

Ivan Ukhov
December 2014

Inspired by
Niko Matsakis

What is Rust?

“A systems programming language that runs blazingly fast, prevents almost all crashes,* and eliminates data races.”

* In theory. Rust is a work-in-progress and may do anything it likes up to and including eating your laundry.

What is Rust?



Why Rust?

Control

Safety

Why Rust?

C++

Control

Safety

Why Rust?

C++

Haskell

Control

Safety

Why Rust?

C++

Java

Ruby

Haskell

Control

Safety

Why Rust?

Rust

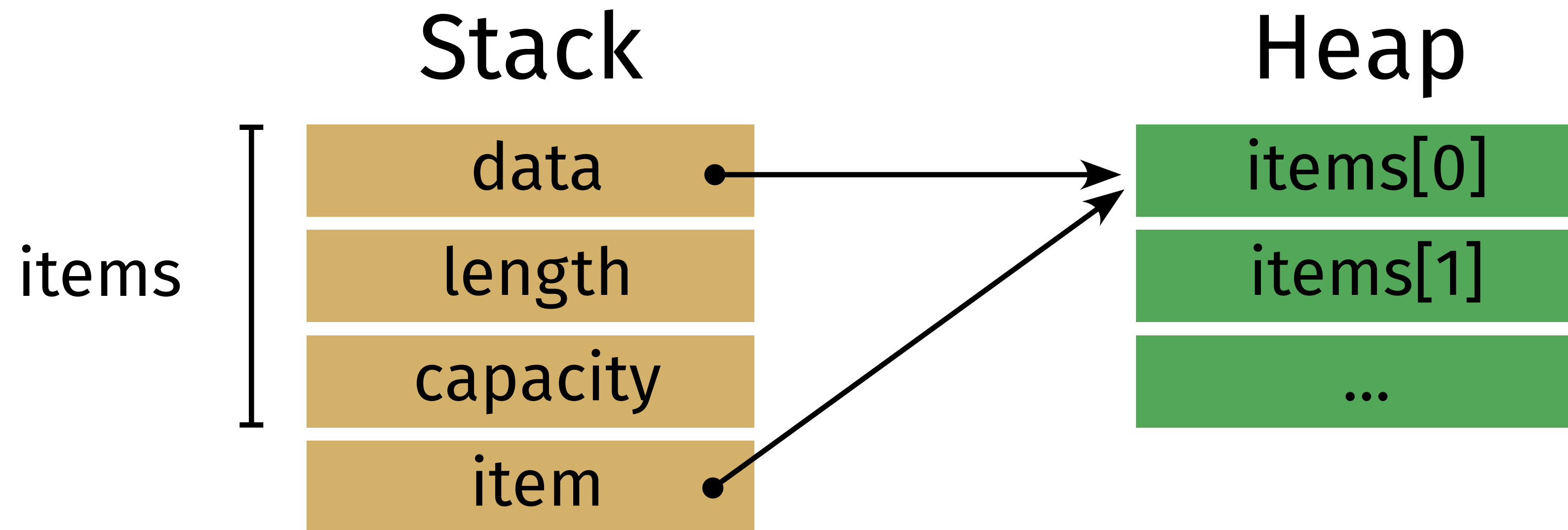
Control & Safety

What is control?

```
void foo() {  
    vector<string> items;  
    ...  
    auto& item = items[0];  
    ...  
}
```

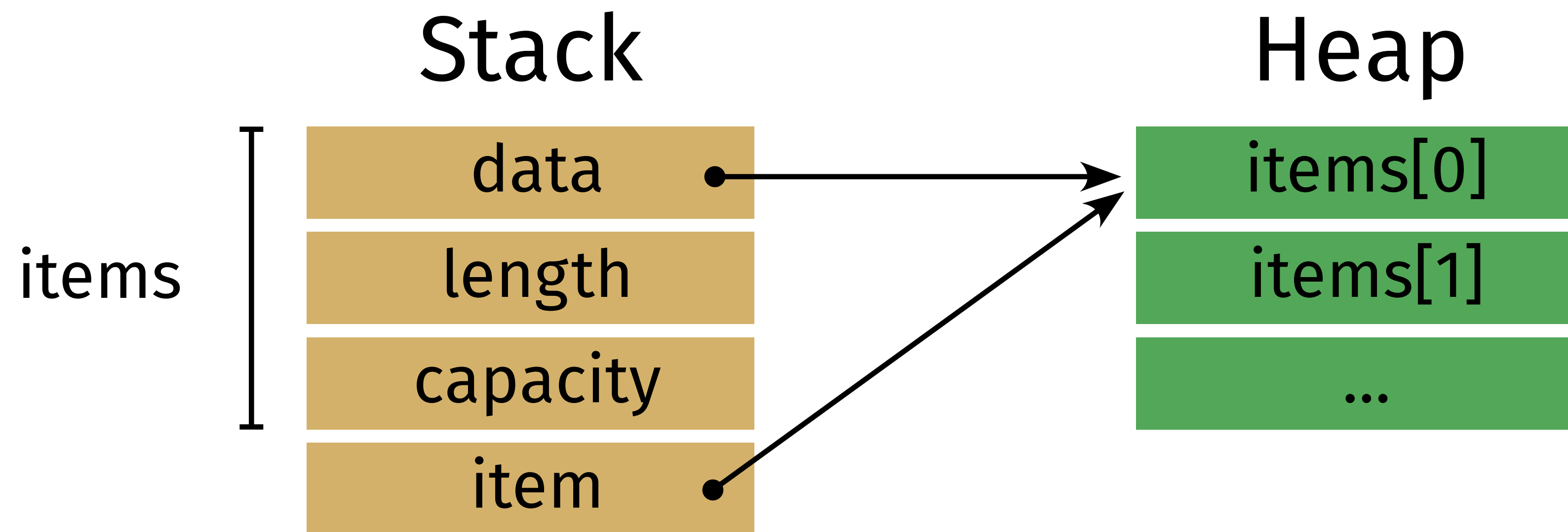
What is control?

```
void foo() {  
    vector<string> items;  
    ...  
    auto& item = items[0];  
    ...  
}
```



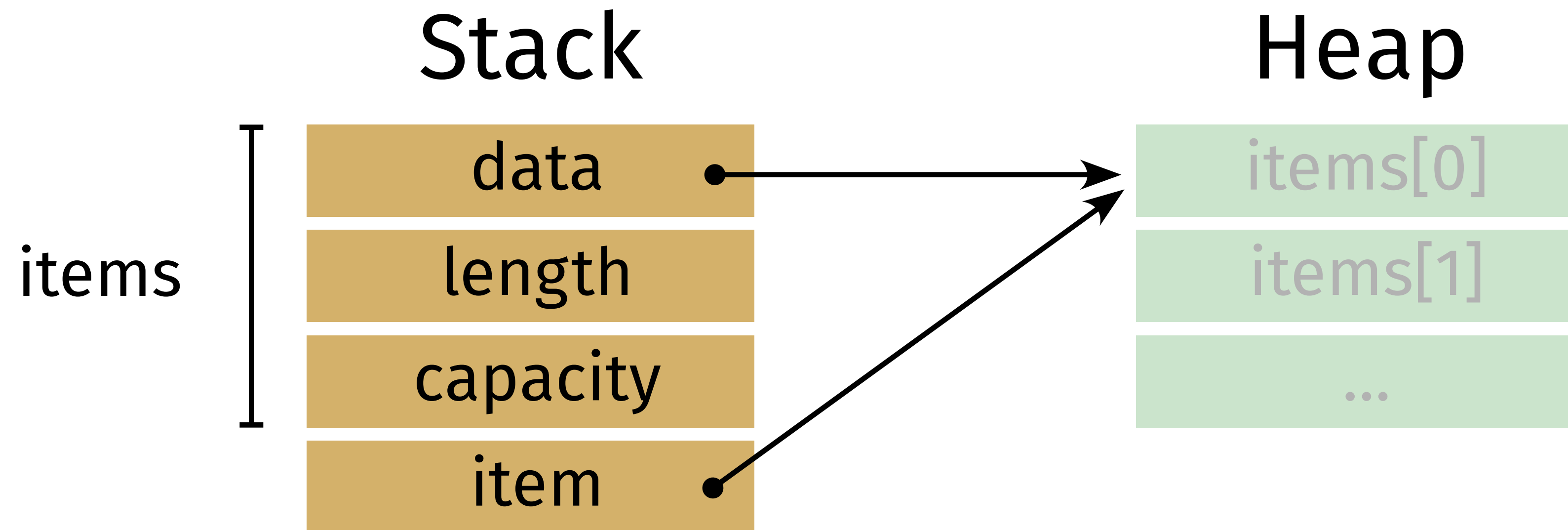
What is control?

```
void foo() {  
    vector<string> items;  
    ...  
    auto& item = items[0];  
    ...  
}
```



What is control?

```
void foo() {  
    vector<string> items;  
    ...  
    auto& item = items[0];  
    ...  
}
```

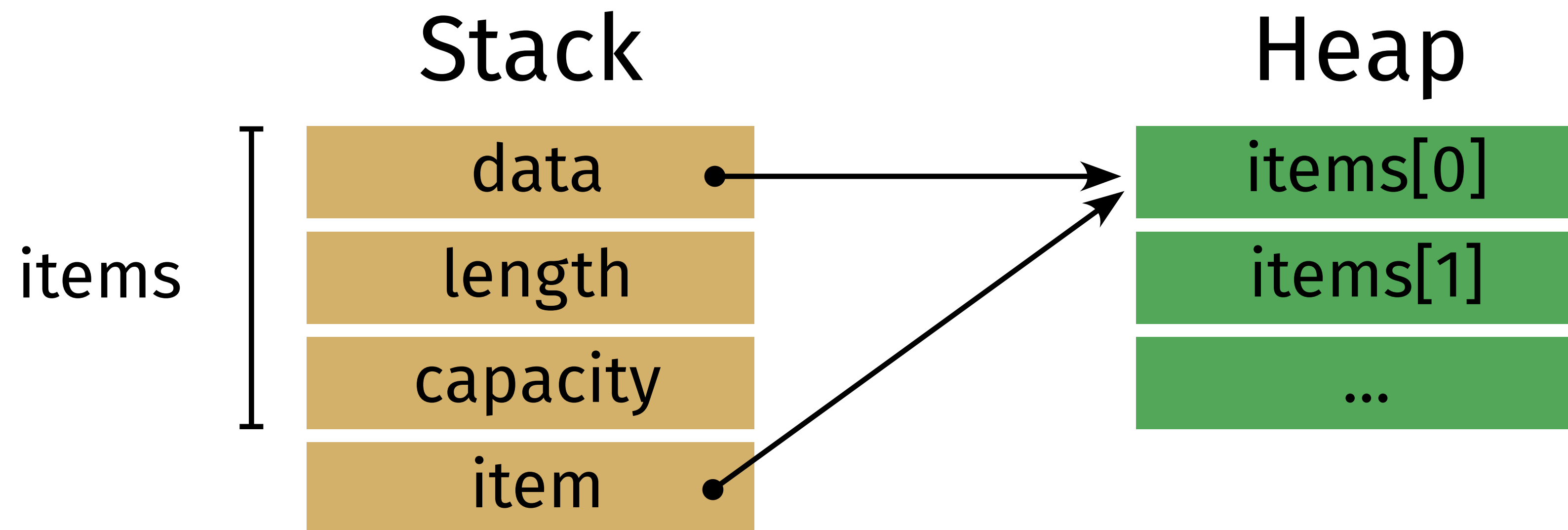


What is control?

- Zero-cost abstractions

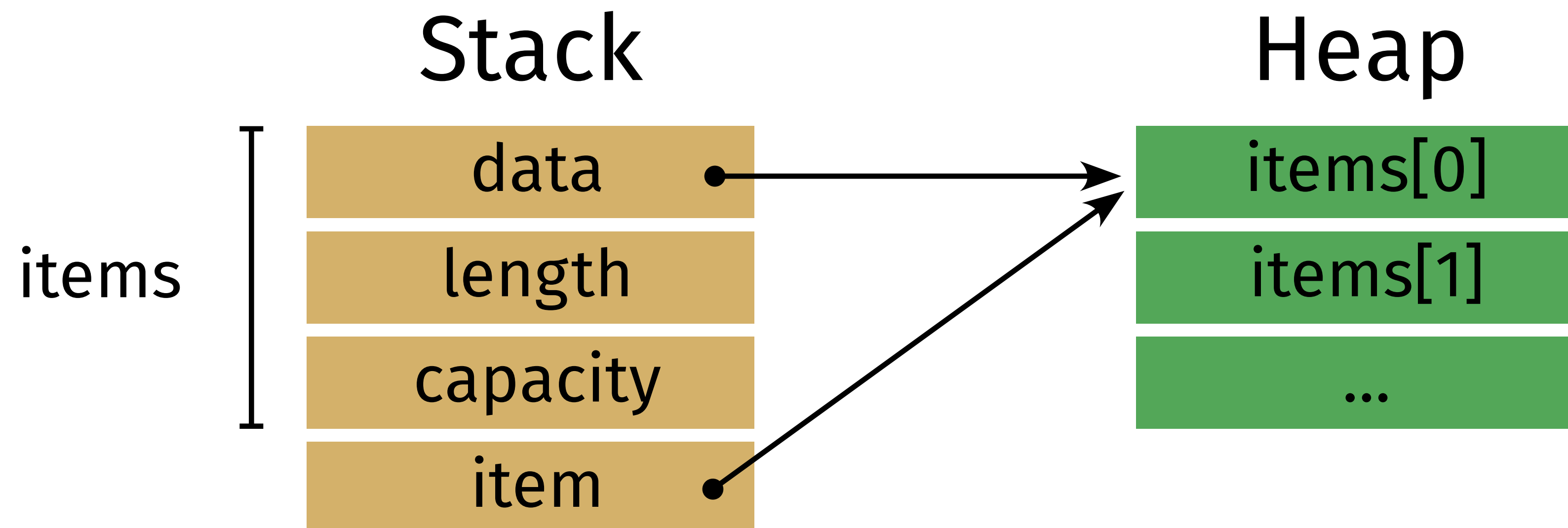
What is safety?

```
void foo() {  
    vector<string> items;  
    ...  
    auto& item = items[0];  
    ...  
}
```



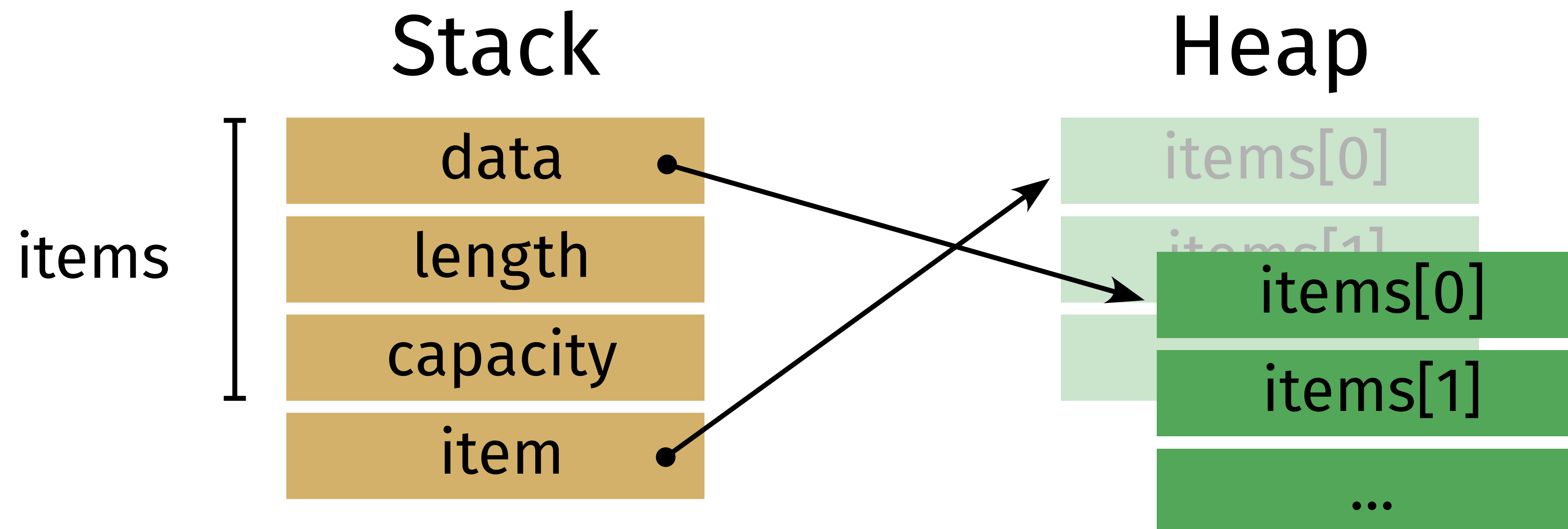
What is safety?

```
void foo() {  
    vector<string> items;  
    ...  
    auto& item = items[0];  
    items.push_back(...);  
    ...  
    use(item);  
}
```



What is safety?

```
void foo() {  
    vector<string> items;  
  
    ...  
    auto& item = items[0];  
    items.push_back(...);  
  
    ...  
    use(item);  
}
```



What is safety?

- No crashes
- No undefined behaviors

What about GC?

- No control

What about C++?

```
void want_to_read(const Foo& foo) { ... }
```

```
void want_to_write(Foo& foo) { ... }
```

```
void want_to_gut(Foo&& foo) { ... }
```

```
void want_to_take(unique_ptr<Foo> foo) { ... }
```

What about C++?

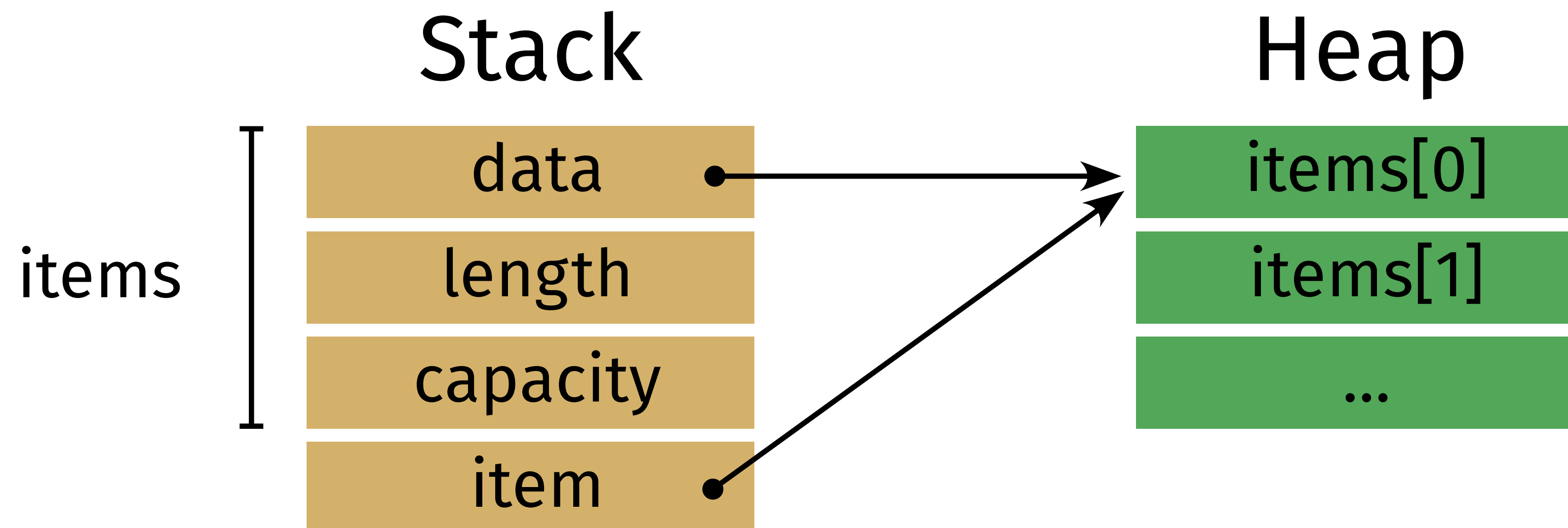
- Not safe
- Conventions unenforced

Solution

- Codify and enforce safe patterns

How to be safe?

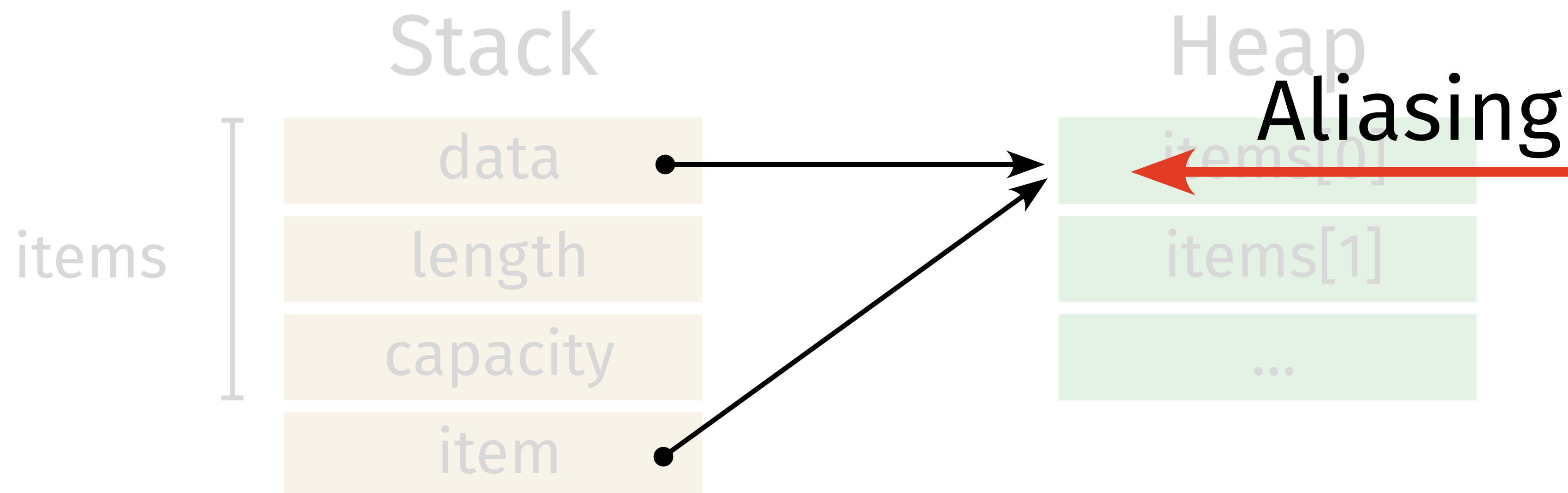
```
void foo() {  
    vector<string> items;  
  
    ...  
    auto& item = items[0];  
    items.push_back(...);  
  
    ...  
    use(item);  
}
```



How to be safe?

```
void foo() {  
    vector<string> items;  
    ...  
    auto& item = items[0];  
    items.push_back(...);  
    ...  
    use(item);  
}
```

Mutation



How to be safe?

Either or neither:

- Aliasing
- Mutation

Basic patterns

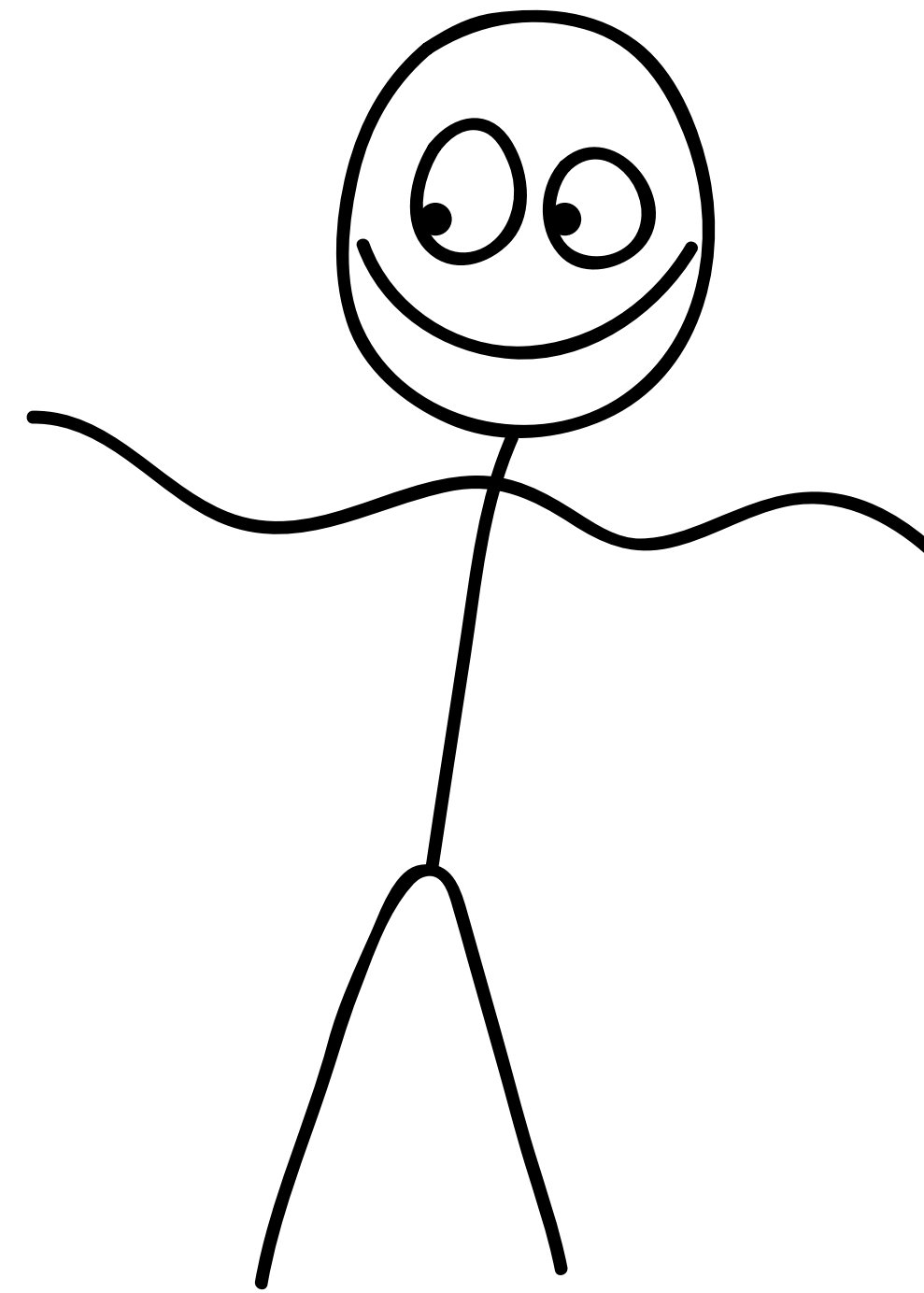
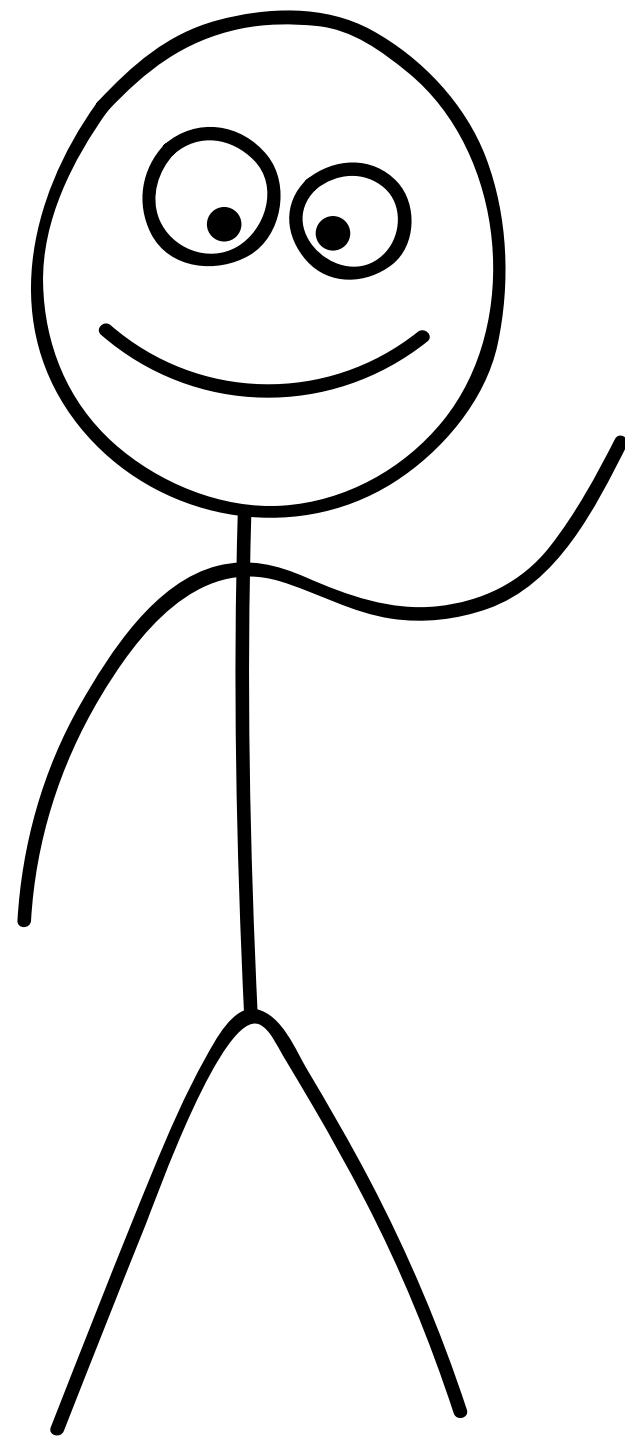
- Ownership
- Shared borrow
- Mutable borrow

Ownership

```
fn want_to_own(foo: Foo) { ... }
```

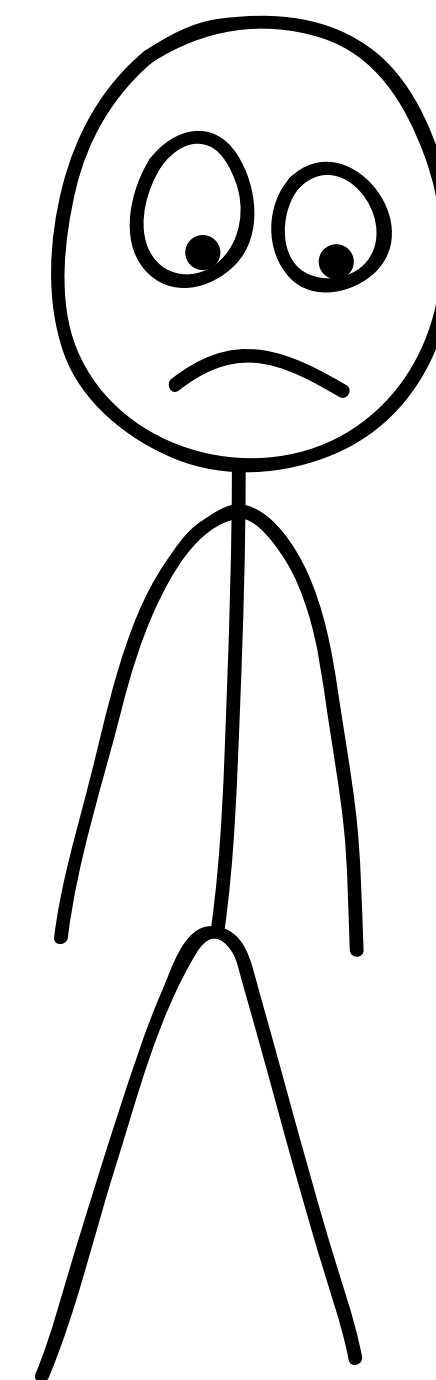
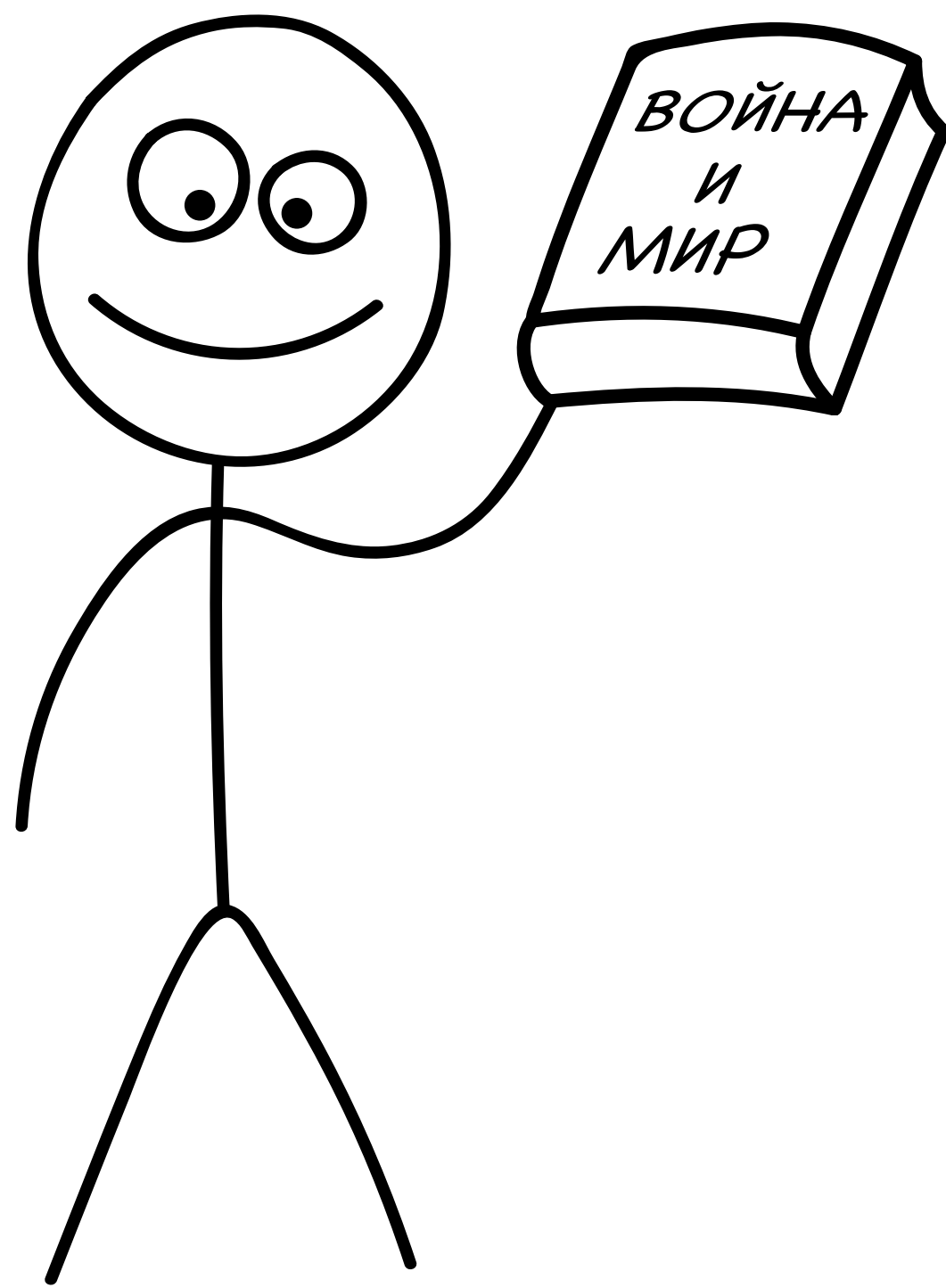
Ownership

```
fn want_to_own(foo: Foo) { ... }
```



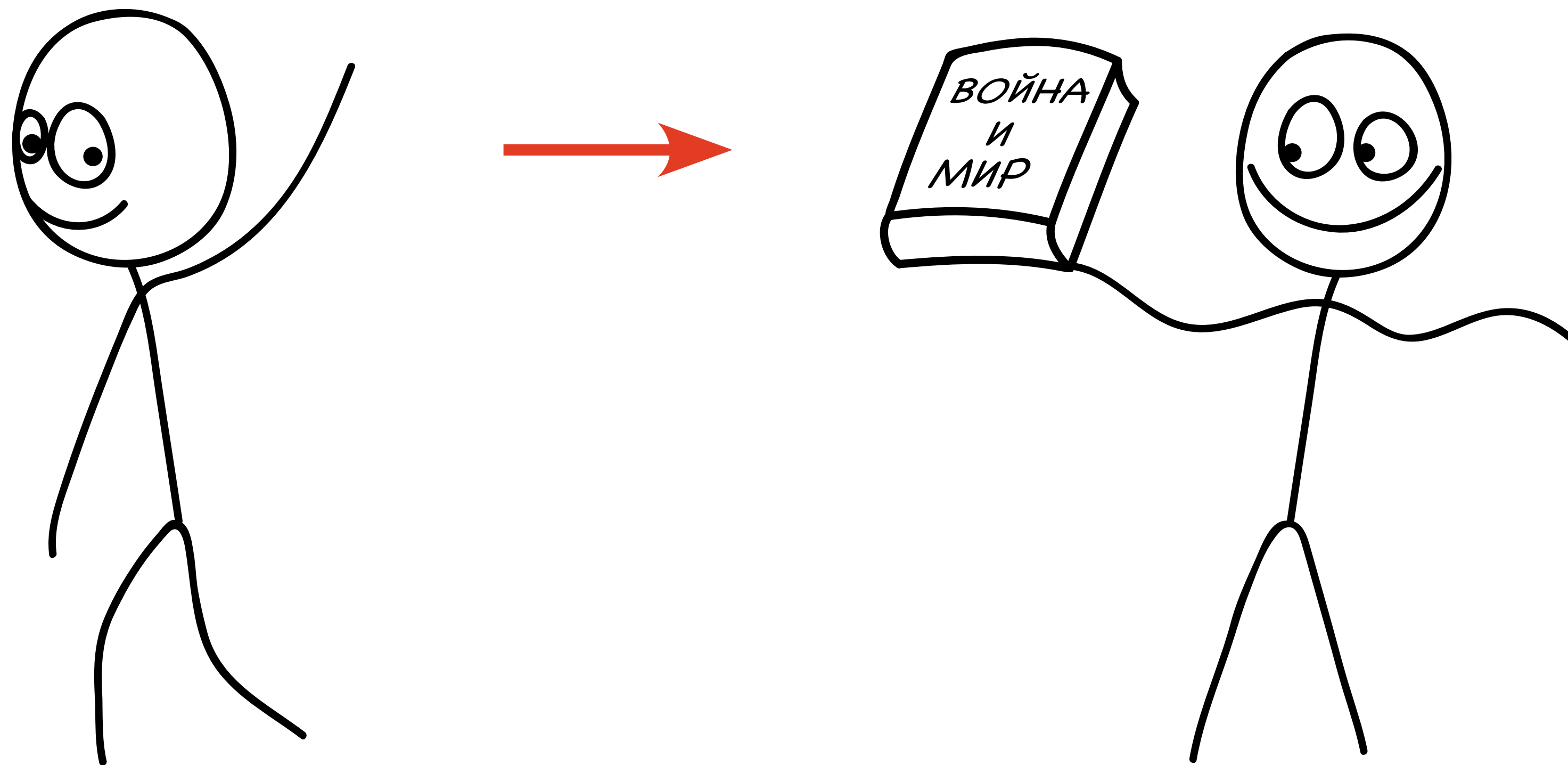
Ownership

```
fn want_to_own(foo: Foo) { ... }
```



Ownership

```
fn want_to_own(foo: Foo) { ... }
```



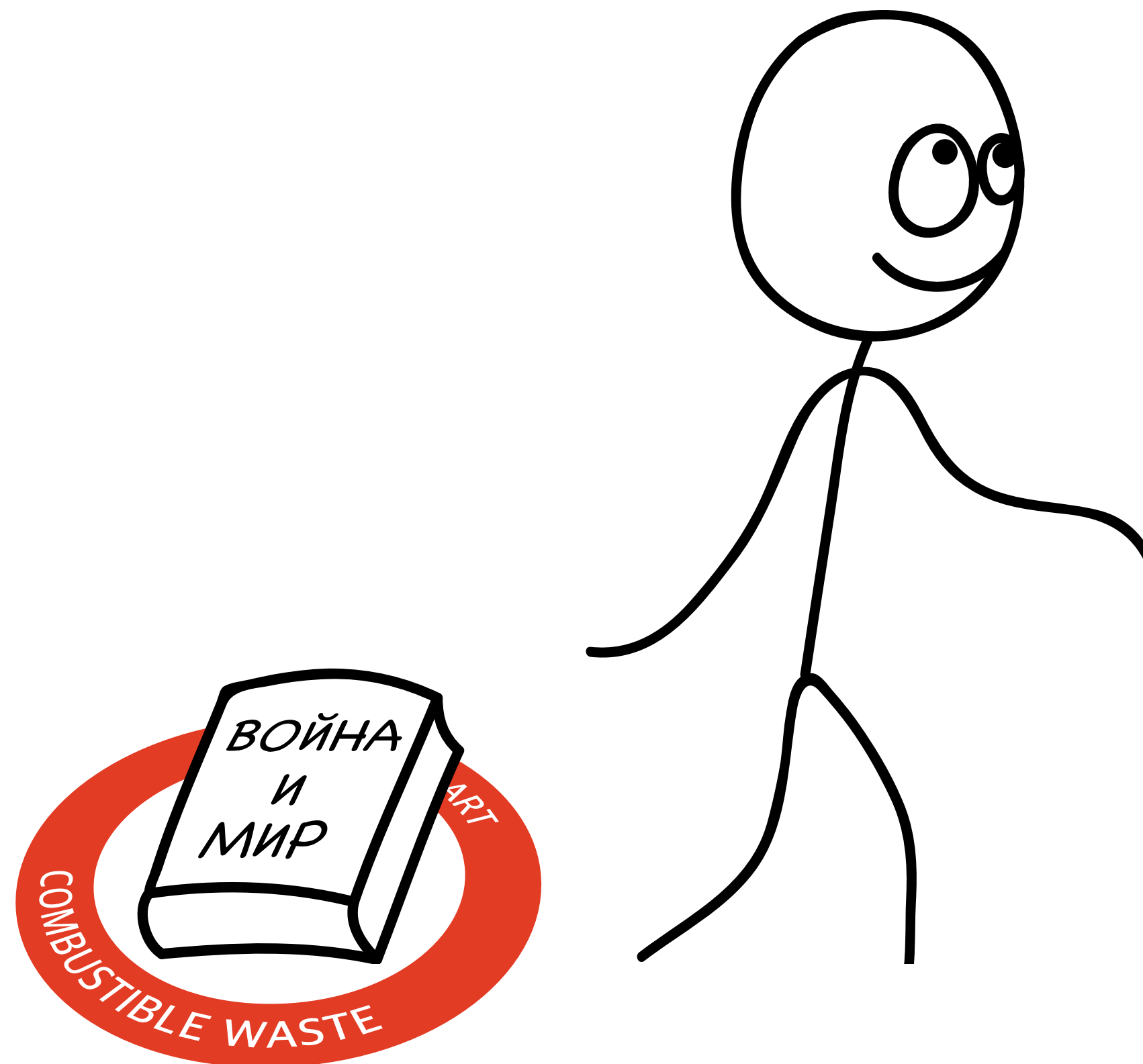
Ownership

```
fn want_to_own(foo: Foo) { ... }
```



Ownership

```
fn want_to_own(foo: Foo) { ... }
```




Ownership

```
fn save(data: Vec<u8>) {  
    let mut file = File::create(...);  
    file.write(data.as_slice());  
}  
  
fn main() {  
    let data = vec![1, 2, 3];  
    save(data);  
}
```


Ownership

```
fn save(data: Vec<u8>) {  
    let mut file = File::create(...);  
    file.write(data.as_slice());  
}
```



Destruction

```
fn main() {  
    let data = vec![1, 2, 3];  
    save(data);  
}
```

Ownership

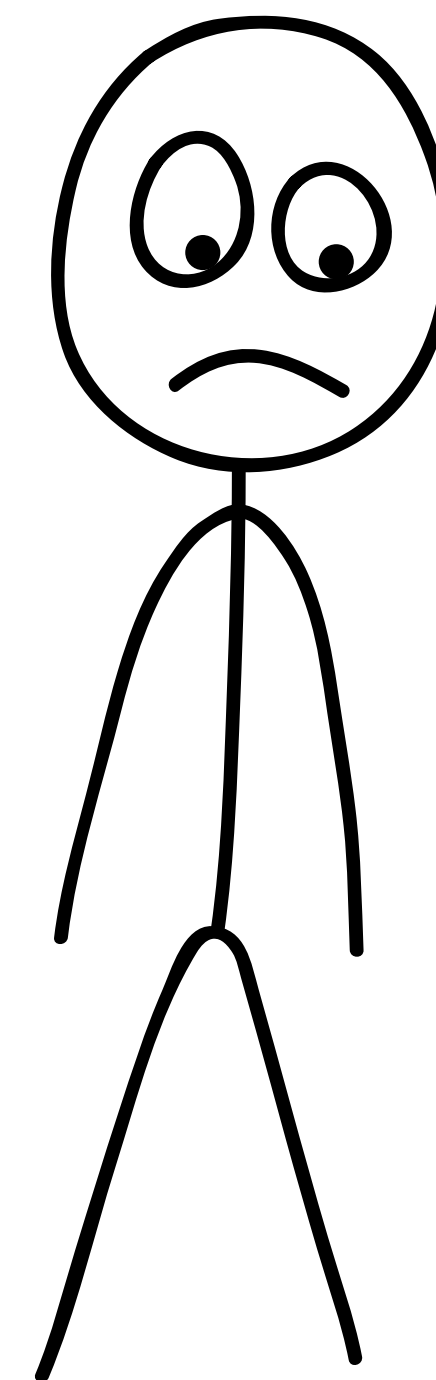
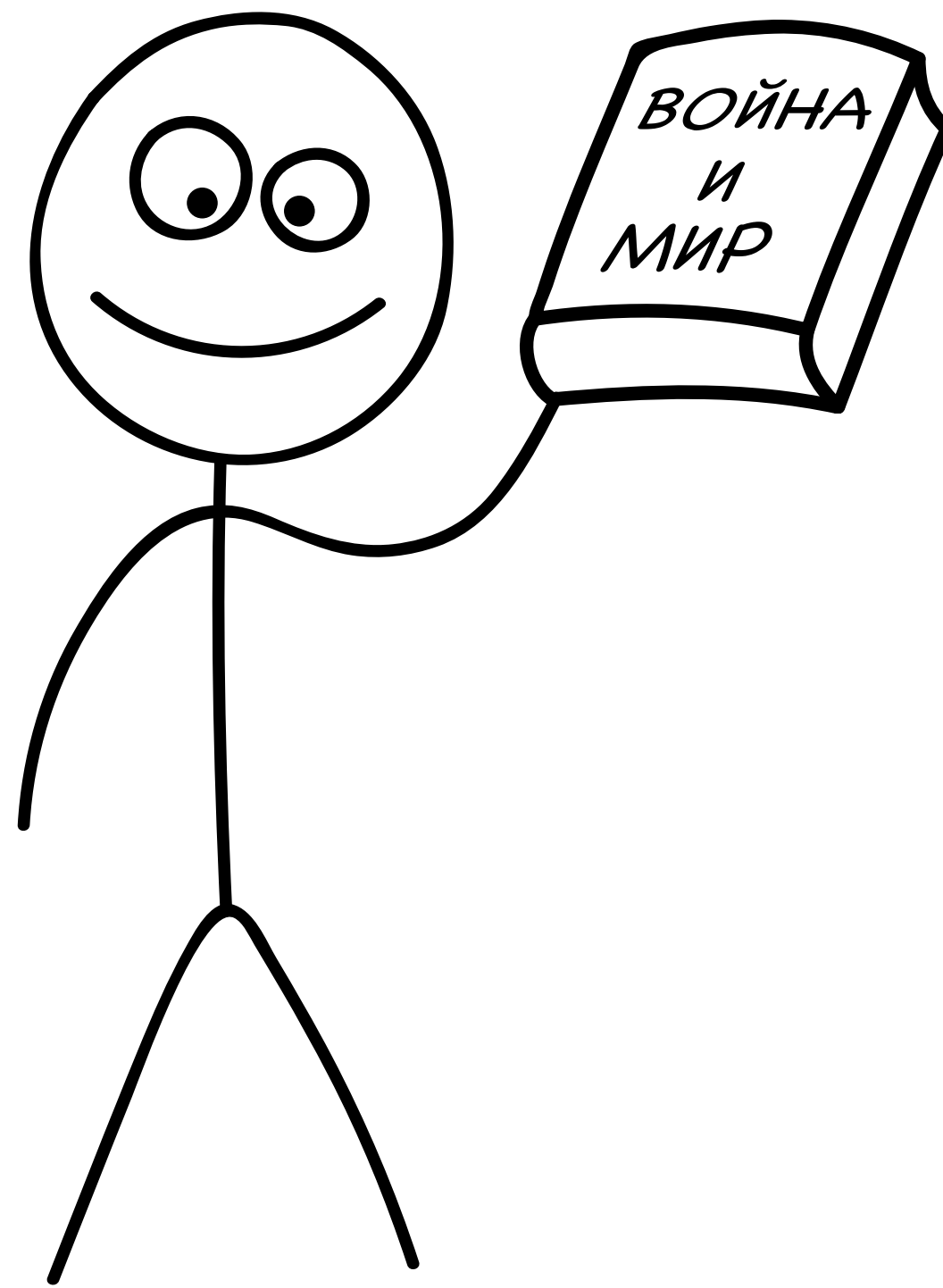
```
fn save(data: Vec<u8>) {  
    let mut file = File::create(...);  
    file.write(data.as_slice());  
}
```

```
fn main() {  
    let data = vec![1, 2, 3];  
    save(data);  
    println!("{}", data);  
}
```

← Compilation error

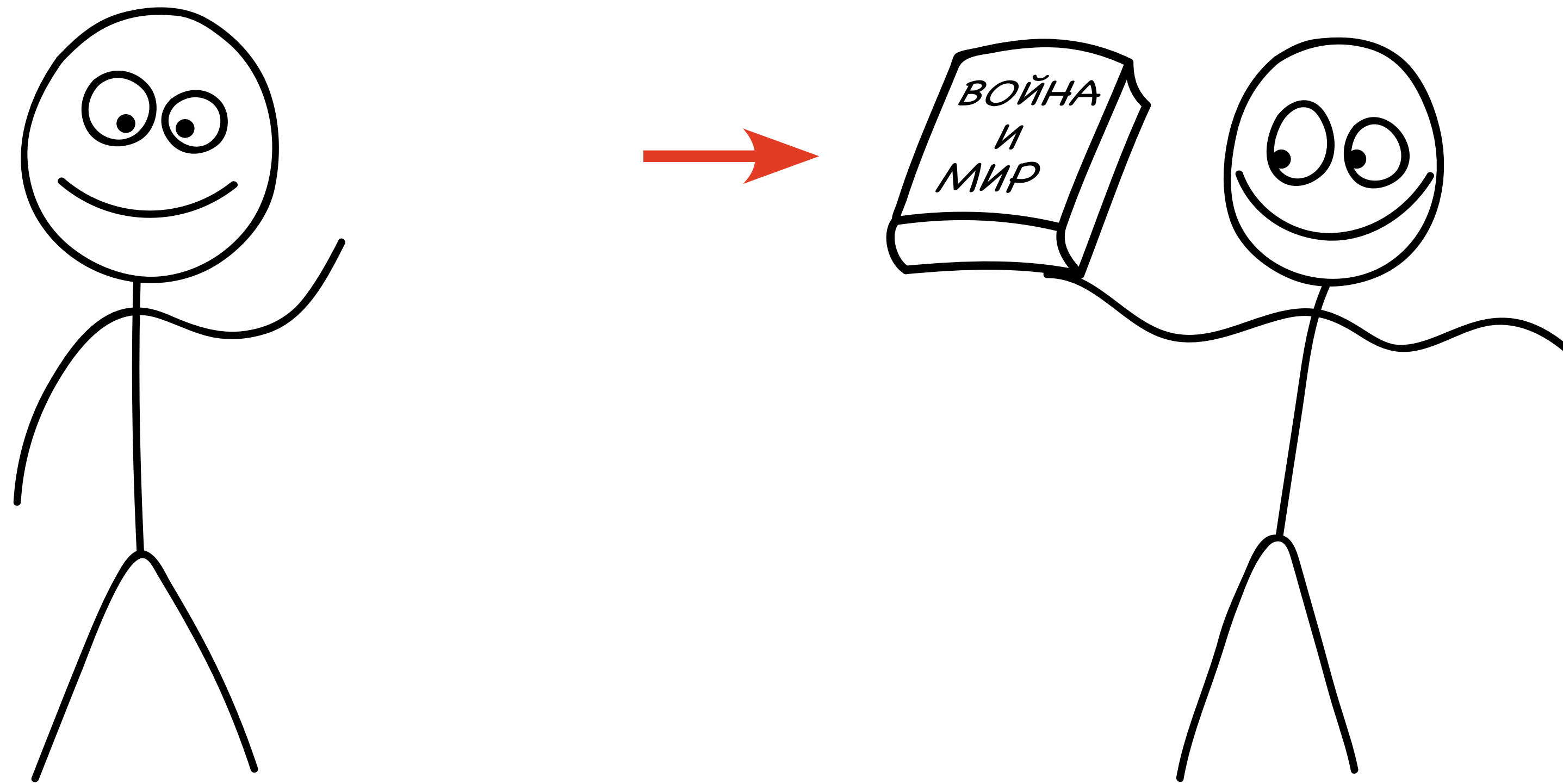
Shared borrow

```
fn want_to_borrow(foo: &Foo) { ... }
```



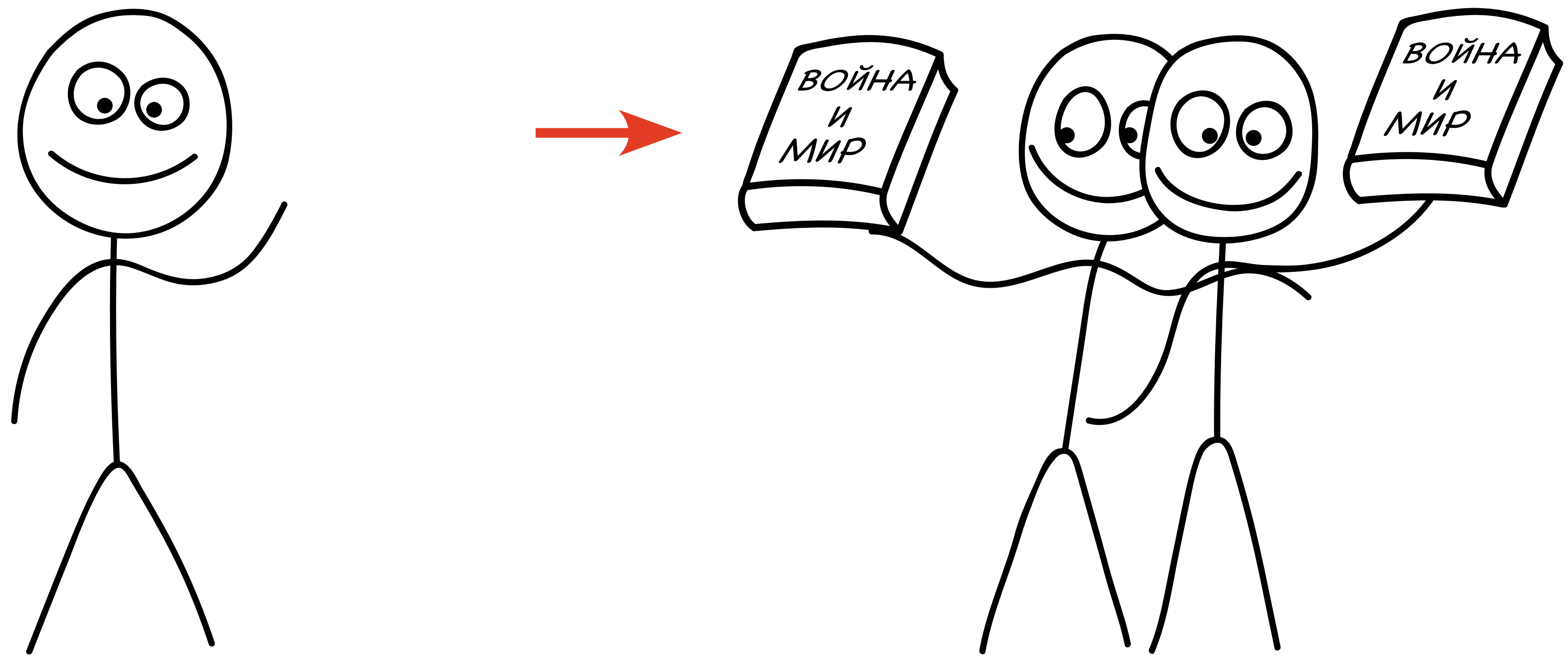
Shared borrow

```
fn want_to_borrow(foo: &Foo) { ... }
```



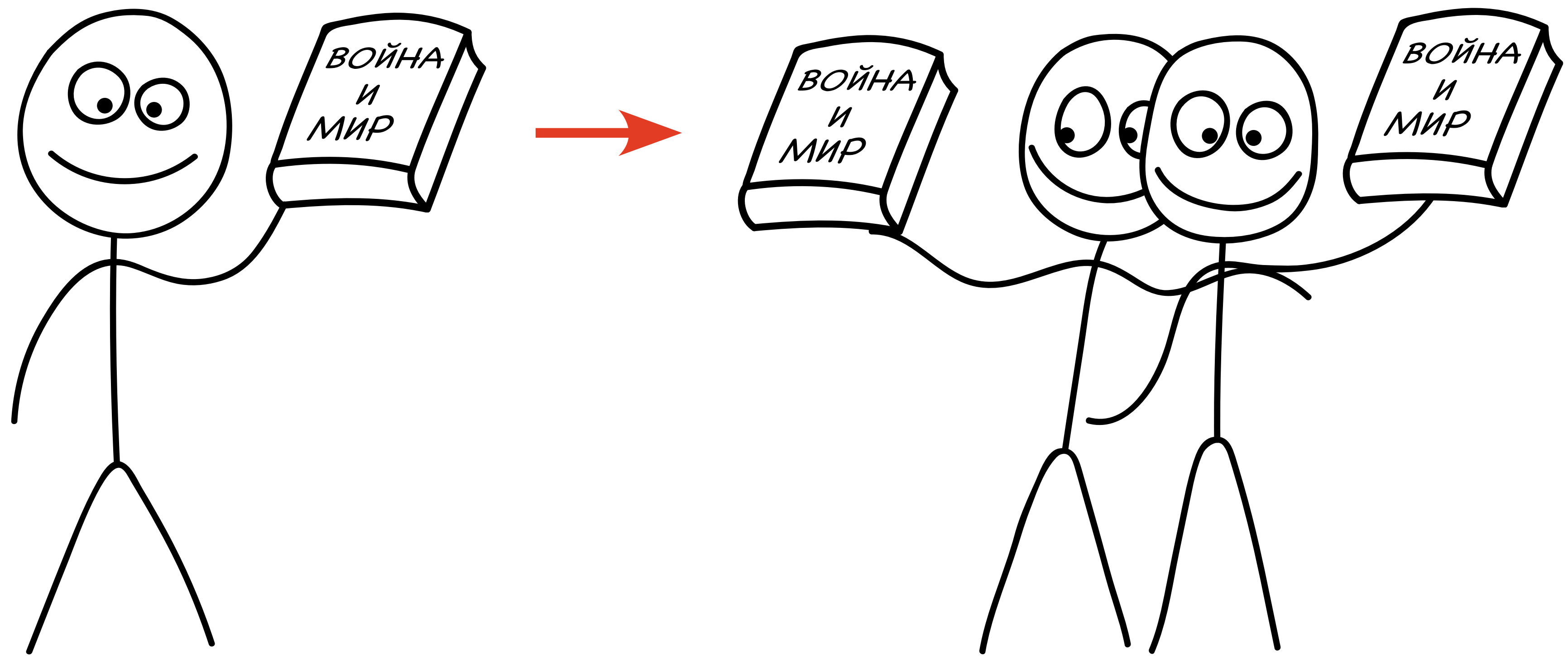
Shared borrow

```
fn want_to_borrow(foo: &Foo) { ... }
```



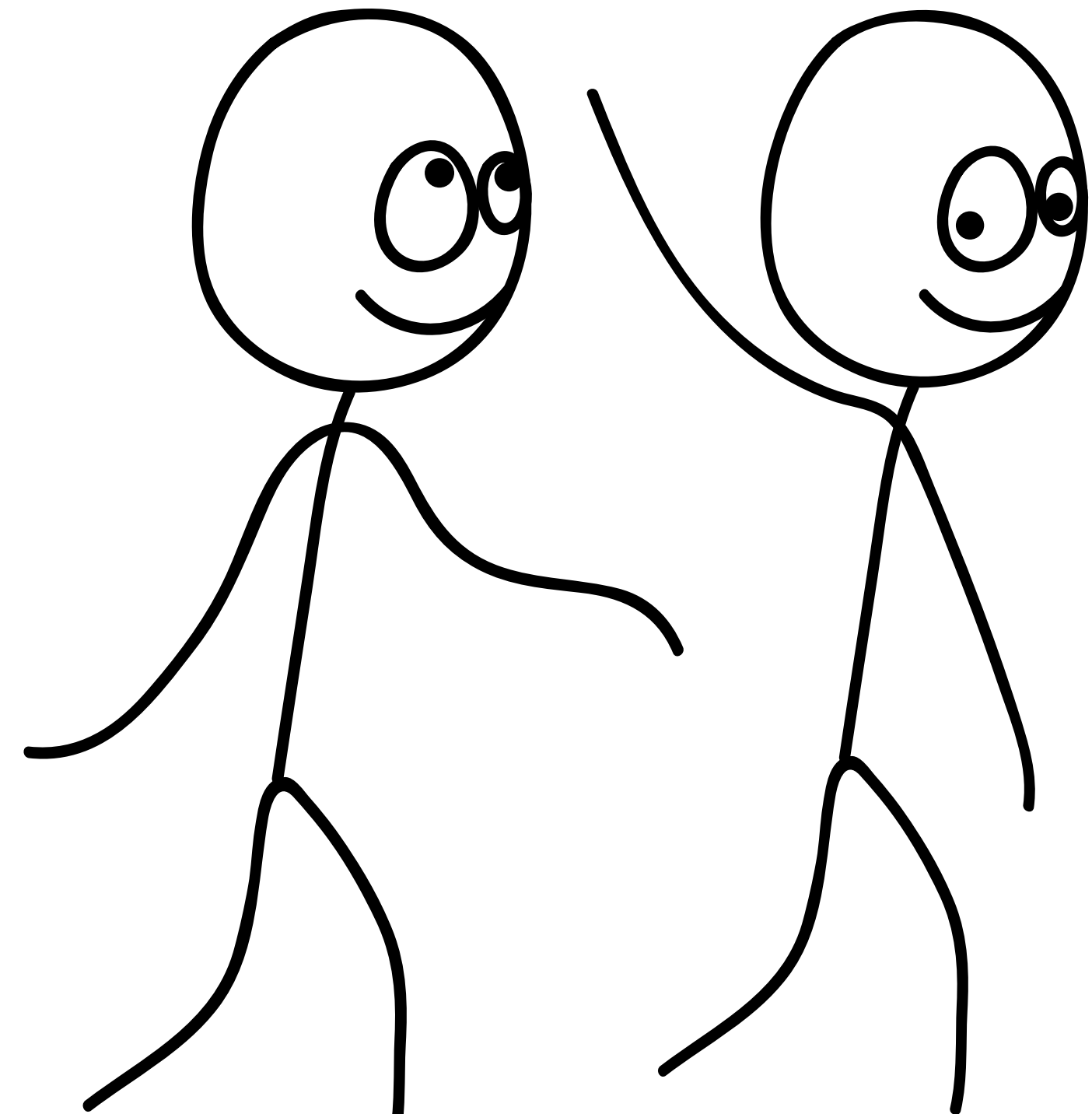
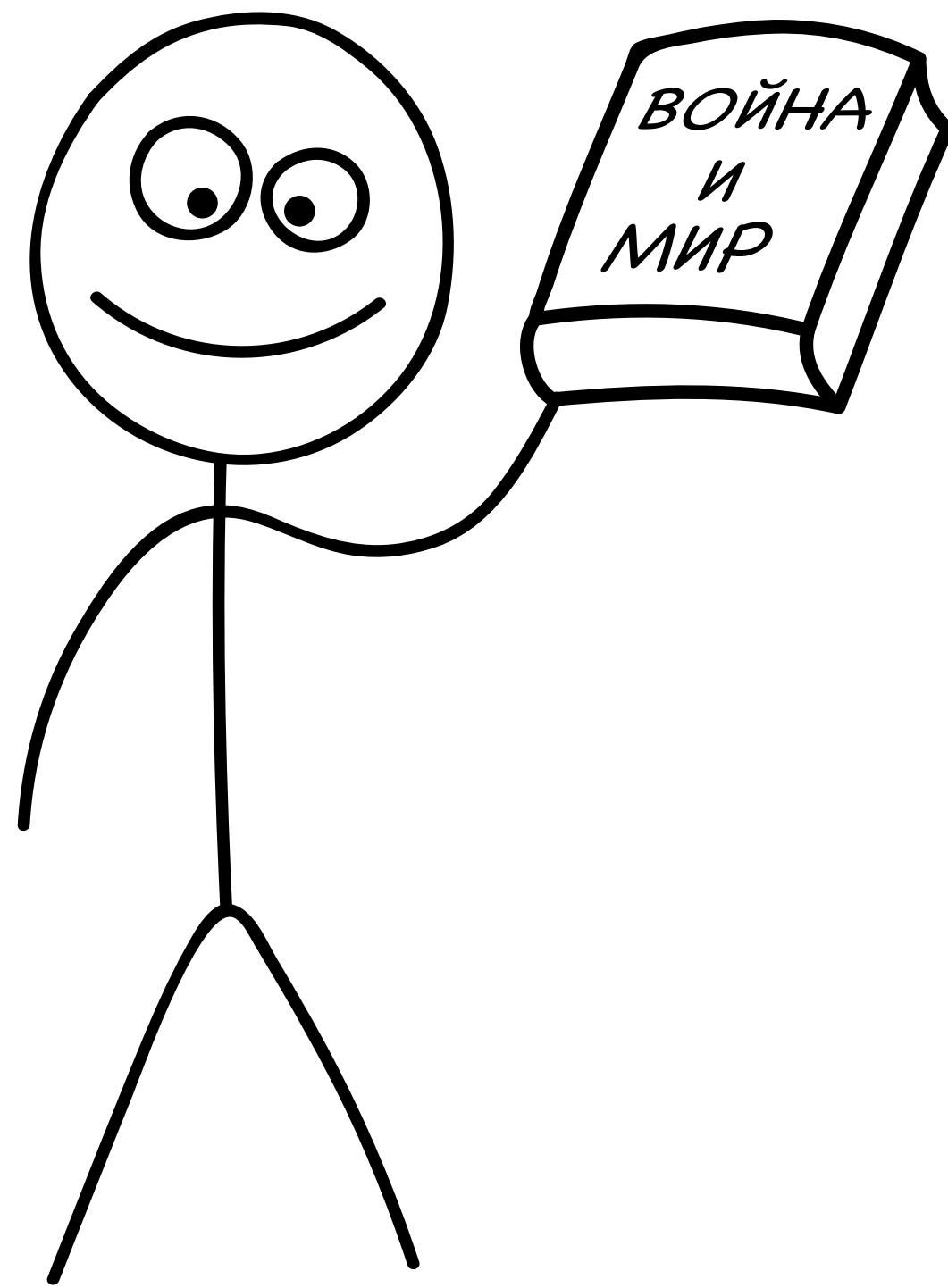
Shared borrow

```
fn want_to_borrow(foo: &Foo) { ... }
```



Shared borrow

```
fn want_to_borrow(foo: &Foo) { ... }
```



Shared borrow

```
fn want_to_borrow(foo: &Foo) { ... }
```




Shared borrow

```
fn sum_prod(one: &Vec<int>, two: &Vec<int>) -> int {  
    let mut result = 0;  
    for (x, y) in one.iter().zip(two.iter()) {  
        result += (*x) * (*y);  
    }  
    result  
}
```

```
fn main() {  
    let one = vec![1, 2, 3];  
    let two = vec![4, 5, 6];  
    let result = sum_prod(&one, &two);  
}
```

Shared borrow


```
fn sum_prod(one: &Vec<int>, two: &Vec<int>) -> int {  
    let mut result = 0;  
    for (x, y) in one.iter().zip(two.iter()) {  
        result += (*x) * (*y);  
    }  
    result  
}
```

```
fn main() {  
    let one = vec![1, 2, 3];  
    let result = sum_prod(&one, &one);   
}
```

Shared borrow

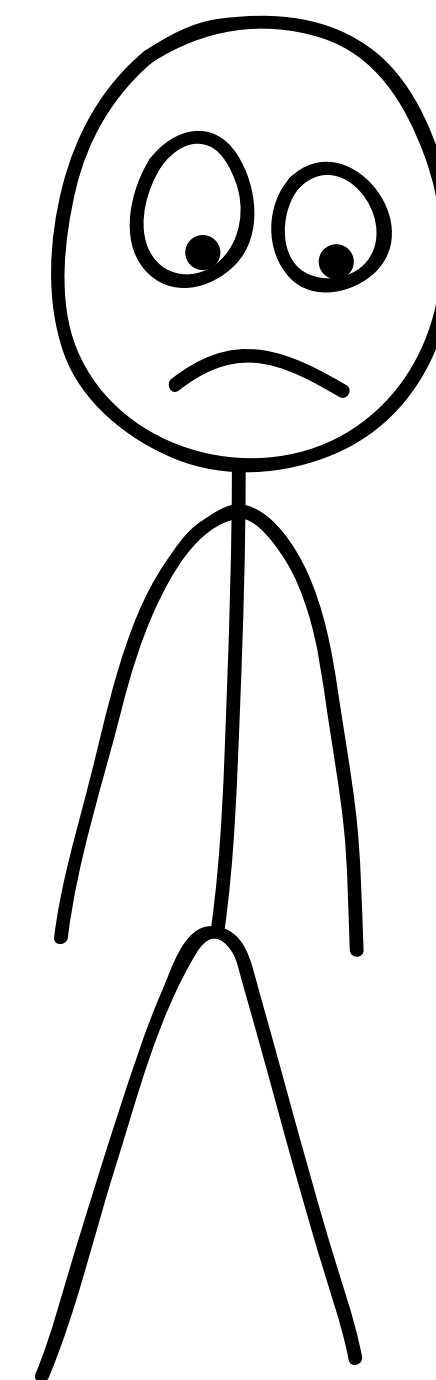
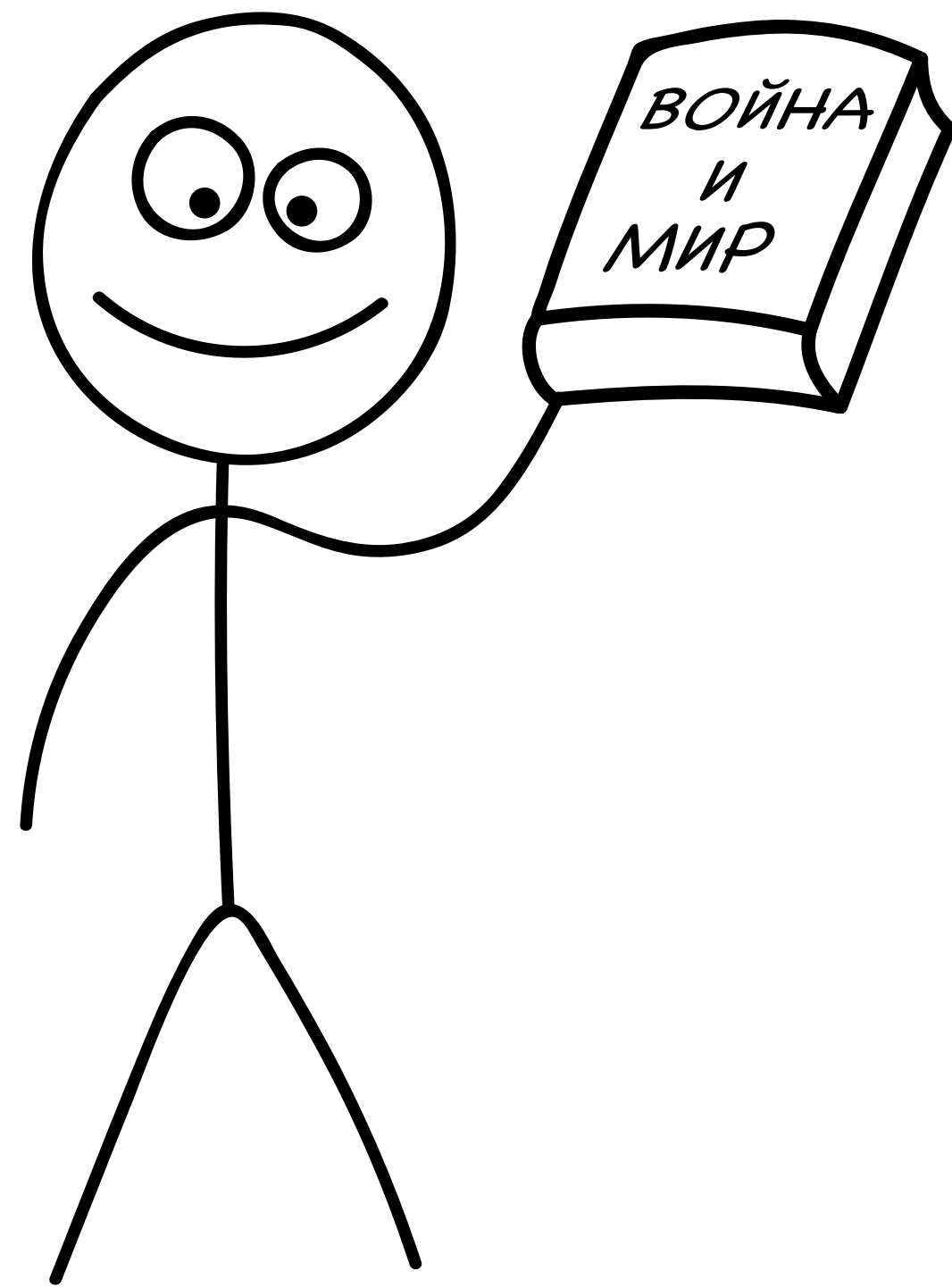
```
fn sum_prod(one: &Vec<int>, two: &Vec<int>) -> int {  
    let mut result = 0;  
    for (x, y) in one.iter().zip(two.iter()) {  
        result += (*x) * (*y);  
    }  
    result  
}
```

```
fn main() {  
    let one = vec![1, 2, 3];  
    let result = sum_prod(&one, &one);  
    println!("{}", one);  
}
```

 OK

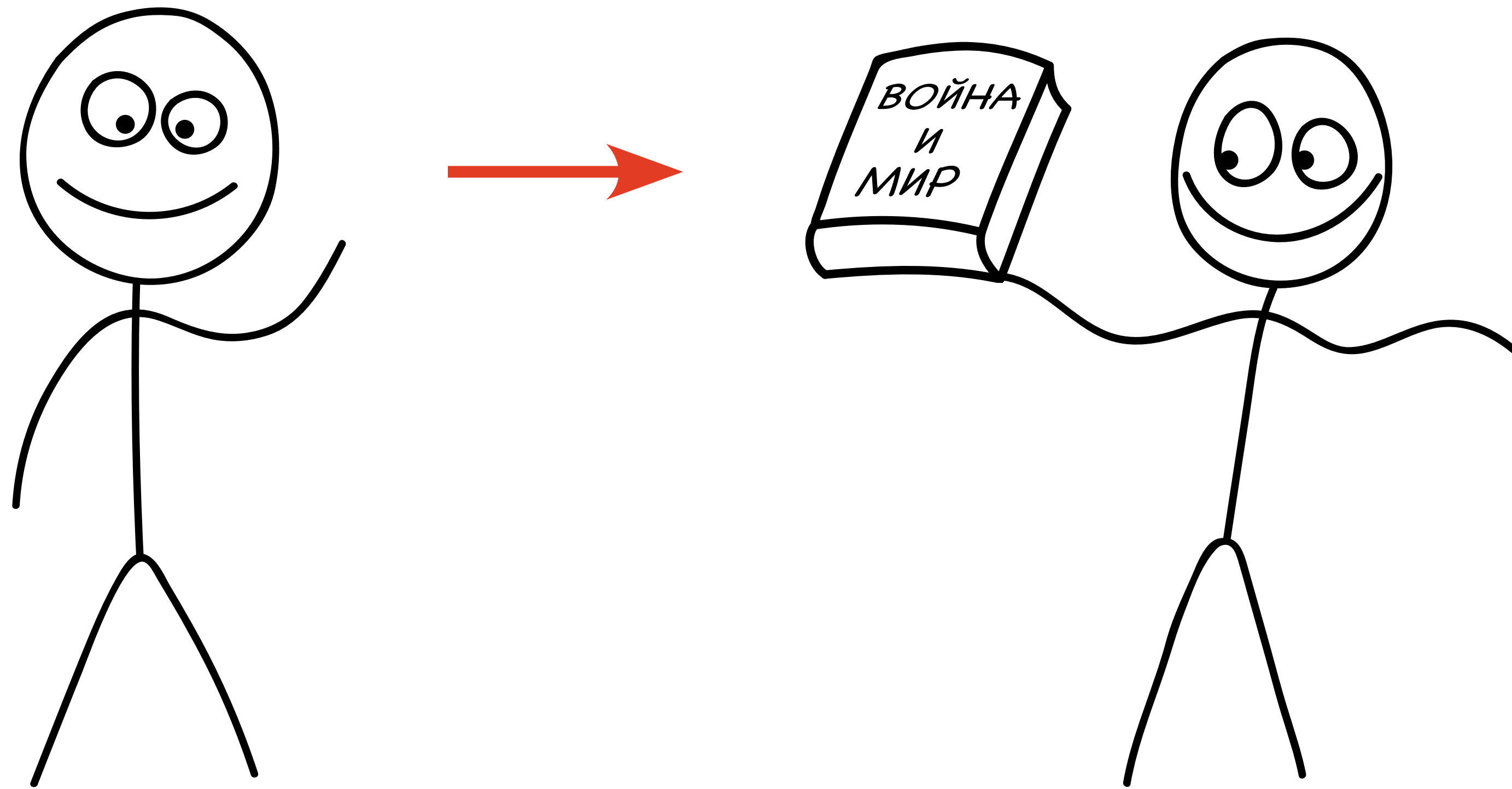
Mutable borrow

```
fn want_to_borrow(foo: &mut Foo) { .. }
```



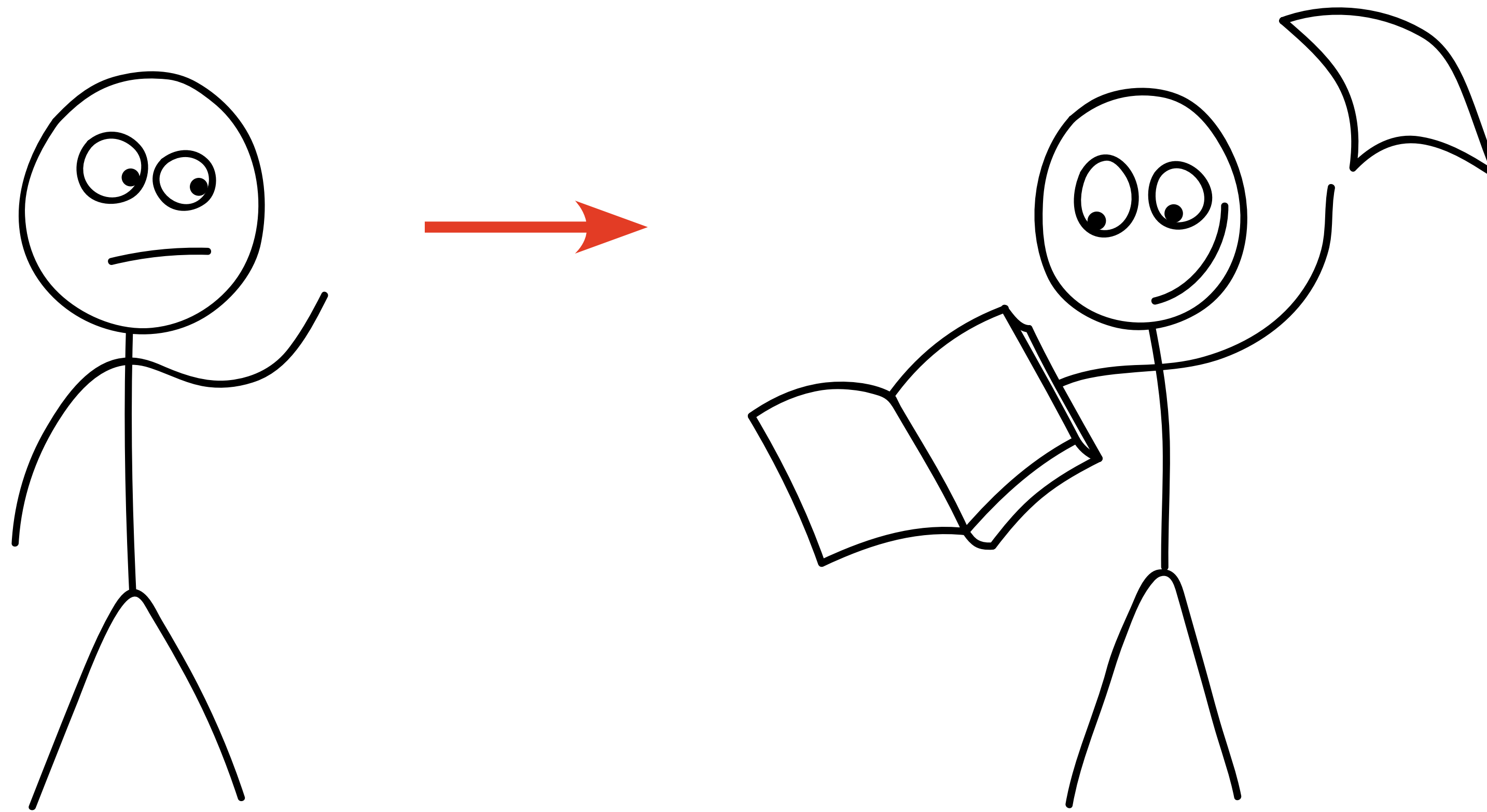
Mutable borrow

```
fn want_to_borrow(foo: &mut Foo) { .. }
```



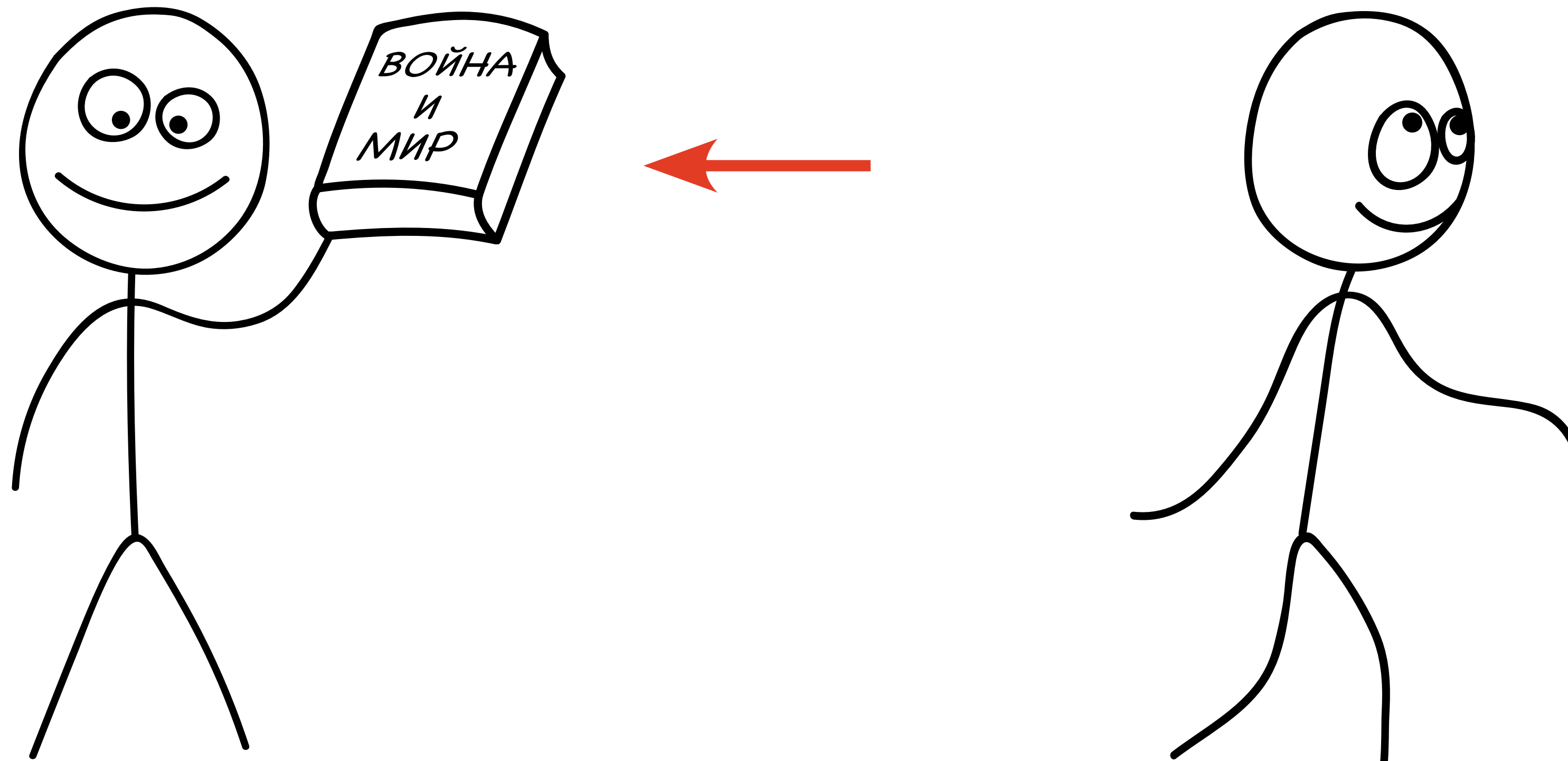
Mutable borrow

```
fn want_to_borrow(foo: &mut Foo) { .. }
```



Mutable borrow

```
fn want_to_borrow(foo: &mut Foo) { .. }
```



Mutable borrow


```
fn want_to_borrow(foo: &mut Foo) { .. }
```



Mutable borrow

```
fn push(from: &Vec<int>, to: &mut Vec<int>) {  
    for item in from.iter() {  
        to.push(item);  
    }  
}  
  
fn main() {  
    let one = vec![1, 2, 3];  
    let mut another = vec![];  
    push(&one, &mut another);  
}
```

Mutable borrow

```
fn push(from: &Vec<int>, to: &mut Vec<int>) {  
    for item in from.iter() {  
        to.push(item);  
    }  
}  
  
fn main() {  
    let mut one = vec![1, 2, 3];  
    push(&one, &mut one);   
}
```

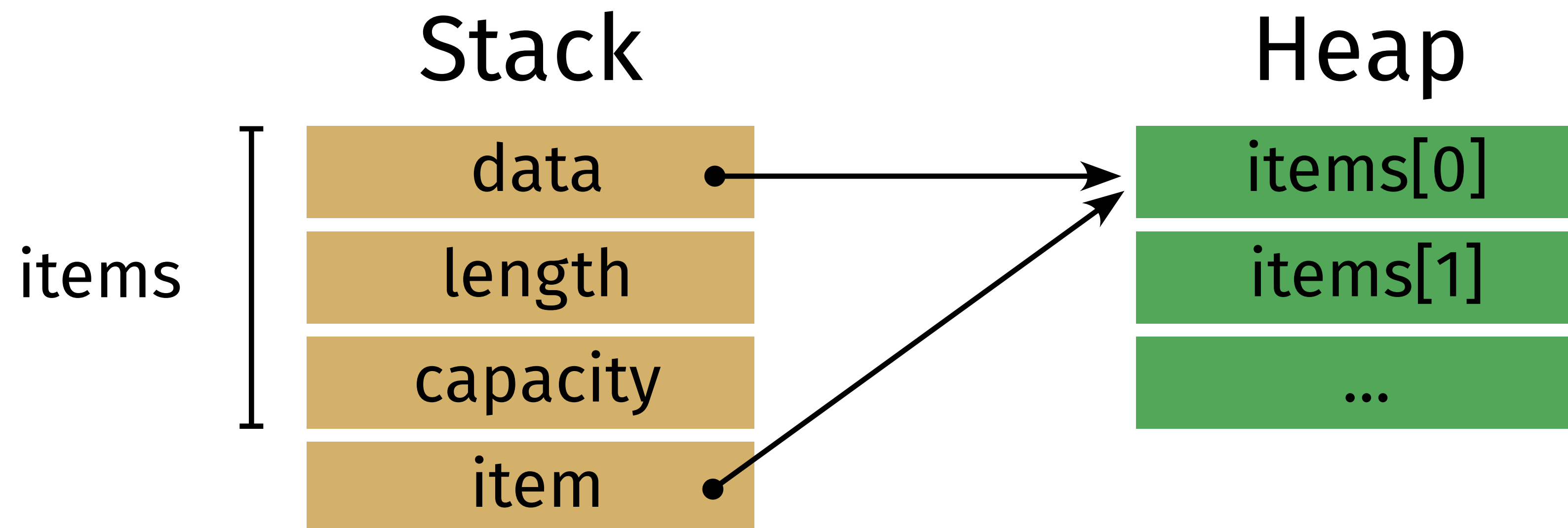
Mutable borrow

```
fn push(from: &Vec<int>, to: &mut Vec<int>) {  
    for item in from.iter() {  
        to.push(item);  
    }  
}  
  
fn main() {  
    let mut one = vec![1, 2, 3];  
    push(&one, &mut one);  
}
```

← Compilation error

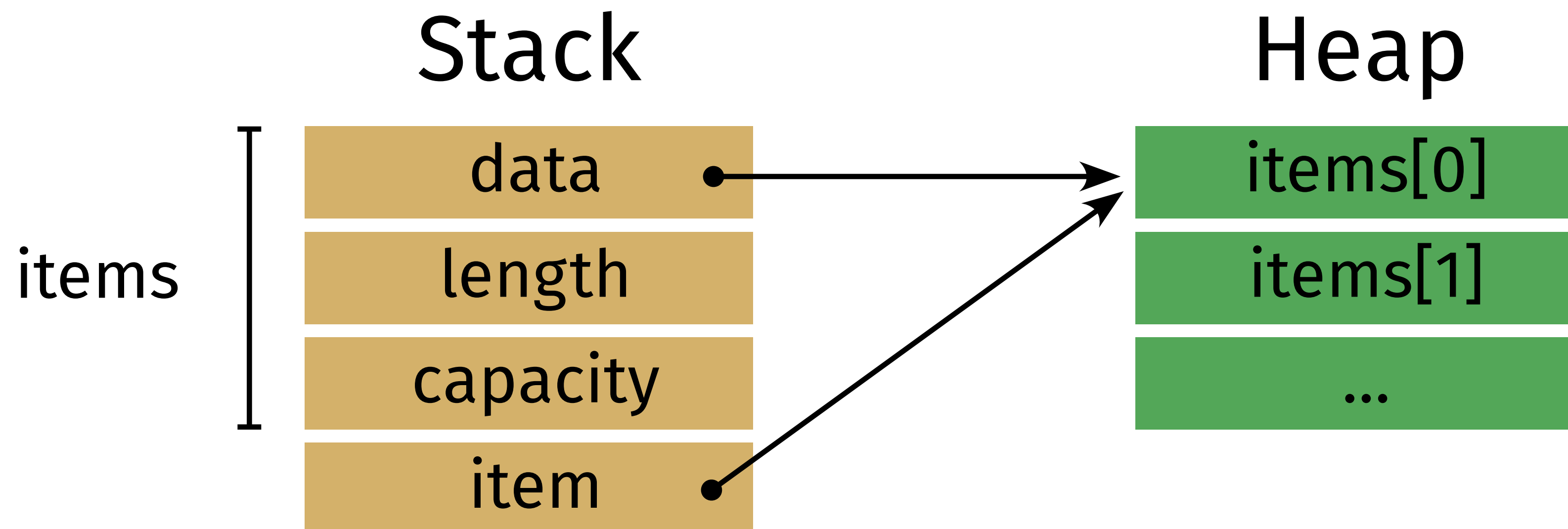
What about foo?

```
fn foo() {  
    let mut items = Vec::new();  
    ...  
    let item = &items[0];  
    items.push(...);  
    ...  
    use(item);  
}
```



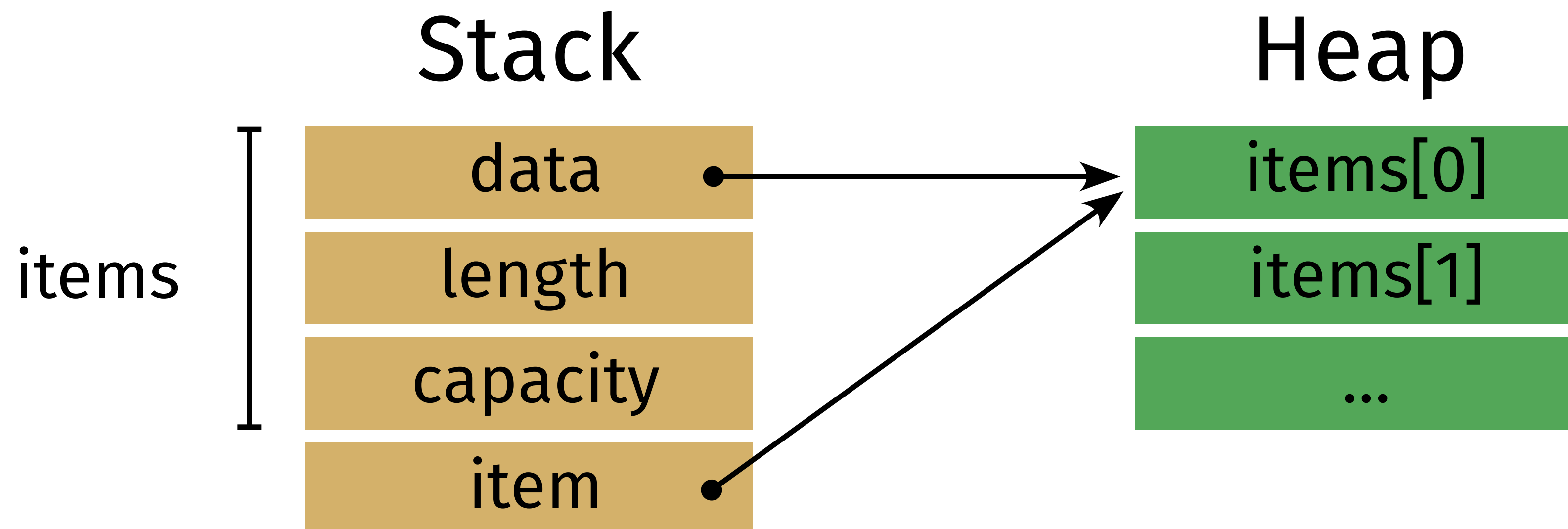
What about foo?

```
fn foo() {  
    let mut items = Vec::new();  
    ...  
    let item = &items[0];  
    items.push(...); ← Compilation error  
    ...  
    use(item);  
}
```



What about foo?

```
fn foo() -> &String {  
    let mut items = Vec::new();  
    ...  
    let item = &items[0];  
    ...  
    return item; ← Compilation error  
}
```



Lifetime

```
fn foo() -> &String {  
    let mut items = Vec::new();  
    ...  
    let item = &items[0];  
    ...  
    return item; }  
    }
```

← Compilation error

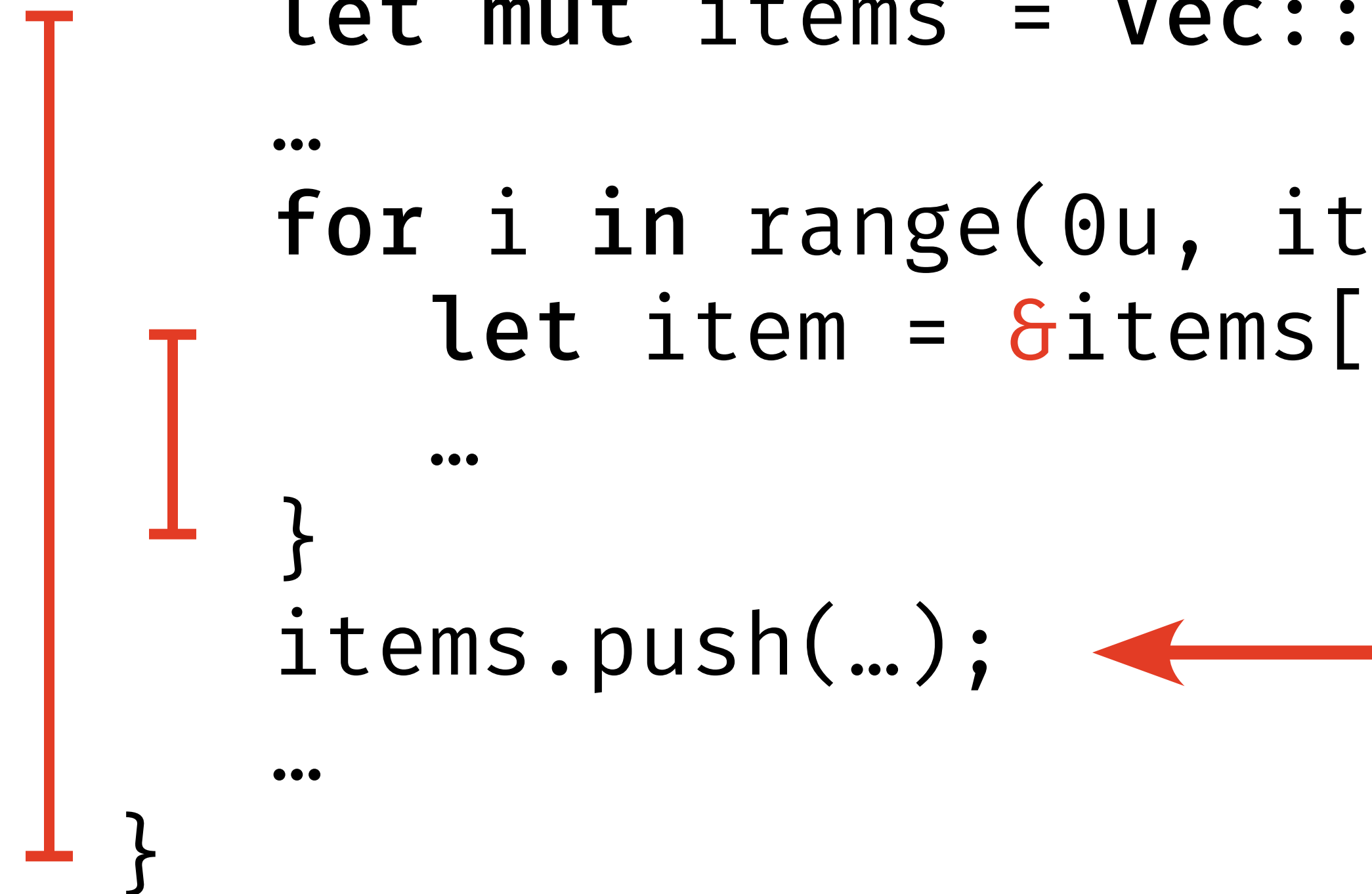
Lifetime

```
fn foo() {  
    let mut items = Vec::new();  
    ...  
    let item = &items[0];  
    items.push(...);  
    ...  
    use(item);  
}
```

← Compilation error


Lifetime

```
fn foo() {  
    let mut items = Vec::new();  
    ...  
    for i in range(0u, items.len()) {  
        let item = &items[i];  
        ...  
    }  
    items.push(...); ← OK  
    ...  
}
```

A diagram illustrating Rust's lifetime system. A long red vertical bracket on the left side of the code block spans from the opening curly brace of the function 'foo' to its closing curly brace, indicating the function's lifetime. A shorter red vertical bracket is positioned to the left of the 'for' loop, spanning from the opening curly brace of the loop to its closing curly brace, indicating the loop's lifetime. A red arrow points from the text 'OK' to the 'items.push(...);' line, indicating that this operation is valid because the data it pushes is created within the function's lifetime.

Conclusion

Rust



Control & Safety

Bicycle gear?



Photo: Andrew Dressel, CC BY-SA 3.0

Fungus!

- Robust
- Distributed
- Parallel



Photo: Andrew Fogg, CC BY 2.0

Thank you!
Questions?

<http://www.rust-lang.org>