The Rust Programming Language for Java Developers

Vienna Java Meetup, May 2022

fettblog.eu - rust-linz.at - rust-training.eu - oida.dev

Seas!

90s The wilderness years
VB, PHP, Perl

2000 Java at JKU
C, C++ (OpenGL)

2003 Java on the job!

2006 C++, C#

2011 JavaScript

Java is to JavaScript
what
Alf is to Gandalf

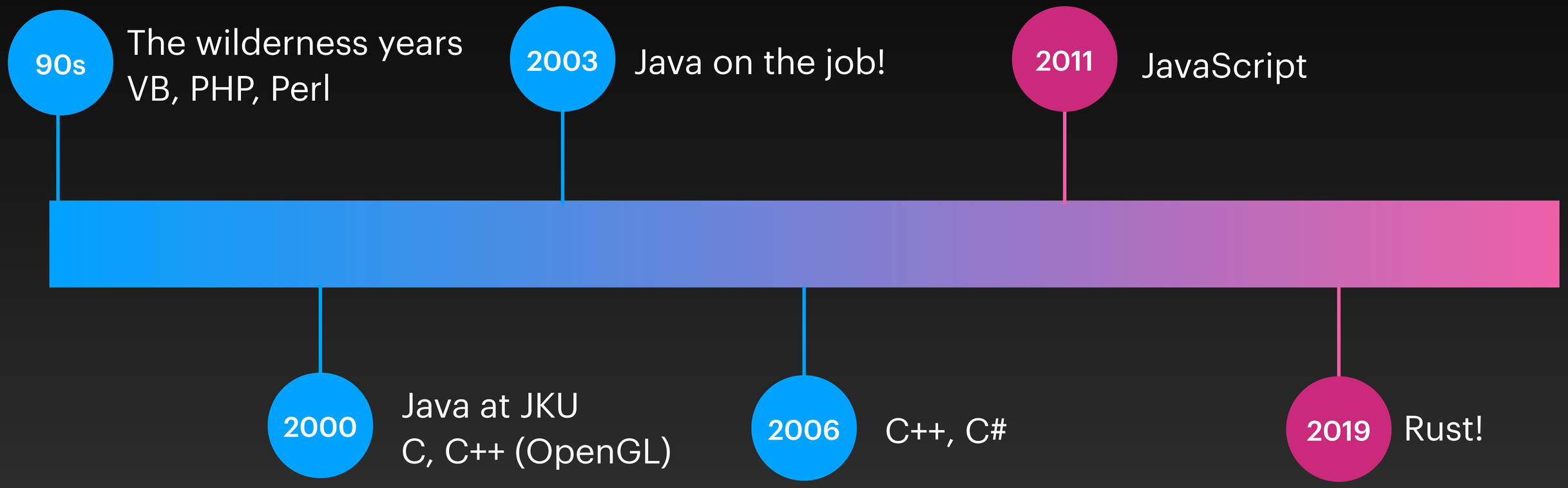
90s The wilderness years
VB, PHP, Perl

2000 Java at JKU
C, C++ (OpenGL)

2003 Java on the job!

2006 C++, C#

2011 JavaScript

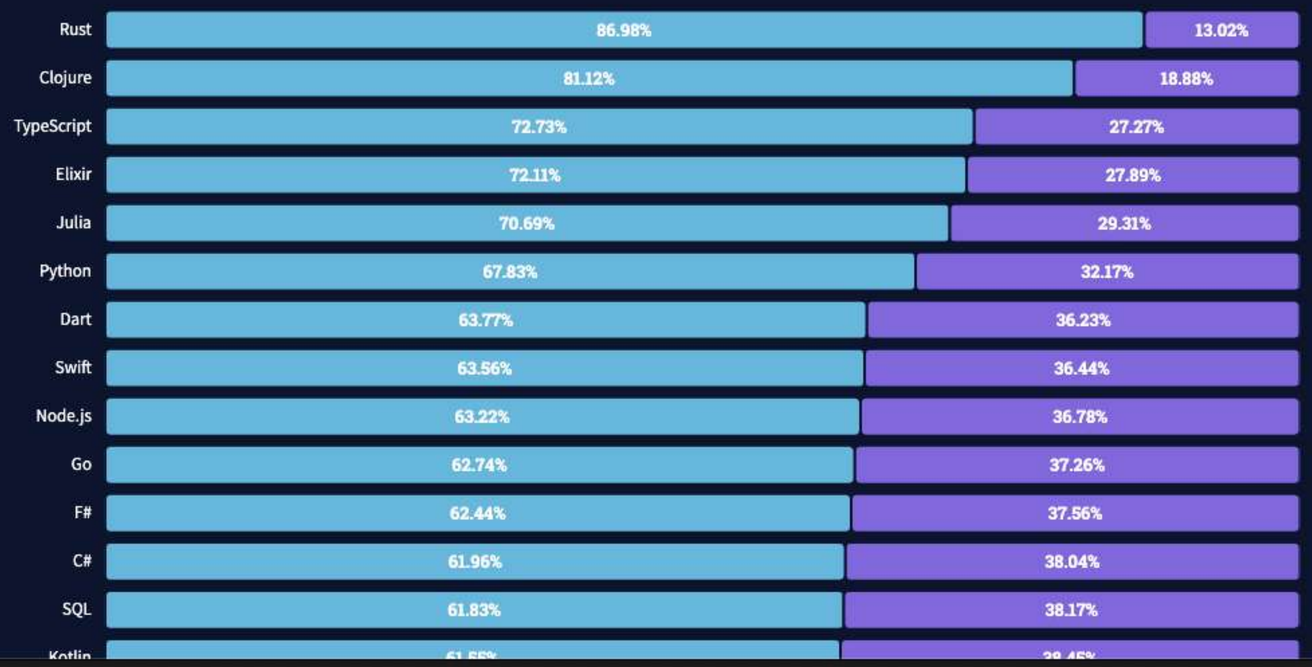

90s
The wilderness years
VB, PHP, Perl
2000
Java at JKU
C, C++ (OpenGL)
2003
Java on the job!
2006
C++, C#
2011
JavaScript
2019
Rust!

Programming, scripting, and markup languages

For the sixth-year, Rust is the most loved language, while Python is the most wanted language for its fifth-year.

Loved vs. Dreaded | Want

82,914 responses



What’s unique about Rust

- Rust is compiled to machine code (via LLVM) — native speed!
- No runtime, optional standard library
- No garbage collection
- Guaranteed memory safety
- “Debug at compile time”
- “Hack without fear”
- “Zero cost abstractions”

Rust in Linux

- No undefined behavior [...], including memory safety and **the absence of data races**.
- **Stricter type system** for further reduction of logic errors.
- A clear distinction between **safe and unsafe** code.
- **Featureful** language [...]
- Extensive freestanding standard library [...]
- Integrated **out of the box tooling** [...]

Memory safety
without garbage collection

Abstraction without overhead

Fearless concurrency

Memory safety without garbage collection

Memory safety

- In unmanaged programming languages:
  - Use after free
  - Double free
  - Buffer overreads and overwrites
  - Null pointers!

Google Online Security Blog: An

security.googleblog.com/2021/09/an-update-on-mem...

Update

Google Security Blog

The latest news and insights from Google on security and safety on the Internet

An update on Memory Safety in Chrome

September 21, 2021

Adrian Taylor, Andrew Whalley, Dana Jansens and Nasko Oskov, Chrome security team

Security is a cat-and-mouse game. As attackers innovate, browsers always have to mount new defenses to stay ahead, and Chrome has invested in ever-stronger multi-process architecture built on sandboxing and site isolation. Combined with fuzzing, these are still our primary lines of defense, but they are reaching their limits, and we can no longer solely rely on this strategy to defeat in-the-wild attacks.

Last year, we showed that more than 70% of our severe security bugs are memory safety problems. That is, mistakes with pointers in the C or C++ languages which cause memory to be misinterpreted.

This sounds like a problem! And, certainly, memory safety is an issue which needs to be taken seriously by the global software engineering community. Yet it's also an opportunity because many bugs have the same sorts of root-causes, meaning we may

Google Online Security Blog: An

security.googleblog.com/2021/09/an-update-on-mem...

Update

Google Security Blog

The latest news and insights from Google on security and safety on the Internet

An update on Memory Safety in Chrome

September 21, 2021

Adrian Taylor, Andrew Whalley, Dana Jansens and Nasko Oskov, Chrome security team

Security is a cat-and-mouse game. As attackers innovate, browsers always have to mount new defenses to stay ahead, and Chrome has invested in ever-stronger multi-process architecture built on sandboxing and site isolation. Combined with fuzzing, these are still our primary lines of defense, but they are reaching their limits, and we can no longer solely rely on this strategy to defeat in-the-wild attacks.

Last year, we showed that more than 70% of our severe security bugs are memory safety problems. That is, mistakes with pointers in the C or C++ languages which cause memory to be misinterpreted.

This sounds like a problem! And, certainly, memory safety is an issue which needs to be taken seriously by the global software engineering community. Yet it's also an opportunity because many bugs have the same sorts of root-causes, meaning we may

Last year, we showed that more than 70% of our severe security bugs are memory safety problems. That is, mistakes with pointers in the C or C++ languages which cause memory to be misinterpreted.

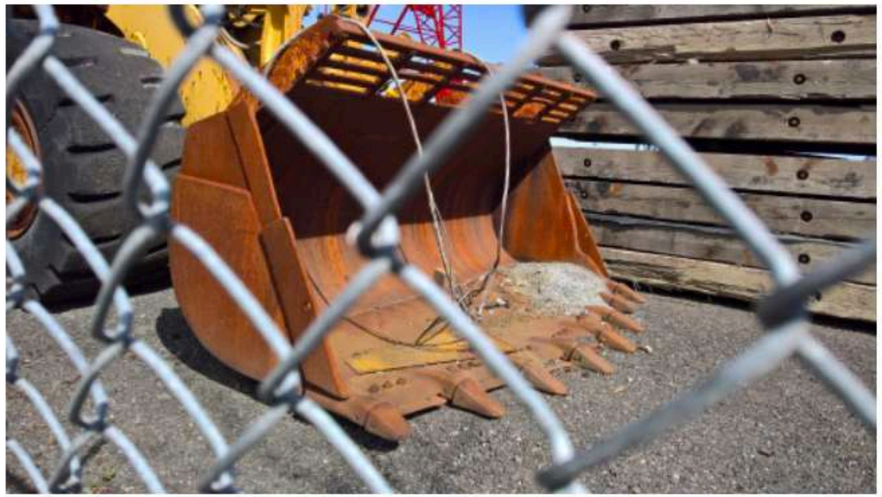
A proactive approach to more s

msrc-blog.microsoft.com/2019/07/16/a-proactive-...

Update

Microsoft Security Response Center

Report an issue | Security Update Guide | About MSRC



A proactive approach to more secure code

Security Research & Defense / By MSRC Team / July 16, 2019 / Memory Safety, Rust, Safe Systems Programming Languages, Secure Development

What if we could eliminate an entire class of vulnerabilities before they ever happened?

Since 2004, the Microsoft Security Response Centre (MSRC) has triaged every reported Microsoft security vulnerability. From all that triage one astonishing fact sticks out: as Matt Miller discussed in his 2019 presentation at BlueHat IL, the majority of vulnerabilities fixed and

https://msrc-blog.microsoft.com

Search ...

Categories

- BlueHat (181)
- Japan Security Team (949)
- MSRC (986)
- Security Research & Defense (372)

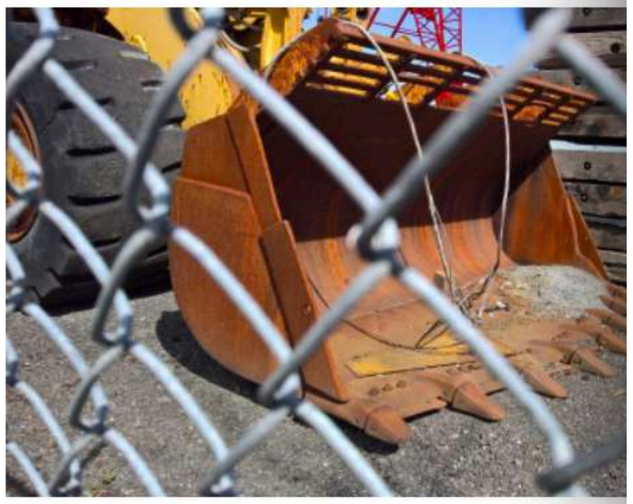
Tags

advisory (60) | ANS (47) | Attack (43) | Attack Vector (68) | Black Hat (33) | BlueHat Security Briefings (55) | Community-based Defense (95) | Defense-in-depth (38)

A proactive approach to more s

msrc-blog.microsoft.com/2019/07/16/a

Microsoft Security Response Cen



A proactive approach to more secure code

Security Research & Defense / By MSRC Team / July 16, 2019 / Memory Safety, Rust, Safe Systems Programming Languages, Secure Development

What if we could eliminate an entire class of vulnerabilities before they ever happened?

Since 2004, the Microsoft Security Response Centre (MSRC) has triaged every reported Microsoft security vulnerability. From all that triage one astonishing fact sticks out: as Matt Miller discussed in his 2019 presentation at BlueHat IL, the majority of vulnerabilities fixed and

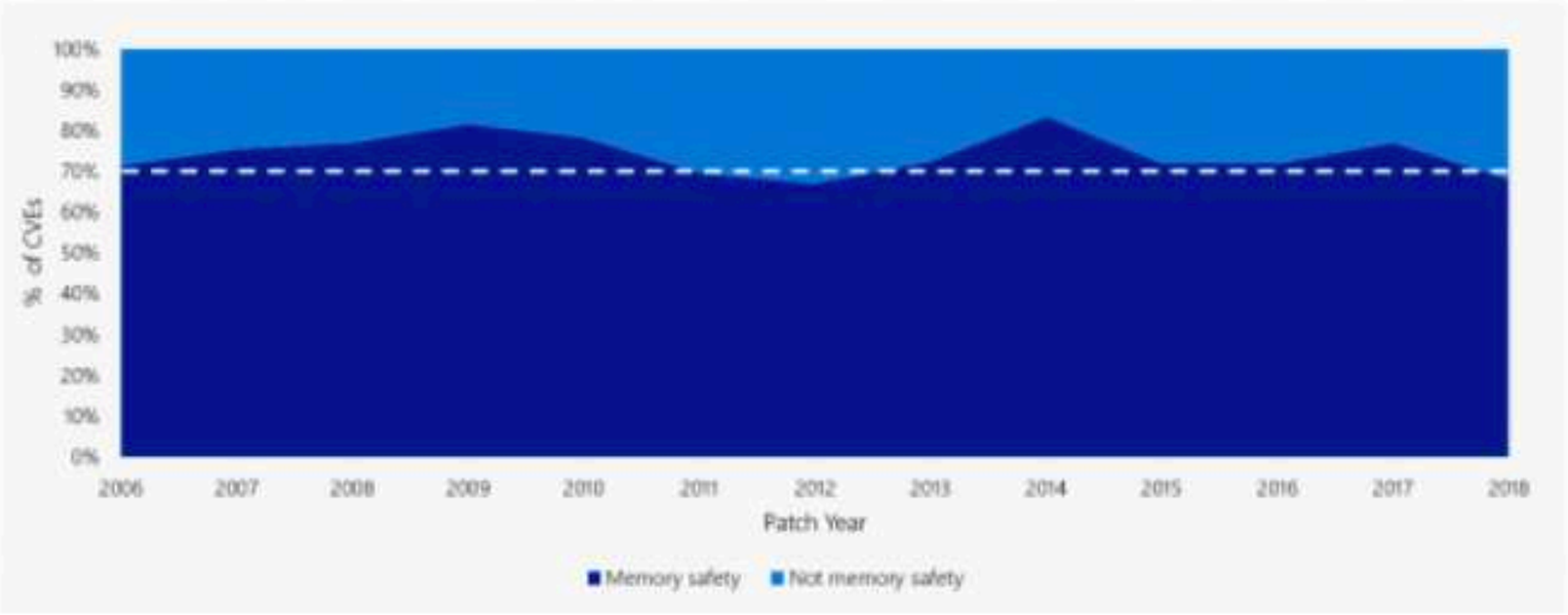


Figure 1: ~70% of the vulnerabilities Microsoft assigns a CVE each year continue to be memory safety issues

Attack (43)

Attack Vector (68)

Black Hat (33)

BlueHat Security Briefings (55)

Community-based Defense (95)

Defense-in-depth (38)

https://msrc-blog.microsoft.com

Ownership

- Each value has an owner
- There is exactly one owner of each piece of data
- When the owner goes out of scope, that data get cleaned up
- Owner can transfer ownership or “borrow” data

Ownership

Exactly one owner

```
fn main() {
    let numbers = vec![1, 1, 2, 3, 5, 8];

    let other_numbers = numbers;

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

Ownership

Exactly one owner

```rust
fn main() {
    let numbers = vec![1, 1, 2, 3, 5, 8];

    let other_numbers = numbers;

    println!("{:?}", other_numbers);
    println!("{:?}", numbers);  💥
}
```

Ownership

Exactly one owner

```rust
fn main() {
    let numbers = vec![1, 1, 2, 3, 5, 8];

    let other_numbers = numbers;

    println!("{:?}", other_numbers);
    println!("{:?}", numbers);  💥
}
```

```
error[E0382]: borrow of moved value: `numbers`
 --> src/main.rs:7:22
  |
2 |     let numbers = vec![1, 1, 2, 3, 5, 8];
  |         ------- move occurs because `numbers` has type `Vec<i32>`, which does not implement the `Copy` trait
3 |
4 |     let other_numbers = numbers;
  |                         ------- value moved here
...
7 |     println!("{:?}", numbers);
  |                      ^^^^^^^ value borrowed here after move

error: aborting due to previous error
```

Ownership

Exactly one owner

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

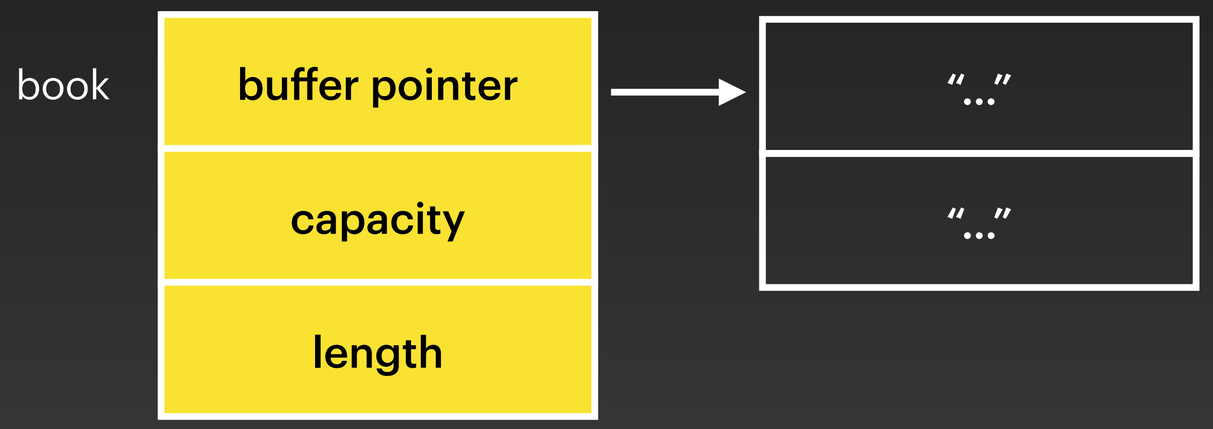
```rust
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```

Ownership

Exactly one owner

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

```rust
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```

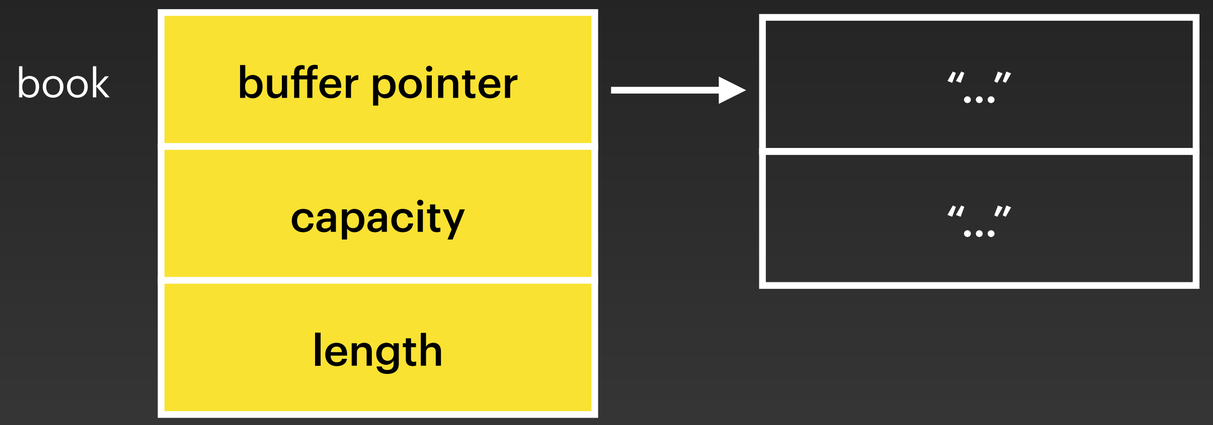


Ownership

Exactly one owner

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

```rust
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```



Ownership

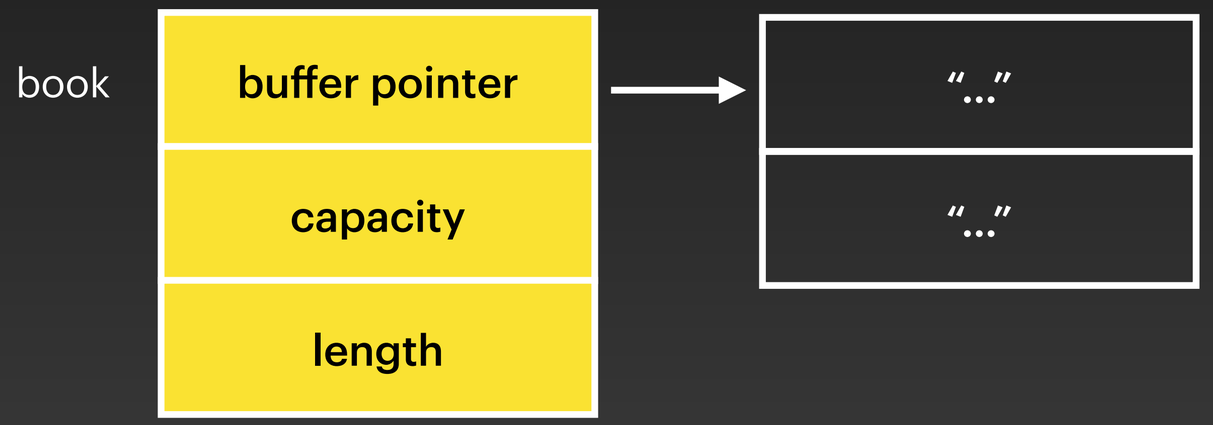
Exactly one owner

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

Give ownership

Take ownership of the vector

```
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```



Ownership

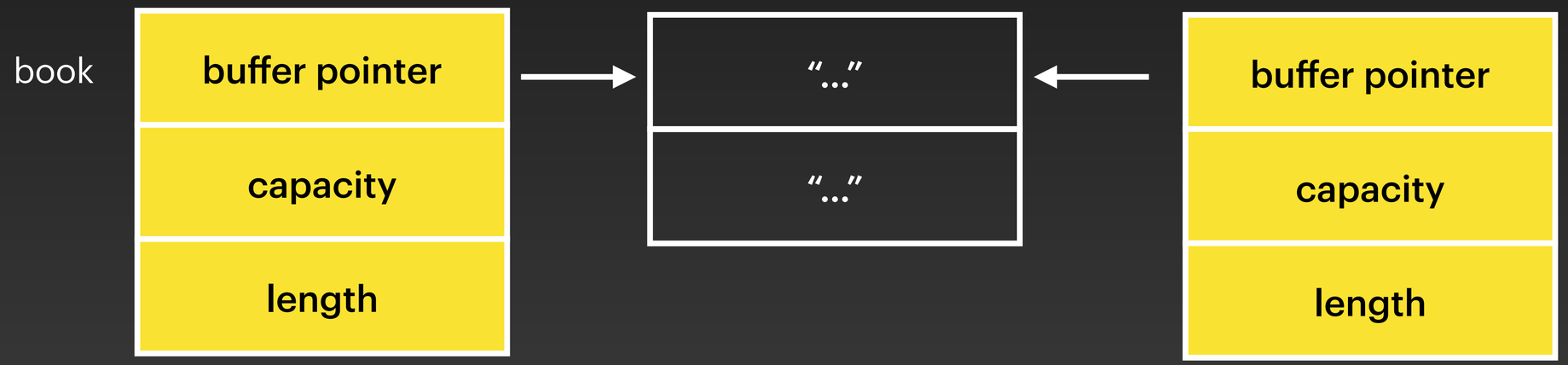
Exactly one owner

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

Give ownership

Take ownership of the vector

```
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```



Ownership

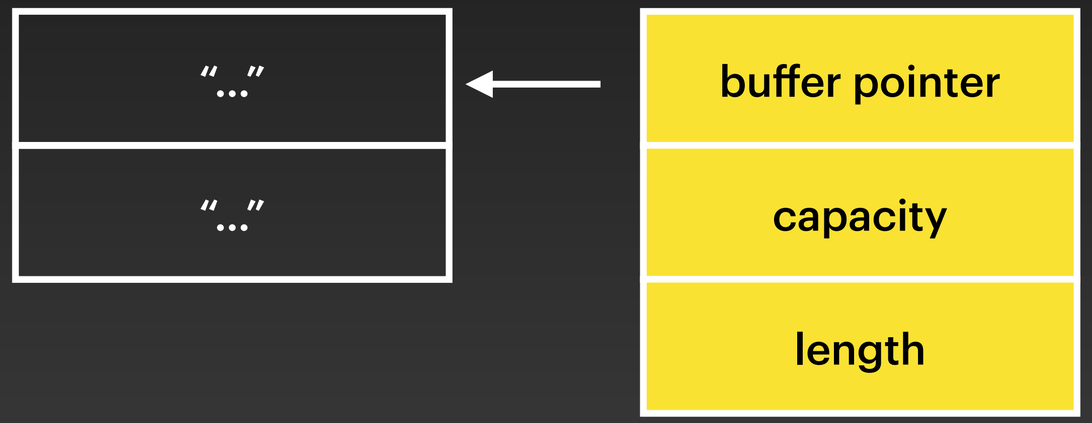
Exactly one owner

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

Give ownership

Take ownership of the vector

```
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```



Ownership

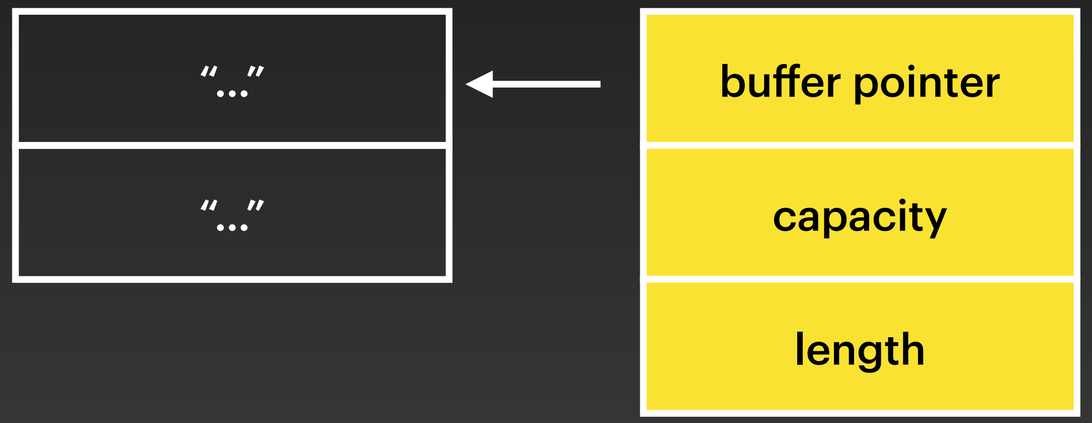
Exactly one owner

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

Give ownership

Take ownership of the vector

```rust
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```



Ownership

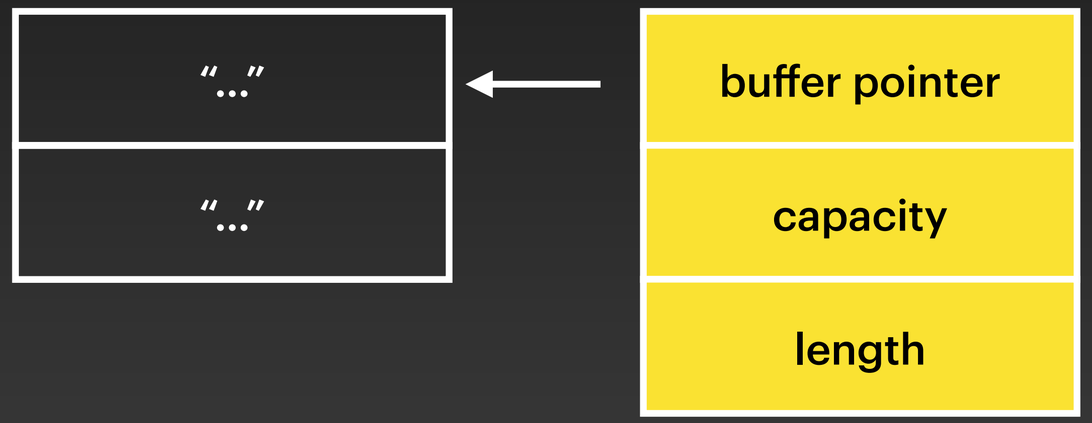
Exactly one owner

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

Give ownership

Take ownership of the vector

```
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```



Ownership

Exactly one owner

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

Give ownership

Take ownership of the vector

```
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```

Ownership

Exactly one owner

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

Give ownership

Take ownership of the vector

```
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```

Ownership

Exactly one owner

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(book);
    publish(book);
}
```

Give ownership

ERROR

Take ownership of the vector

```
fn publish(book: Vec<String>) {
    println!("{:?}", book);
}
```

Ownership

- There are no copy constructors
- Moves are enforced at **compile time**!
- For book to be used multiple times, you can .clone() your data structure
- This is however, expensive!
- There are of course better ways to do that ...

Borrowing

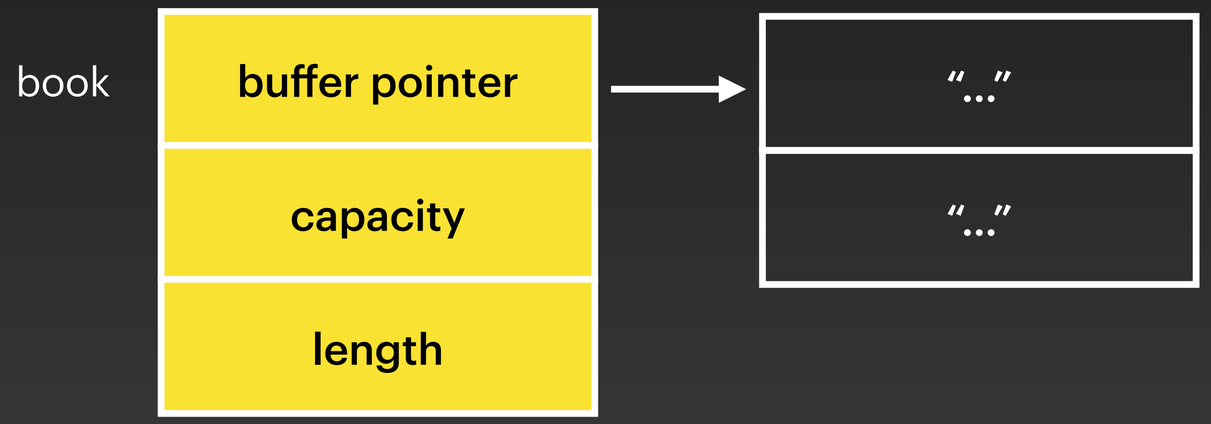
A shared reference

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(&book);
    publish(&book);
}
```

Borrow the vector, create a reference

A reference to a vector

```
fn publish(book: &Vec<String>) {
    println!("{:?}", book);
}
```



Borrowing

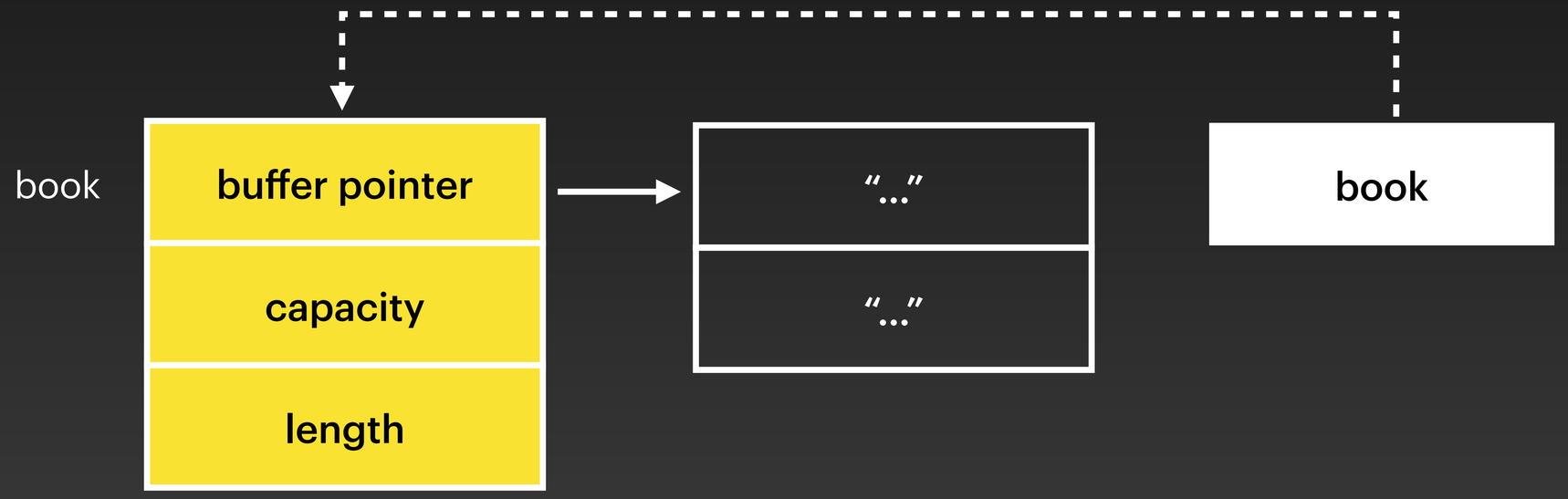
A shared reference

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(&book);
    publish(&book);
}
```

Borrow the vector, create a reference

A reference to a vector

```
fn publish(book: &Vec<String>) {
    println!("{:?}", book);
}
```



Borrowing

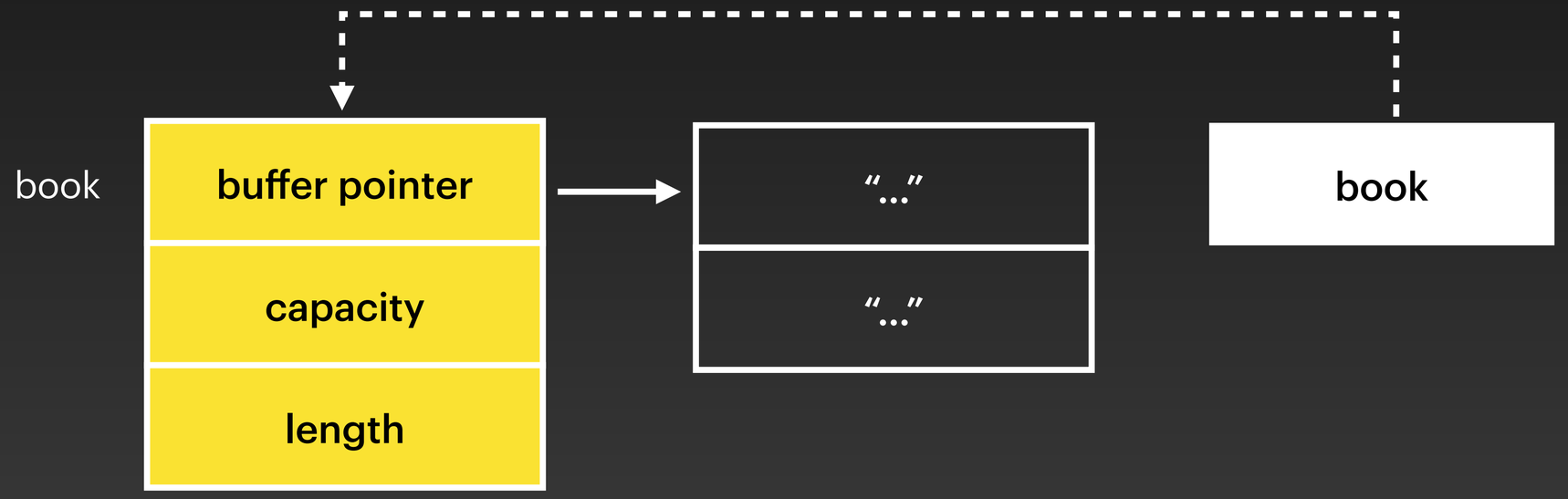
A shared reference

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(&book);
    publish(&book);
}
```

Borrow the vector, create a reference

A reference to a vector

```rust
fn publish(book: &Vec<String>) {
    println!("{:?}", book);
}
```



Borrowing

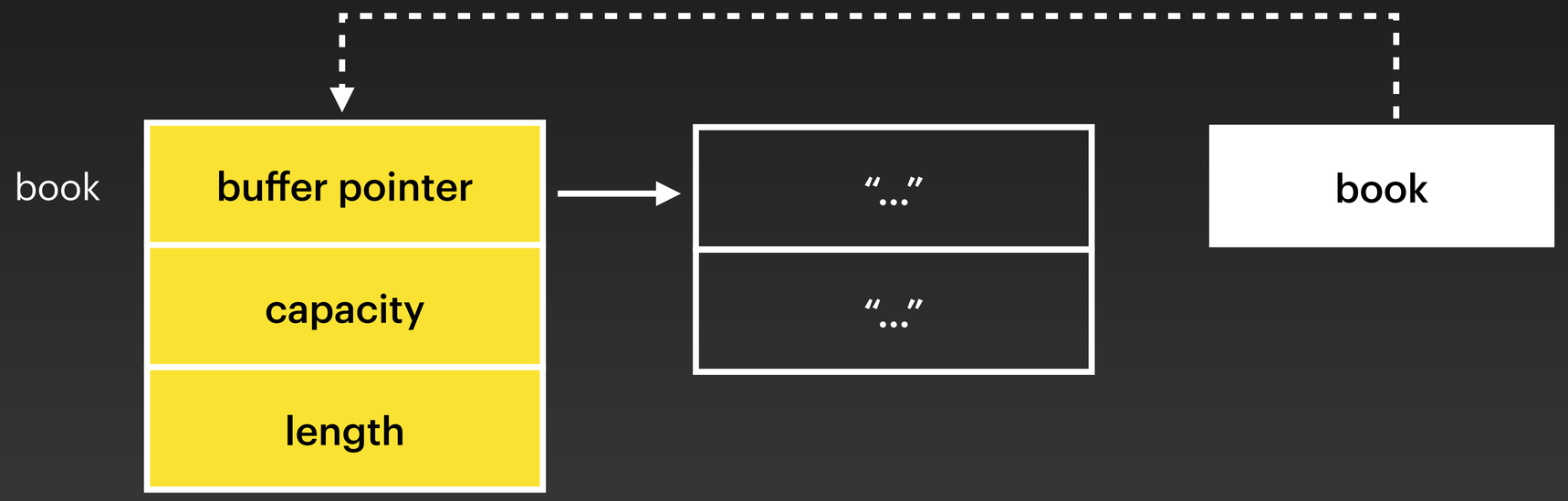
A shared reference

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(&book);
    publish(&book);
}
```

Borrow the vector, create a reference

A reference to a vector

```
fn publish(book: &Vec<String>) {
    println!("{:?}", book);
}
```



Borrowing

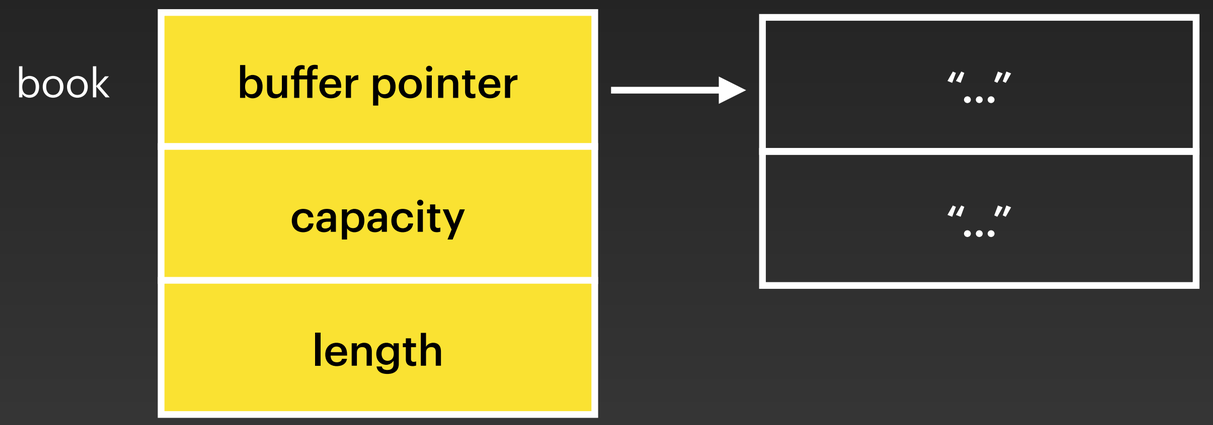
A shared reference

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(&book);
    publish(&book);
}
```

Borrow the vector, create a reference

A reference to a vector

```
fn publish(book: &Vec<String>) {
    println!("{:?}", book);
}
```



Borrowing

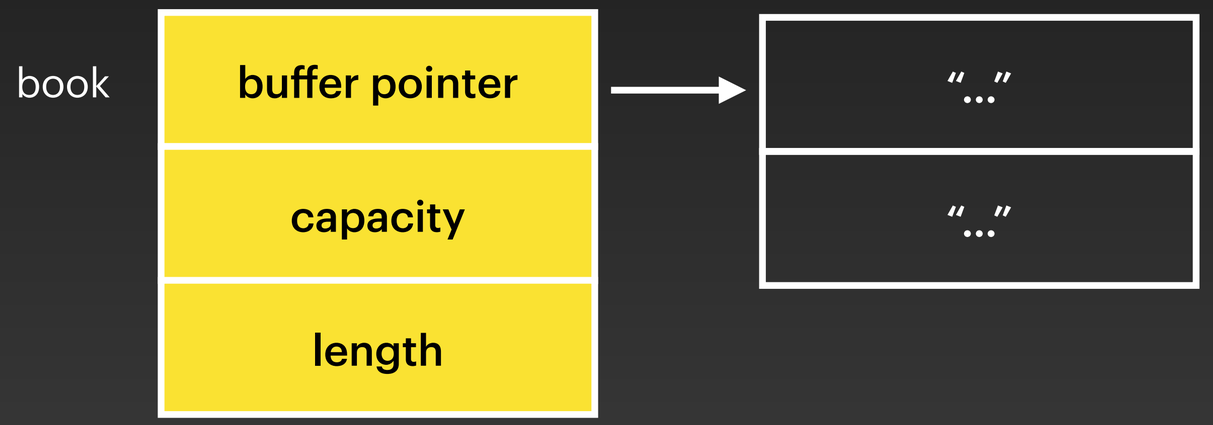
A shared reference

```
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    publish(&book);
    publish(&book);
}
```

Borrow the vector, create a reference

A reference to a vector

```
fn publish(book: &Vec<String>) {
    println!("{:?}", book);
}
```

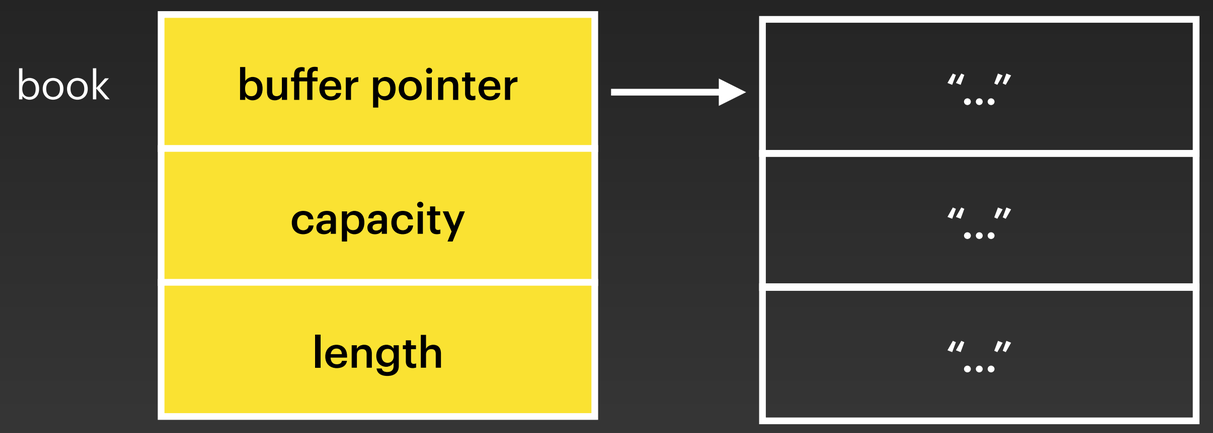


Borrowing

A mutable borrow

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    edit(&mut book);
    edit(&mut book);
}
```

```rust
fn edit(book: &mut Vec<String>) {
    book.push("...".to_string());
}
```

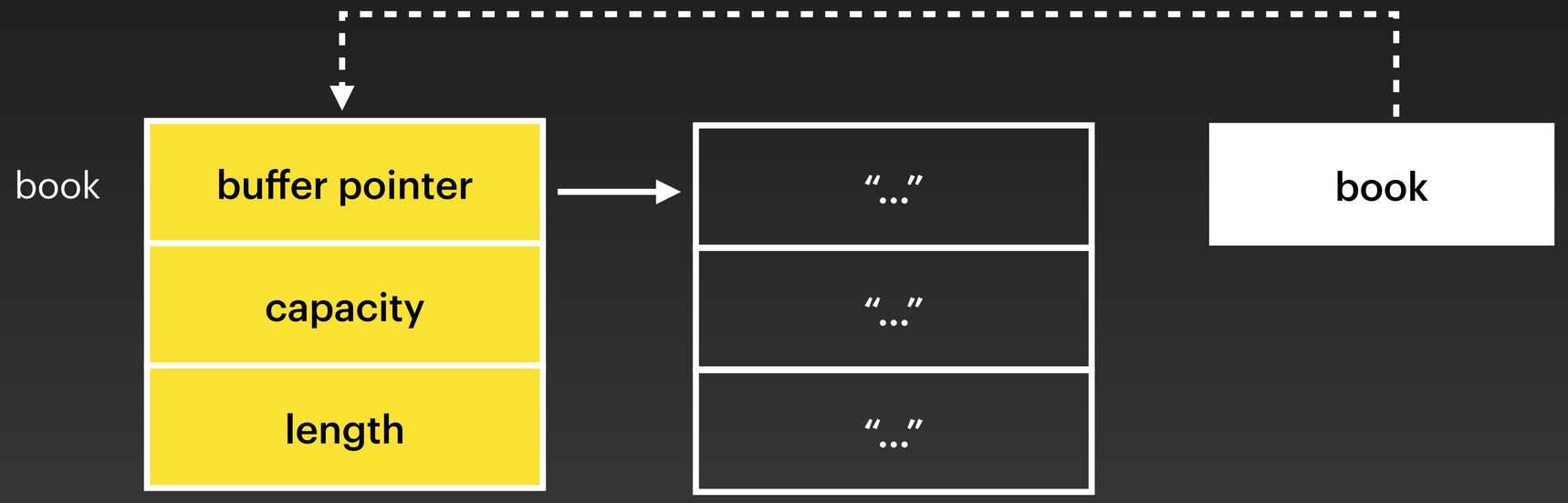


Borrowing

A mutable borrow

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    edit(&mut book);
    edit(&mut book);
}
```

```rust
fn edit(book: &mut Vec<String>) {
    book.push("...".to_string());
}
```



Borrowing

A mutable borrow

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    edit(&mut book);
    edit(&mut book);
}
```

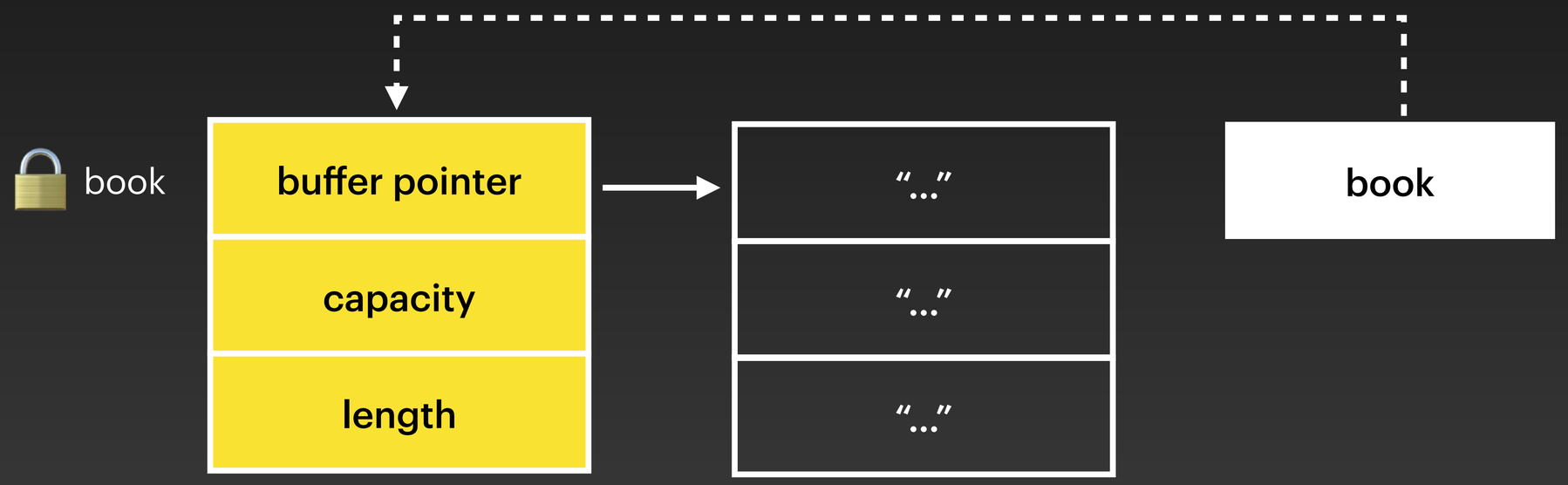
```rust
fn edit(book: &mut Vec<String>) {
    book.push("...".to_string());
}
```

Borrowing

A mutable borrow

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    edit(&mut book);
    edit(&mut book);
}
```

```rust
fn edit(book: &mut Vec<String>) {
    book.push("...".to_string());
}
```

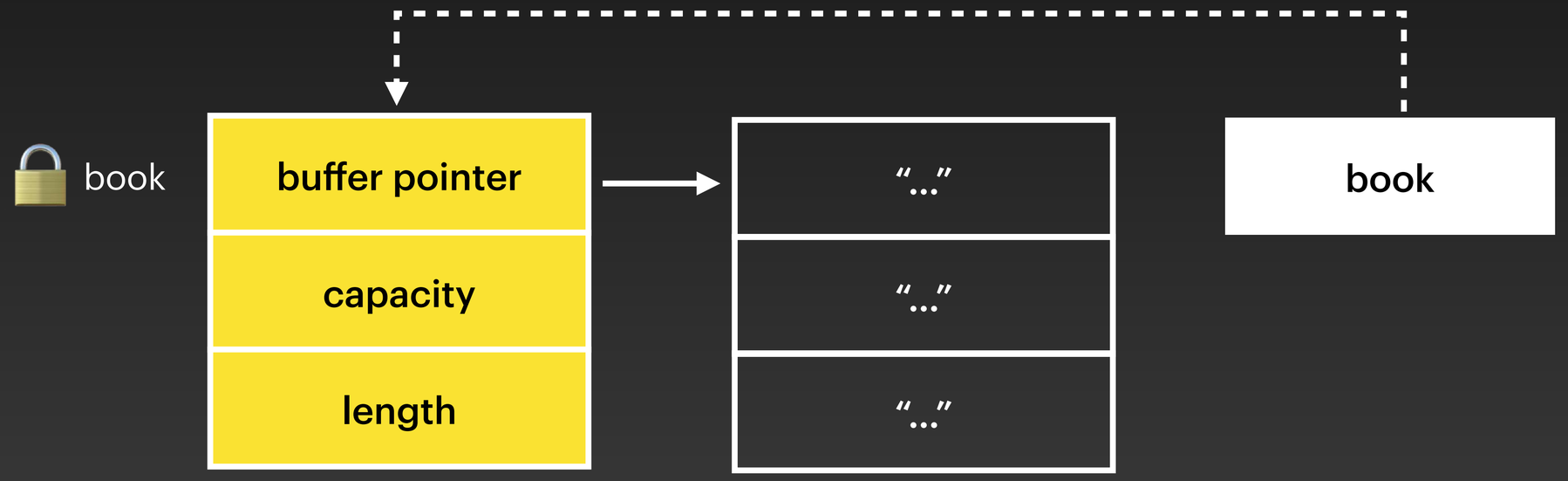


Borrowing

A mutable borrow

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    edit(&mut book);
    edit(&mut book);
}
```

```rust
fn edit(book: &mut Vec<String>) {
    book.push("...".to_string());
}
```

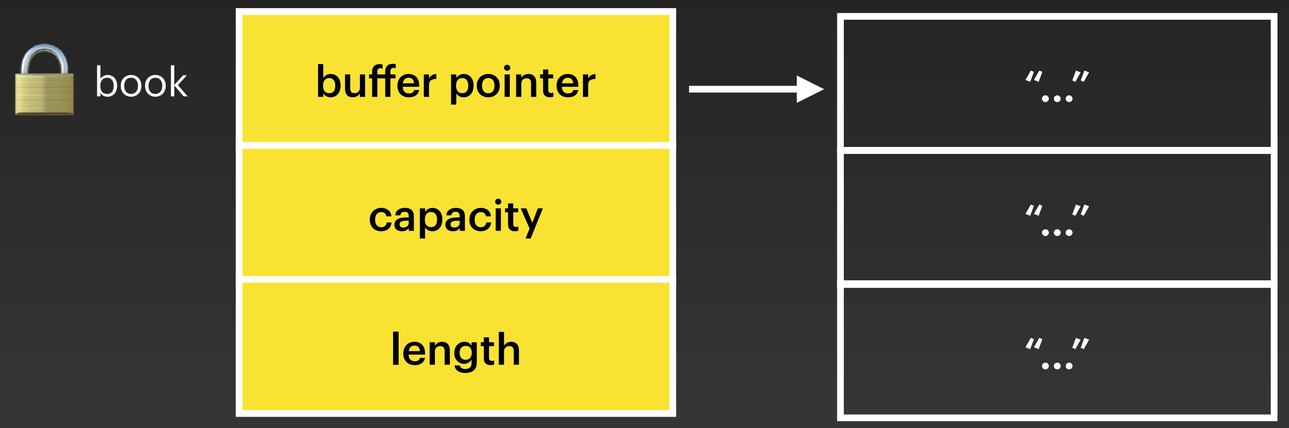


Borrowing

A mutable borrow

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    edit(&mut book);
    edit(&mut book);
}
```

```rust
fn edit(book: &mut Vec<String>) {
    book.push("...".to_string());
}
```

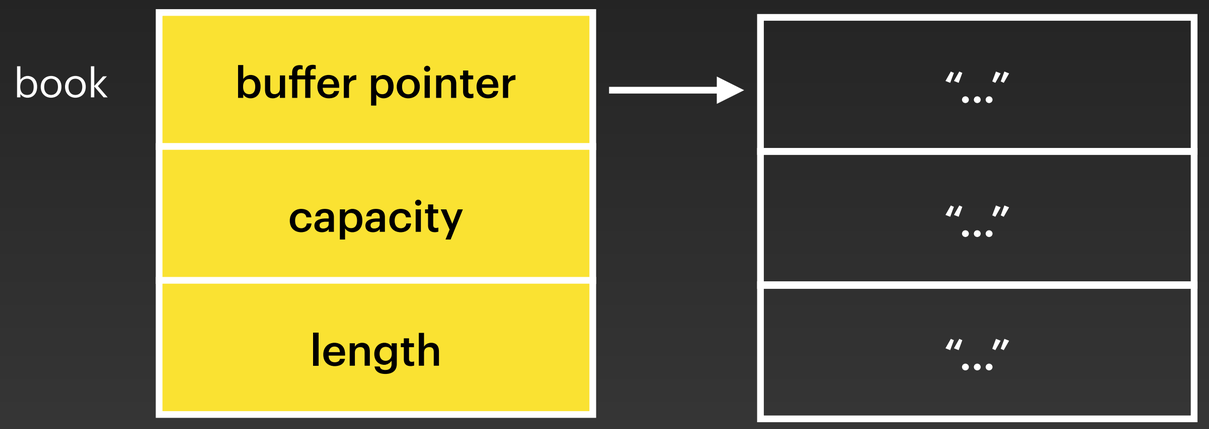


Borrowing

A mutable borrow

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    edit(&mut book);
    edit(&mut book);
}
```

```rust
fn edit(book: &mut Vec<String>) {
    book.push("...".to_string());
}
```

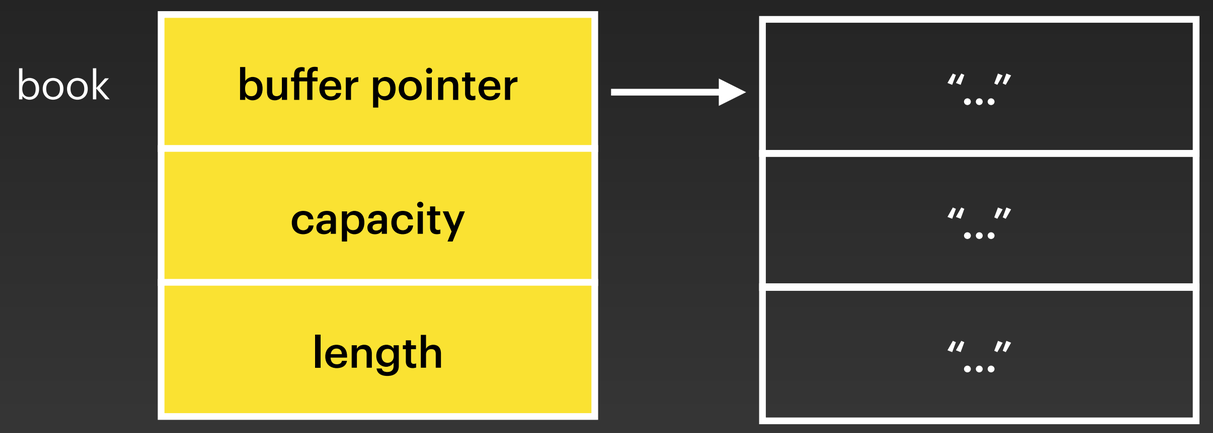


Borrowing

A mutable borrow

```rust
fn main() {
    let mut book = Vec::new();

    book.push("...".to_string());
    book.push("...".to_string());

    edit(&mut book);
    edit(&mut book);
}
```

```rust
fn edit(book: &mut Vec<String>) {
    book.push("...".to_string());
}
```



Ownership and Borrowing

Don’t break your friends toys

Type	Ownership	Alias?	Mutate?
T	Owned		✅
&T	Shared reference	✅	
&mut T	Mutable reference		✅

Abstraction without overhead

What are traits

- Traits define shared behaviour between types
- Behaviour is defined in an abstract way
- Traits have similarities to interfaces in other programming languages
- This doesn’t mean Rust has traditional OO! There are differences

The coherence rule

- It’s not possible to implement a trait you don’t own for a type you don’t own
- You can implement your traits for other types
- You can implement your traits for your types
- You can implement foreign traits for your types

“You either own the trait or own the type”

Traits

Default, Iterator

```rust
struct Fibonacci {
    curr: u128,
    next: u128,
}

impl Iterator for Fibonacci {
    type Item = u128;

    fn next(&mut self) -> Option<Self::Item> {
        let new_next = self.curr.checked_add(self.next)?;
        self.curr = self.next;
        self.next = new_next;
        Some(self.curr)
    }
}

impl Default for Fibonacci {
    fn default() -> Self {
        Self { curr: 0, next: 1 }
    }
}
```

Traits

Default, Iterator

```rust
fn main() {
    let fibb = Fibonacci { curr: 0, next: 1 };

    for (idx, i) in fibb.enumerate() {
        println!("{}: {}", idx, i)
    }

    for i in Fibonacci::default() {
        println("{}", i);
    }
}


fn test_fibonacci_vector() {
    let coll: Vec<u128> = Fibonacci::default().take(10).collect();
    assert_eq!(coll, vec![1, 1, 2, 3, 5, 8, 13, 21, 34, 55]);
}
```

Traits

You own the Trait

```
trait Print {
    fn print(&self);
}

impl<T: std::fmt::Debug> Print for T {
    fn print(&self) {
        println!("{:?}", self);
    }
}

"Hello, world".print();
vec![0, 1, 2, 3, 4].print();
"You get the idea".print()
```

Itertools in itertools - Rust

docs.rs/itertools/latest/itertools/trait.Itertools.html

DOCS.RS | itertools-0.10.3 | Platform | Feature flags | Releases | Rust | Find crate

Click or press 'S' to search, '?' for more options...

Trait Itertools

Provided Methods

- all_equal
- all_unique
- at_most_one
- batching
- cartesian_product
- chunks
- circular_tuple_windows
- coalesce
- collect_tuple
- collect_vec
- combinations
- combinations_with_repla...
- concat
- contains
- counts
- counts_by
- dedup
- dedup_by

Trait itertools::Itertools

```
pub trait Itertools: Iterator {
[+] Show 106 methods
}
```

[−] An `Iterator` blanket implementation that provides extra adaptors and methods.

This trait defines a number of methods. They are divided into two groups:

- *Adaptors* take an iterator and parameter as input, and return a new iterator value. These are listed first in the trait. An example of an adaptor is `.interleave()`
- *Regular methods* are those that don't return iterators and instead return a regular value of some other kind. `.next_tuple()` is an example and the first regular method in the list.

Provided methods

```
[−] fn interleave<J>(self, other: J) -> Interleave<Self, J::IntoIter> ⓘ [src]
where
    J: IntoIterator<Item = Self::Item>,
    Self: Sized,
```

Alternate elements from two iterators until both have run out.

Iterator element type is `Self::Item`.

This iterator is *fused*.

```
use itertools::Itertools;

let it = (1..7).interleave(vec![-1, -2]);
```

doc.rust-lang.org/nightly/core/iter/traits/.../trait.Iterator.html


```
pub trait Itertools: Iterator {
[+] Show 106 methods
}
```

Result and Option

Making impossible states impossible

- Tony Hoare’s Billion Dollar Mistake
- There is no null (and no equivalent) in Rust
- There are no uninitialised states
- Rust forces you to deal with potential errors or values that don’t exist

Result and Option

Making impossible states impossible

```rust
pub enum Option<T> {
    None,
    Some(T),
}
```

```rust
pub enum Result<T, E> {
    Err(E),
    Ok(T),
}
```

Result

Deal with it!

```rust
fn read_username_from_file(path: &str) -> Result<String, io::Error> {
    let f = File::open(path);

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(err) => Err(err),
    }
}
```

Result

Deal with it!

```rust
fn read_username_from_file(path: &str) -> Result<String, io::Error> {
    let f = File::open(path);

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(err) => Err(err),
    }
}
```

Unwrap the value on your own

Result

Deal with it!

```rust
fn read_username_from_file(path: &str) -> Result<String, io::Error> {
    let f = File::open(path);

    let mut f = match f {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut s = String::new();

    match f.read_to_string(&mut s) {
        Ok(_) => Ok(s),
        Err(err) => Err(err),
    }
}
```

Return the error

Unwrap the value on your own

Result

Ignore it

```rust
fn read_username_from_file(path: &str) -> Result<String, io::Error> {
    let mut f = File::open(path).unwrap();

    let mut s = String::new();

    f.read_to_string(&mut s).unwrap();

    Ok(s)
}
```

Result

Ignore it

```rust
fn read_username_from_file(path: &str) -> Result<String, io::Error> {
    let mut f = File::open(path).unwrap();

    let mut s = String::new();

    f.read_to_string(&mut s).unwrap();

    Ok(s)
}
```

unwrap — this panics if there's an error

SEARCH

unwrap()

Aa ab .*

files to include

./javascript-runtime

files to exclude

129 results in 15 files - Open in editor

Result

Propagate the error

- If your function returns `Result<T, E>`, why not deal with the errors one level above?
- Propagate the error with the question mark operator

```rust
fn read_username_from_file(path: &str) -> Result<String, io::Error> {
    let mut f = File::open(path)?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}

fn main() {
    match read_username_from_file("user.txt") {
        Ok(username) => println!("Welcome {}", username),
        Err(err) => eprintln!("Whoopsie! {}", err)
    };
}
```

Typestate

```rust
pub struct Worker {
    workload: String,
    memsize: u128,
    keep_alive: bool,
}

struct NoWorkload;

struct WorkerBuilder<W> {
    workload: W,
    memsize: u128,
    keep_alive: bool,
}
```

```rust
impl<W> WorkerBuilder<W> {
    pub fn memsize(&mut self, memsize: u128)
      -> &mut Self {
        self.memsize = memsize;
        self
    }

    pub fn keep_alive(&mut self, keepalive: bool)
      -> &mut Self {
        self.keep_alive = keepalive;
        self
    }
}
```

Typestate

```rust
impl WorkerBuilder<NoWorkload> {
    pub fn new() -> Self {
        Self {
            workload: NoWorkload,
            memsize: 128 * 1024,
            keep_alive: false,
        }
    }

    pub fn workload(
        &self,
        workload: impl Into<String>
    ) -> WorkerBuilder<String> {
        WorkerBuilder {
            workload: workload.into(),
            memsize: self.memsize,
            keep_alive: self.keep_alive,
        }
    }
}
```

```rust
impl WorkerBuilder<String> {
    pub fn build(&mut self) -> Worker {
        let workload = self.workload.clone();
        Worker {
            workload,
            memsize: self.memsize,
            keep_alive: self.keep_alive,
        }
    }
}
```

Fearless concurrency

Preconditions for data races

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There's no mechanism being used to synchronize access to the data

Ownership and thread safety

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Ownership and thread safety

```rust
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

ERROR

```
closure may outlive the current
function, but it borrows `v`,
which is owned by the current
function
```

Ownership and thread safety

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Here's a vector: {:?}", v);
    });

    handle.join().unwrap();
}
```

Ownership and thread safety

```rust
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

Ownership and thread safety

```rust
use std::sync::Mutex;
use std::thread;

fn main() {
    let counter = Mutex::new(0);
    let mut handles = vec![];

    for _ in 0..10 {
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

ERROR

```
use of moved value: `counter`
```

Ownership and thread safety

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
           *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

Rayon

```rust
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.iter() // <-- just change that!
         .map(|&i| i * i)
         .sum()
}
```

Rayon

```rust
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.iter() // <-- just change that!
         .map(|&i| i * i)
         .sum()
}
```

```rust
use rayon::prelude::*;
fn sum_of_squares(input: &[i32]) -> i32 {
    input.par_iter()
         .map(|&i| i * i)
         .sum()
}
```

... and more!

- Built-in multi-producer/single-consumer channels
- Sync + Send marker traits to extend concurrency
- Built-in primitives for green-thread concurrency, independent of run-times
- Rich ecosystem around the async runtime Tokio

And more!

Immutable by default

- Bindings are immutable by default
- The `mut` keyword allows for mutations

```
fn main() {
    let immutable_number = 10;
    println!("The number is {}", immutable_number);

    immutable_number = 11; // 💥 Error!

    let mut mutable_number = 10;
    mutable_number = 11;
}
```

- Bindings are block-scoped

Shadowing

- Re-declare bindings to create new bindings

```
// The first binding is created with the name "number"
let number = 5;

// A different binding shadows the name "number"
let number = number + 5;

// Again, another new binding is created
let number = number * 2;
```

- A shadowing technique to temporarily make bindings immutable

```
let mut number = 5;

{
    let number = number;
    // number is now frozen within this scope
}
// number here is mutable again
```

Control flow

Match expressions

- Pattern matching on types
- Needs to be exhaustive (otherwise the compiler errors)
- Works with ranges, union groups, enums
- Either concrete values, or catch-all branches with optional binding names

```
match i {
    0..=10 => "Still ok",
    15 => "Halfway through",
    _ => "No idea what to do with that"
}
```

```
match ch {
    'a' | 'e' | 'i' | 'o' | 'u' | 'l' | 'n' | 'r' | 's' | 't' => 1,
    'd' | 'g' => 2,
    'b' | 'c' | 'm' | 'p' => 3,
    'f' | 'h' | 'v' | 'w' | 'y' => 4,
    'k' => 5,
    'j' | 'x' => 8,
    'q' | 'z' => 10,
     _ => 0,
}
```

Expressions

- Rust is expression-heavy.
- Semi-colons denote a statement.
- Dropping the semi-colon returns a value

```
let msg = if num == 5 {
    "five"
 } else if num == 4 {
    "four"
 } else {
    "other"
 };
```

Tooling

- Dependency Management via Cargo
- Version and Toolchain Management via Rustup
- First-class listing with Clippy
- Formatting standards with Rustfmt
- Testable documentation, yes!!

Adopting Rust

What you will love

- Cargo + crates.io
- Rust Analyzer
- The community
- Expressions
- match
- Enums
- This lovely feeling of being empowered

What will be hard ...

- Fighting the borrow checker
- Its steep learning curve
- Letting go of OO thinking
- Finding the non-standard standard library

Help needed for idiomatic rust.

reddit.com/r/rust/comments/uzm857/help_needed_for_idiomatic...

Update

reddit Home Search Reddit Free

Help needed for idiomatic rust. Trying inheritance like

Close

r/rust · Posted by u/hunter714 7 hours ago

1

Help needed for idiomatic rust. Trying inheritance like

Hi guys,

I need a bit of direction. I'm trying my hand at rust by making a small tauri app to draw ascii art shapes. Comming from the java world, I'm quite biased toward OOP and a bit lost with how to approch this using idiomatic rust while minimising verbosity and code duplication.

A shape is basicly described by a start and end position (x,y), a type (how to draw the shape between the two points) and it's z-index position (for shape selection and handling superposed shapes).

The basic description of the program flow is this :- user choose wich shape to draw

- User clic on the canvas
  - A new shape object is created with the mouse position .
  - The shape is saved in memory (a collection)
  - The shape contains a Array2D<char> representing the matrix of ascii chars to display and is filled using the specific draw implementation for the selected shape
  - The stored shapes are itered upon and an output string is sent back to the client
- User drag the mouse
  - The last shape in memory is updated (end point)
  - The stored shapes are itered upon and an output string is sent back to the client
- User release the mouse
  - The last shape in memory is updated (end point)
  - The stored shapes are itered upon and an output string is sent back to the client

...we were able to optimise away 700 CPU and 300 GB of memory

Tenable - Change from Node.js to Rust

...while the Java server could use up to 5GB of RAM, the comparable Rust server only used 50MB.

Tilde - Rust Adoption white paper

It’s true: I’ve never written more robust code!

A Rust Summit attendee

Rust enabled me to do things I didn’t think I was able to do before

A bettercode.eu attendee

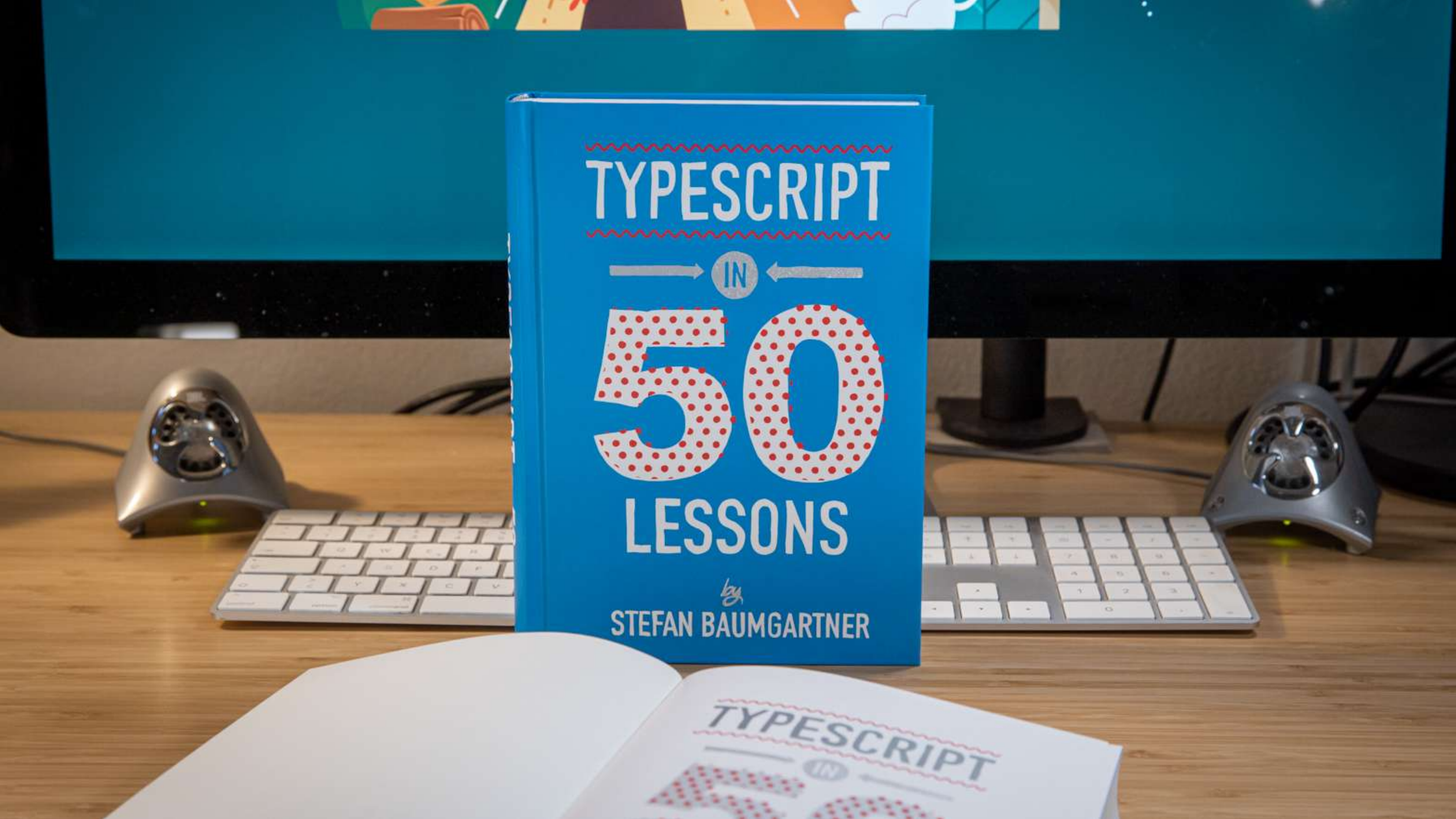
rust-linz.at

rust-training.eu

oida.dev

fettblog.eu


@ddprrt


TYPESCRIPT
IN
50
LESSONS
by
STEFAN BAUMGARTNER