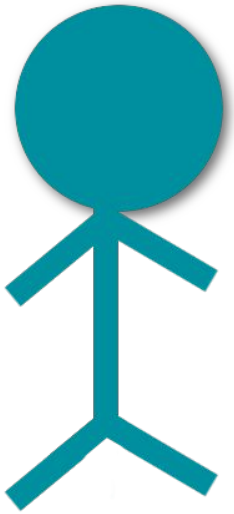
Borrowing with **&mut T**

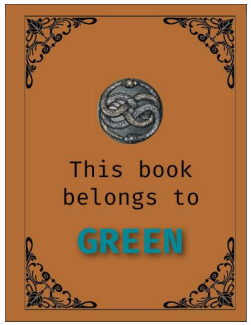






Borrowing with **&mut T**





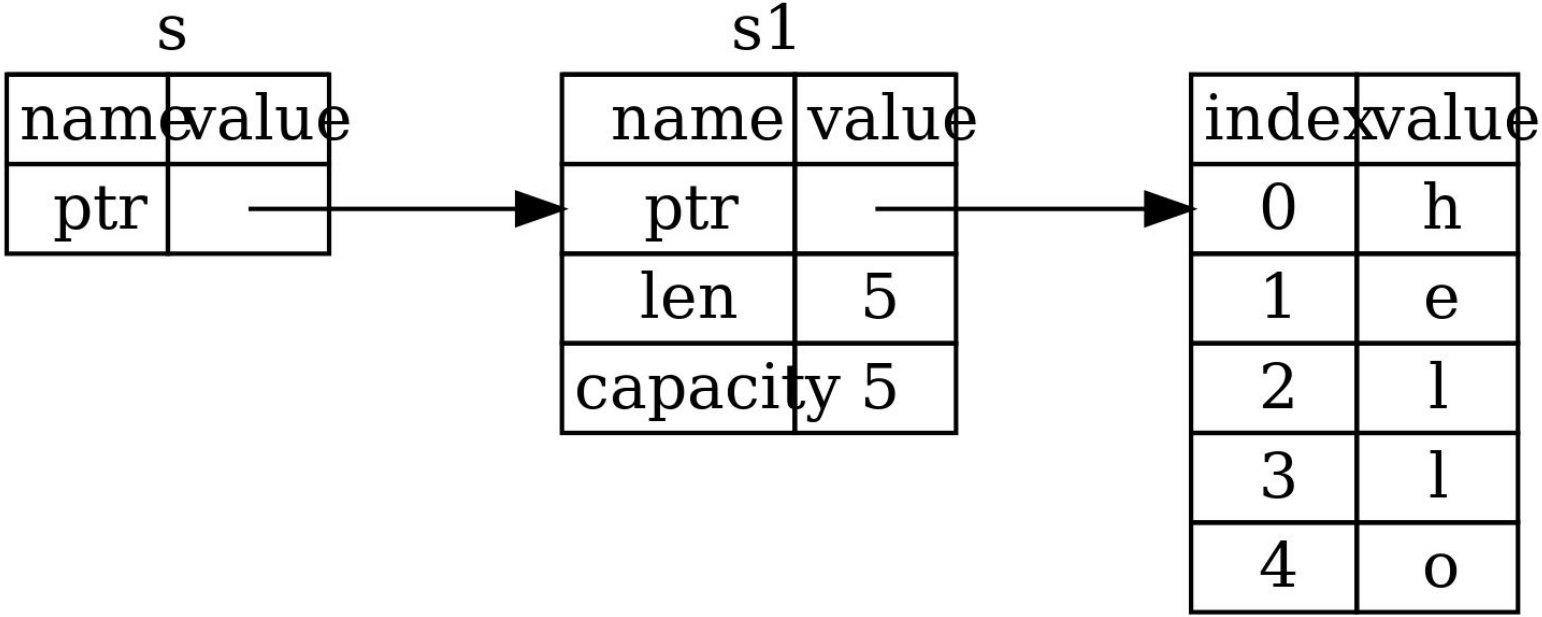
Green can use **its** book again

Borrowing

```
let s1 = String::from("hello");
let s = &s1;

println!("{}, world!", s1);
println!("{}, world!", s);
```

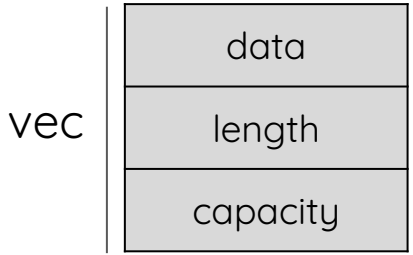
Borrowing (Memory)



Borrowing

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

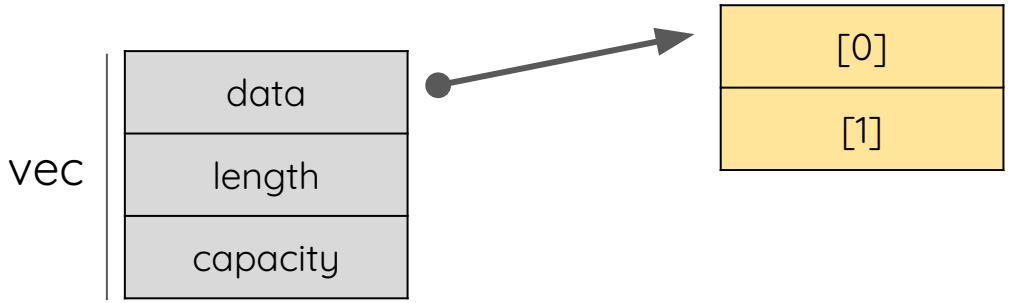
```
fn user(vec: &Vec<i32>) {
    //…
}
```



Borrowing

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

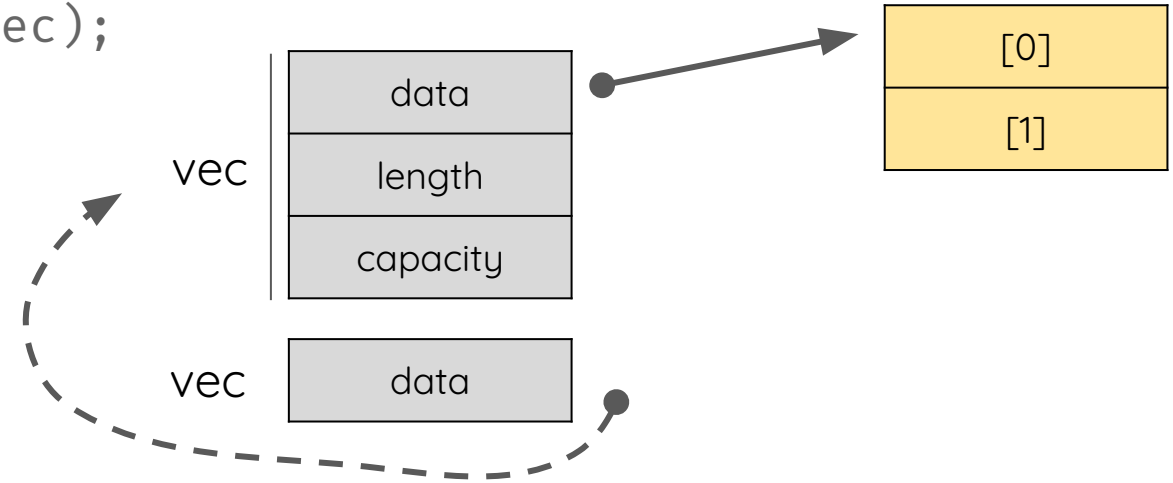
```
fn user(vec: &Vec<i32>) {
    //…
}
```



Borrowing

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

```
fn user(vec: &Vec<i32>) {
⇨   //…
}
```



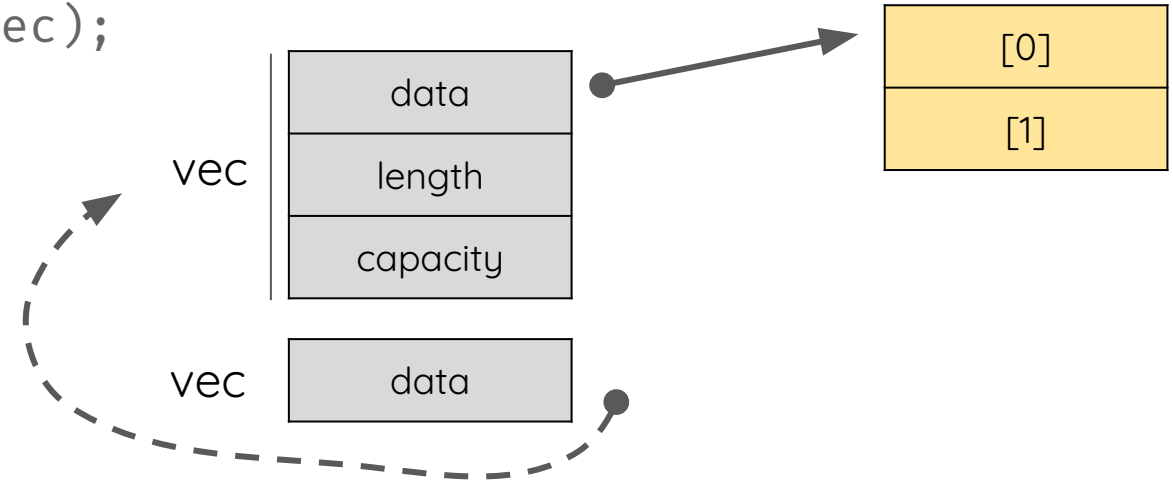
Borrowing

shared ref to vec

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

```
fn user(vec: &Vec<i32>) {
⇨   //…
}
```

loan out vec



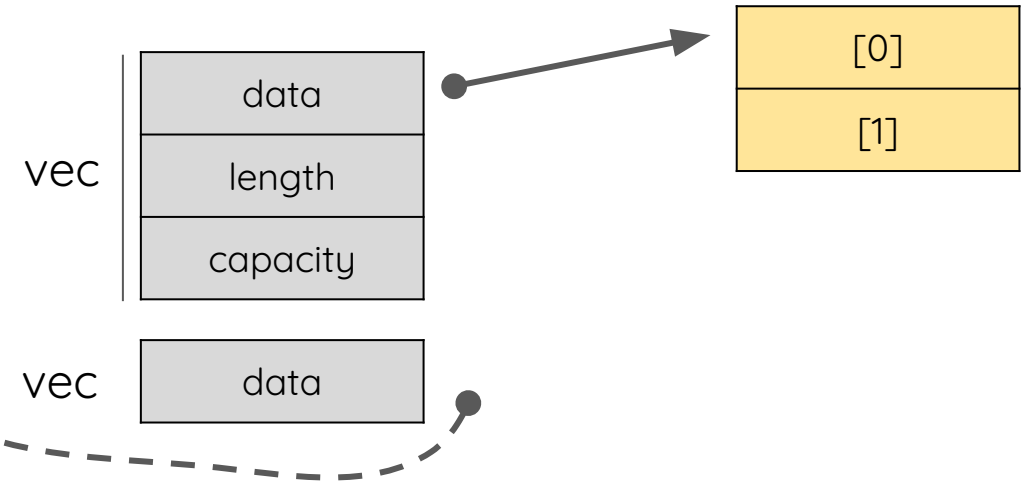
Borrowing

what happens if I try to modify it?

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

loan out vec

```
fn user(vec: &Vec<i32>) {
⇨   vec.push(3);
}
```



Borrowing (Error)

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

```
fn user(vec: &Vec<i32>) {
⇨   vec.push(3);
}
```

```
error: cannot borrow immutable borrowed content `*vec` as mutable
  --> src/main.rs:23:5
   |
22 | fn user(vec: &Vec<i32>) {
   |              --------- use `&mut Vec<i32>` here to make mutable
23 |     vec.push(3);
   |     ^^^
```

Lifetimes

Lifetimes

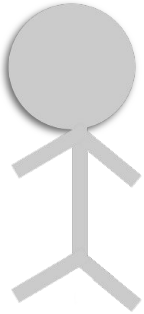
Remember that the owner has always the ability to destroy (deallocate) a resource!

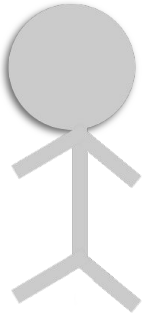
Define the scope for which a reference is valid.

Every reference has a lifetime.

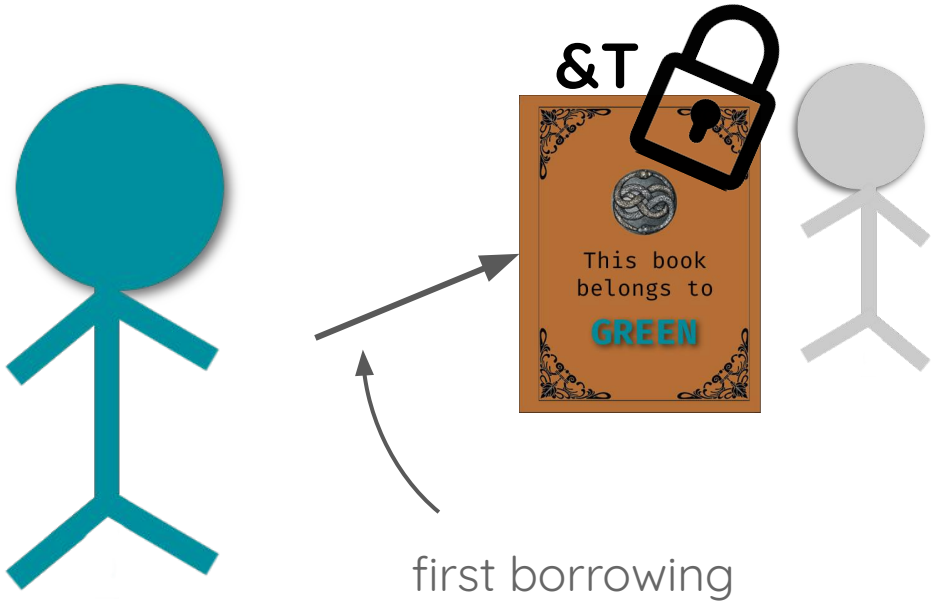
Most of the time inferred by the compiler.
In more complex scenarios compiler **needs an hint**.

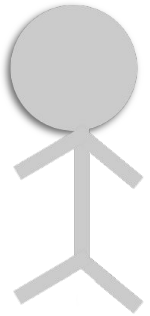
Lifetimes



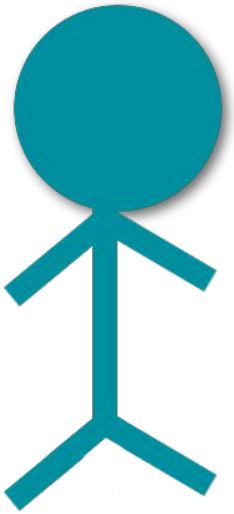


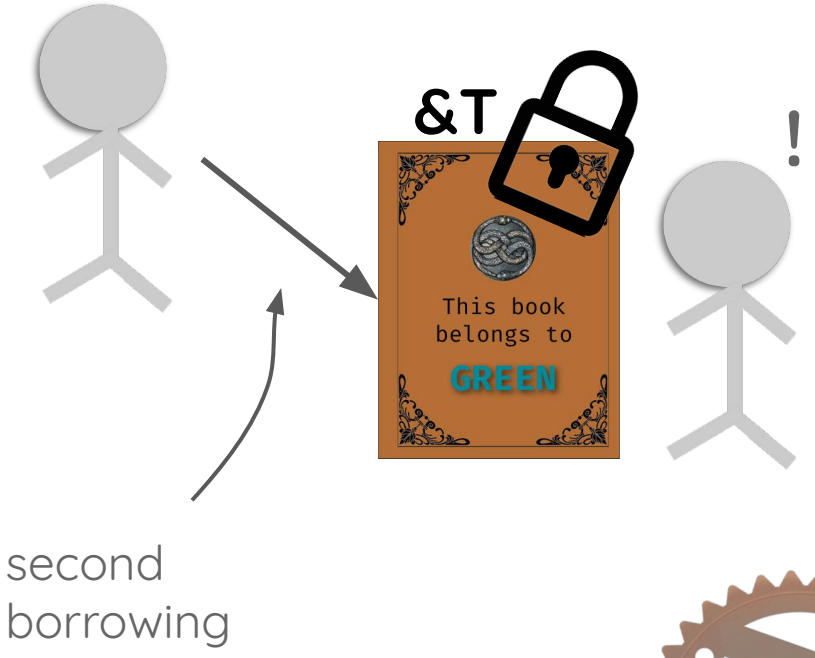
Lifetimes





Lifetimes





Lifetimes

Lifetime

```
struct Foo<'a> {
    x: &'a i32,
}

fn main() {
    let x;                      // -+ x goes into scope
                                //  |
    {                           //  |
        let y = &5;             // ---+ y goes into scope
        let f = Foo { x: y };   // ---+ f goes into scope
        x = &f.x;               //  | | error here
    }                           // ---+ f and y go out of scope
                                //  |
    println!("{}", x);          //  |
}                               // -+ x goes out of scope
```

Lifetime - first example

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}

fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

Lifetime - first example

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}
```

```
fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

Lifetime - first example

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}
```

first borrowing

second borrowing
(we return something that is **not ours**)

```
fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

Lifetime - first example

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}
```

first borrowing

second borrowing
(we return something that is **not ours**)

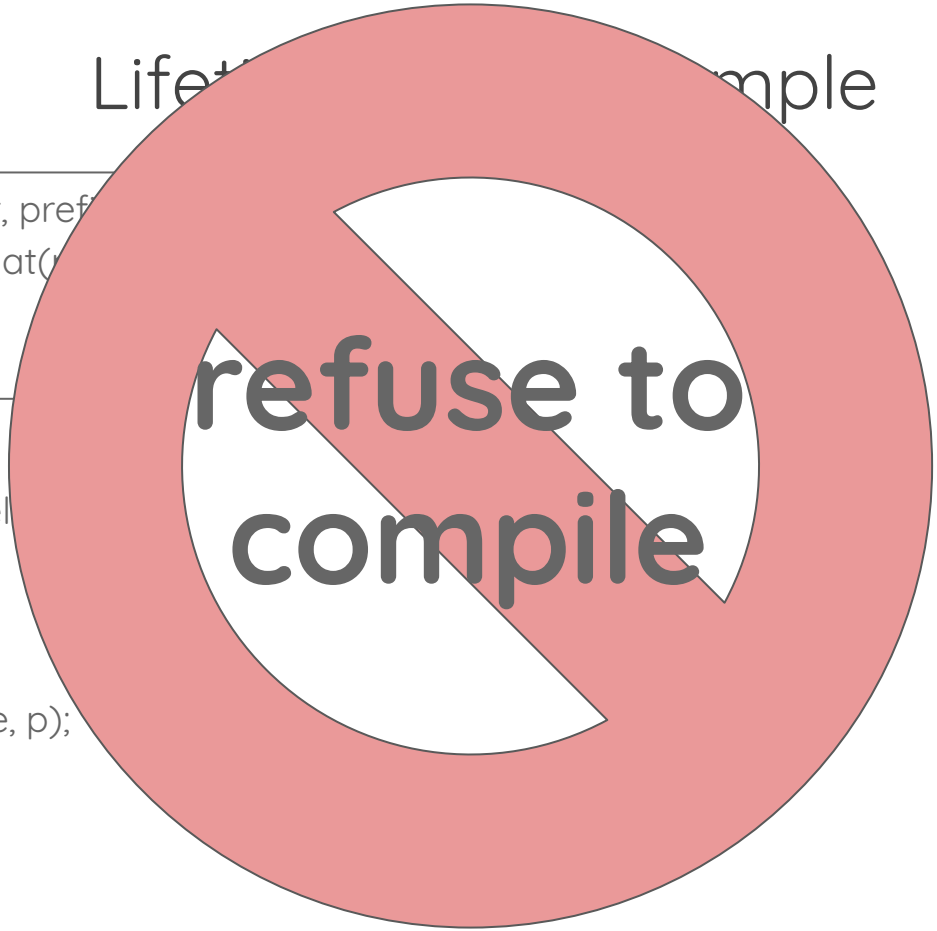
```
fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

we know that “s2” is valid as long as “line” is valid, but **compiler doesn’t know**

Lifet… …mple

```
fn skip_prefix(line: &str, pref
    let (s1,s2) = line.split_at(
    s2
}
```

```
fn print_hello() {
    let line = "lang:en=Hel
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```



Lifetime - first example

```
fn skip_prefix(line: &str, prefix: &str) -> &str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}

fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let prefix = "lang:en=";
        v = skip_prefix(line, prefix);
    }
    println!("{}", v);
}
```

```
error[E0106]: missing lifetime specifier
  --> src/main.rs:37:45
   |
37 | fn skip_prefix(line: &str, prefix: &str) -> & str {
   |                                             ^ expected lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `line` or `prefix`
```

Lifetime - example reviewed

```
fn skip_prefix<'a>(line: &'a str, prefix: &str) -> &'a str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}

fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

Lifetime - example reviewed

```
fn skip_prefix<'a>(line: &'a str, prefix: &str) -> &'a str {
    let (s1,s2) = line.split_at(prefix.len());
    s2
}

fn print_hello() {
    let line = "lang:en=Hello World!";
    let v;
    {
        let p = "lang:en=";
        v = skip_prefix(line, p);
    }
    println!("{}", v);
}
```

borrowing source is now explicit, through the **lifetime parameter**

Hello World!

String vs &str

str

str is a sequence of chars, and you can looks it as a string slice and they are stored with their len in memory

```
let s: &str = "Rust Evangelism Strike Force";
```

Supports UTF-8

```
let hello: &str = "こんにちは";
println!("{}", hello); // output: こんにちは
```

str limitations

str size is unknown at compile time

str management is complicate (need **lifetime**)

two **str** cannot been added

Ugly to see and impossible to do

```
fn get_str() → str {
    *("hello world")
}
```

```
   Compiling playground v0.0.1 (/playground)
error[E0277]: the size for values of type `str` cannot be known at compilation time
 --> src/main.rs:4:17
  |
4 | fn get_str() -> str {
  |                 ^^^ doesn't have a size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for `str`
  = note: to learn more, visit <https://doc.rust-lang.org/book/ch19-04-advanced-types.html#dynamicall
  = note: the return type of a function must have a statically known size

error[E0277]: the size for values of type `str` cannot be known at compilation time
 --> src/main.rs:9:5
  |
9 | let s = get_str();
  |     ^ doesn't have a size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for `str`
  = note: to learn more, visit <https://doc.rust-lang.org/book/ch19-04-advanced-types.html#dynamicall
  = note: all local variables must have a statically known size
  = help: unsized locals are gated as an unstable feature

error[E0277]: the size for values of type `str` cannot be known at compilation time
 --> src/main.rs:9:9
  |
9 | let s = get_str();
  |         ^^^^^^^ doesn't have a size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for `str`
  = note: to learn more, visit <https://doc.rust-lang.org/book/ch19-04-advanced-types.html#dynamicall
  = note: the return type of a function must have a statically known size
```

Management is complicate due to rust Memory safety restrictions

```
// this not work, return size must be known at compile time
fn get_str() → &str {
    "hello world"
}
```

Needs lifetime

```
fn get_str<'a>() → &'a str {
    "hello world"
}
```

By default they are compiled as static

```
fn set_str<'a>(s: &'a mut str) → &'a str {
    s = "hello world";
    s
}

fn main() {
  let mut s = "";
  let s = set_str(s);
  println!("{}", s);
}
```

```rust
fn set_str<'a>(s: &'a mut str) -> &'a str {
    s = "hello world";
    s
}

fn main() {
  let mut s = "";
  // error: differs in mutability
  let s = set_str(s);
  println!("{}", s);
}
```

```rust
fn set_str<'a>(s: &'a mut str) -> &'a str {
    // s is declared mut and "hello world" is static
    s = "hello world";
    s
}

fn main() {
  let mut s = "";
  // error: differs in mutability
  let s = set_str(s);
  println!("{}", s);
}
```

you cannot add two or more **str**

```
let rust = "Rust";
let rome = "Rome";
println!("{}", rust + " " + rome);
```

```
   Compiling playground v0.0.1 (/playground)
error[E0369]: binary operation `+` cannot be applied to type `&str`
 --> src/main.rs:7:21
  |
7 | println!("{}", rust + " " + rome);
  |                ---- ^ --- &str
  |                |    |
  |                |    `+` cannot be used to concatenate two `&str` strings
  |                &str
help: `to_owned()` can be used to create an owned `String` from a string reference. String concatenat
  |
7 | println!("{}", rust.to_owned() + " " + rome);
  |                ^^^^^^^^^^^^^^^^

error: aborting due to previous error

For more information about this error, try `rustc --explain E0369`.
error: could not compile `playground`.

To learn more, run the command again with --verbose.
```

String

It is allocated memory string type

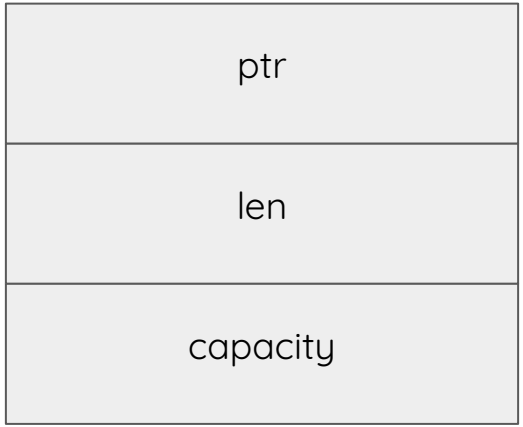
It has ownership over the content

Supports UTF-8

No lifetime required

Easy **String -> &str** and **&str -> String**

Memory layout



String

```
let s: String = String::new();
```

String

```
let s = String::from("Hello world");
```

It is growable

```rust
let s = String::from("Hello world");
s.push(" Rust!");
println!("{}", s); // output: Hello world Rust!
```

No needs lifetime

```
fn get_string() → String {
  String::from("Hello world")
}
let s = get_string();
println!("{}", s); // output: Hello world
```

Can be added

```
let rust = "Rust".to_owned();
let rome = "Rome";
let space = " ";
let r_r = rust + space + rome;
println!("{}", r_r); // output: Rust Rome
```

Can be added

```
let rust = "Rust ".to_owned();
let rome = "Rome".to_owned();
let r_r = rust + rome;
println!("{}", r_r); // output: Rust Rome
```

Easy **&str -> String**

```
let s: String = "Hello world".into();
let s = "Hello world".to_owned();
```

Easy **String -> &str**

```
let s = "Hello world".to_owned();
let s: &str = &s;
```

str vs String

```rust
struct Object<'a> {
  field: &'a str,
}

impl<'a> Object<'a> {
  fn new(s: &'a str) -> Self {
    Object { field: s }
  }
  fn field(&self) -> &'a str {
    self.field
  }
}
```

```rust
struct Object {
  field: String,
}

impl Object {
  fn new(s: String) -> Self {
    Object { field: s }
  }
  fn field(&self) -> String {
    self.field
  }
}
```

Next lesson

Martedì 26 novembre orario 18-20

Per info e domande:

alex179ohm@gmail.com

enrico.risa@gmail.com

Oggetto: **corso rust sapienza**

Slack: http://rust-italia.herokuapp.com channel: #rust-roma