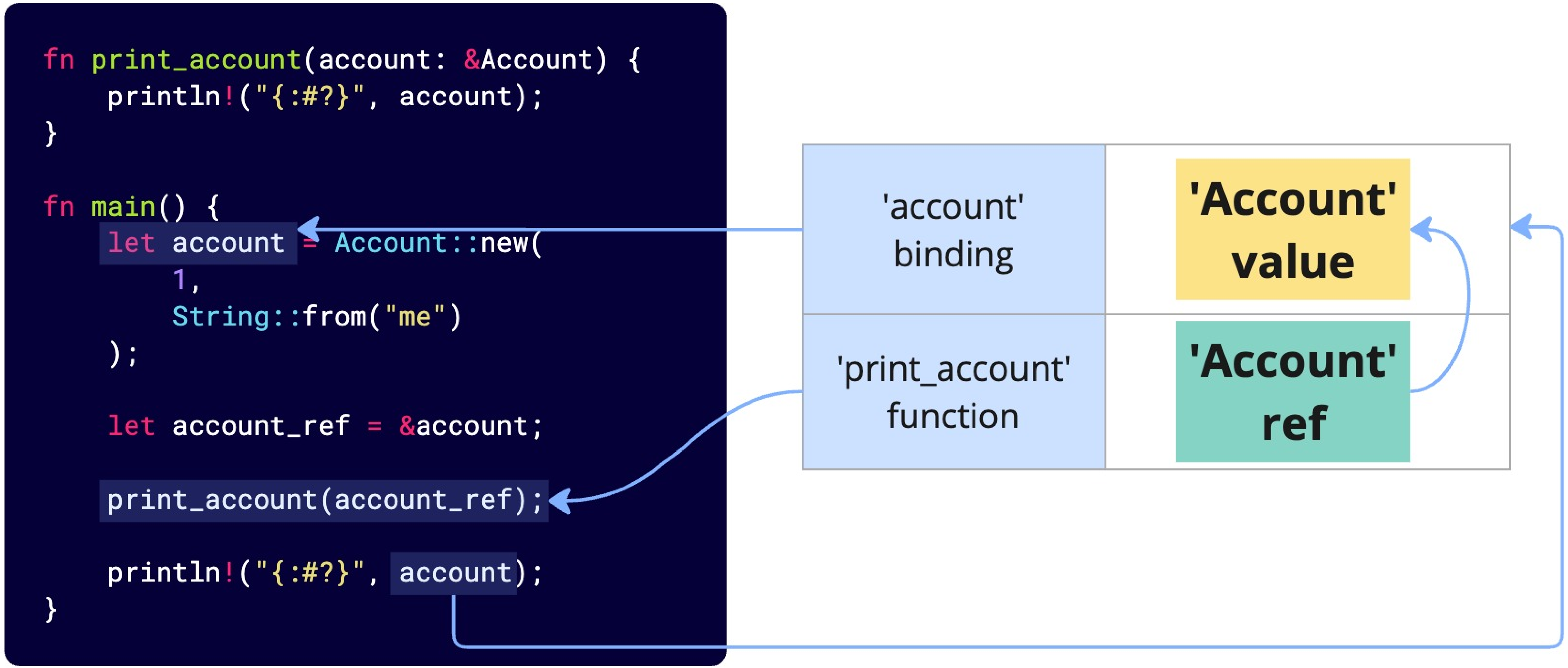
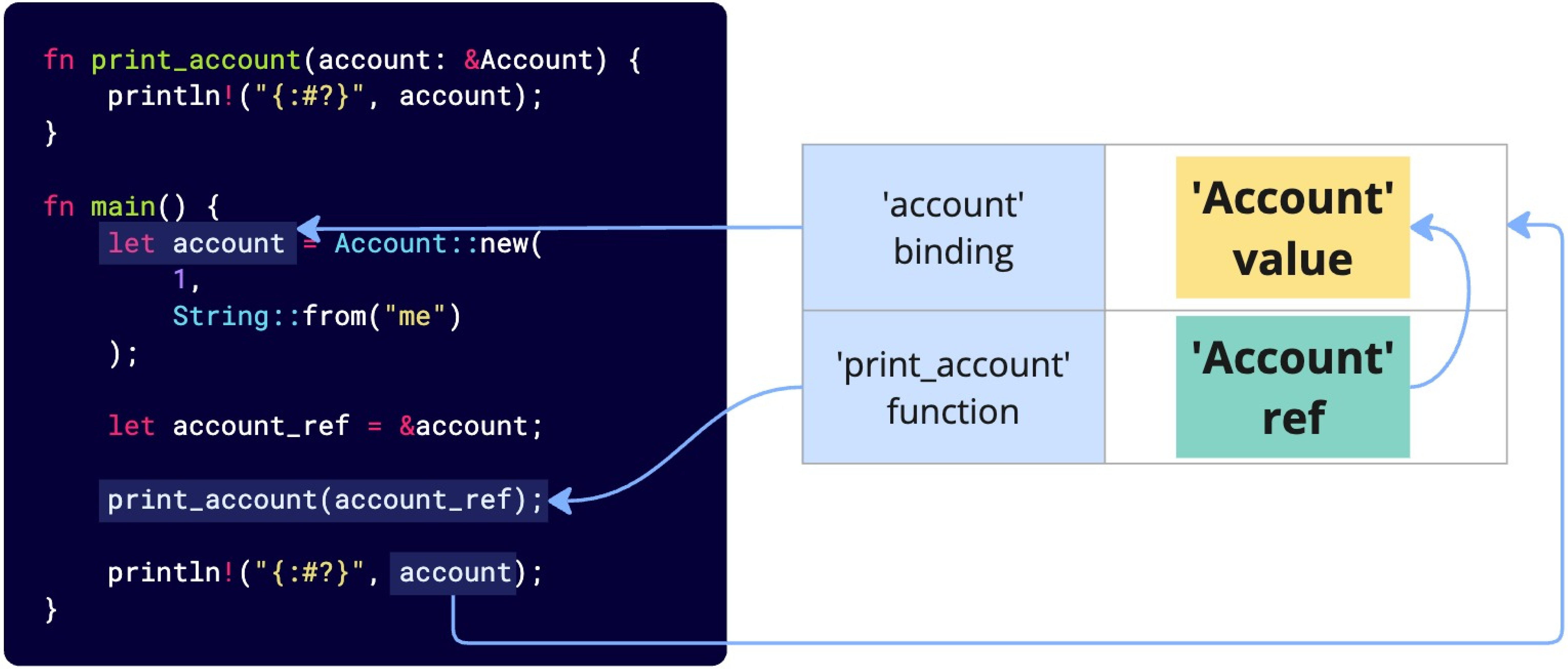
1	Every value is 'owned' by a single variable, argument, struct, vector, etc at a time	Ownership
2	Reassigning the value to a variable, passing it to a function, putting it into a vector, etc, *moves* the value. The old owner can't be used to access the value anymore!	
3	You can create many read-only references to a value that exist at the same time. These refs can all exist at the same time	
4	You can't move a value while a ref to the value exists	
5	You can make a writeable (mutable) reference to a value *only if* there are no read-only references currently in use. One mutable ref to a value can exist at a time	Borrowing
6	You can't mutate a value through the owner when any ref (mutable or immutable) to the value exists	
7	Some types of values are *copied* instead of moved (numbers, bools, chars, arrays/tuples with copyable elements)	
8	When a variable goes out of scope, the value owned by it is *dropped* (cleaned up in memory)	
9	Values can't be dropped if there are still active references to it	Lifetimes
10	References to a value can't outlive the value they refer to.	
11	**These rules will dramatically change how you write code (compared to other languages)**	
12	**When in doubt, remember that Rust wants to minimize unexpected updates to data**	



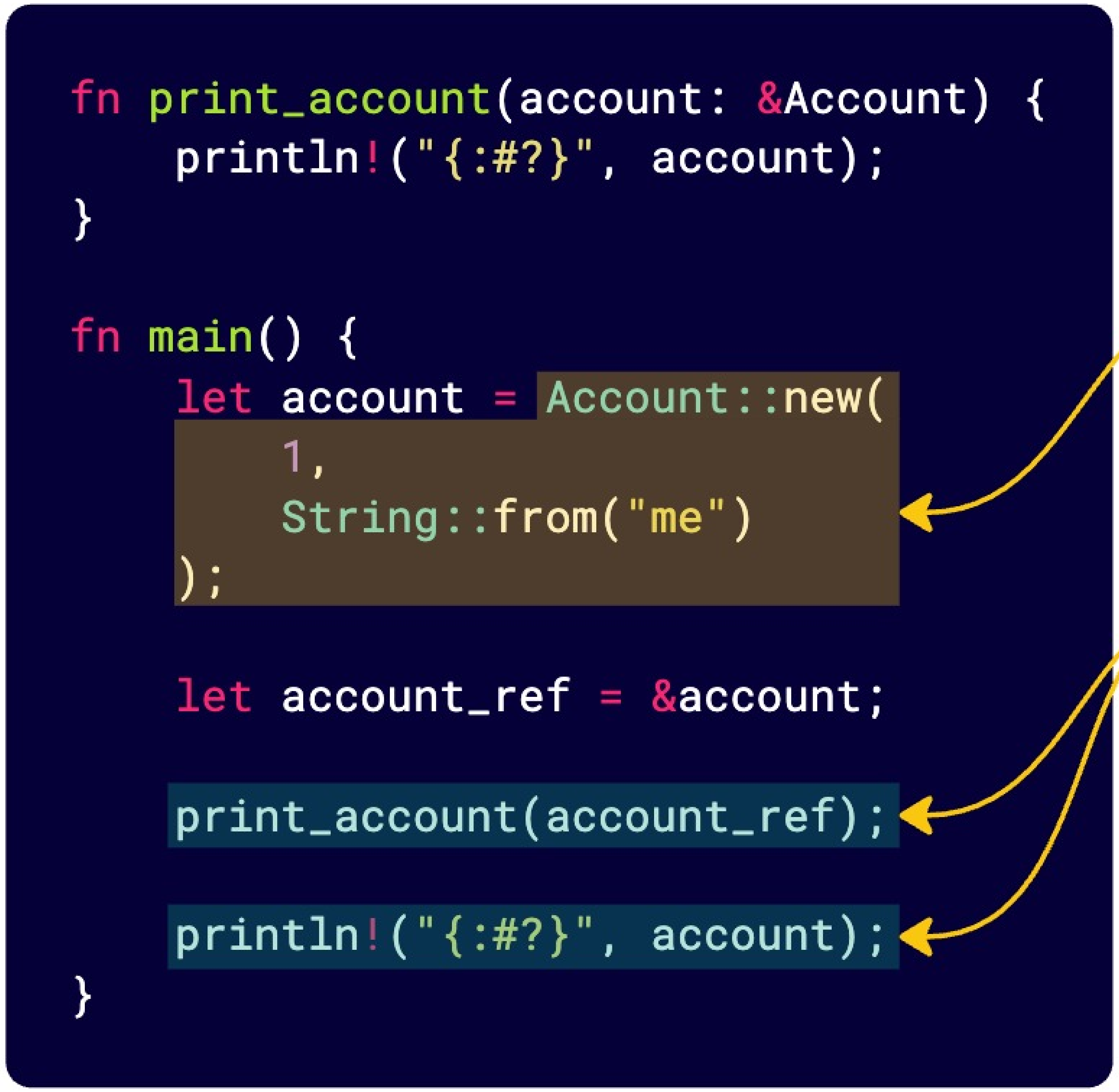
```rust
fn print_account(account: &Account) {
    println!("{:#?}", account);
}

fn main() {
    let account = Account::new(
        1,
        String::from("me")
    );

    let account_ref = &account;

    print_account(account_ref);

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

Read-only ref's allow us to look at a value without moving the value

Useful whenever we need to read a value in a function



```rust
fn print_account(account: &Account) {
    println!("{:#?}", account);
}

fn main() {
    let account = Account::new(
        1,
        String::from("me")
    );

    let account_ref = &account;

    print_account(account_ref);

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



```rust
fn print_account(account: &Account) {
    println!("{:#?}", account);
}

fn main() {
    let account = Account::new(
        1,
        String::from("me")
    );

    let account_ref = &account;

    print_account(account_ref);

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

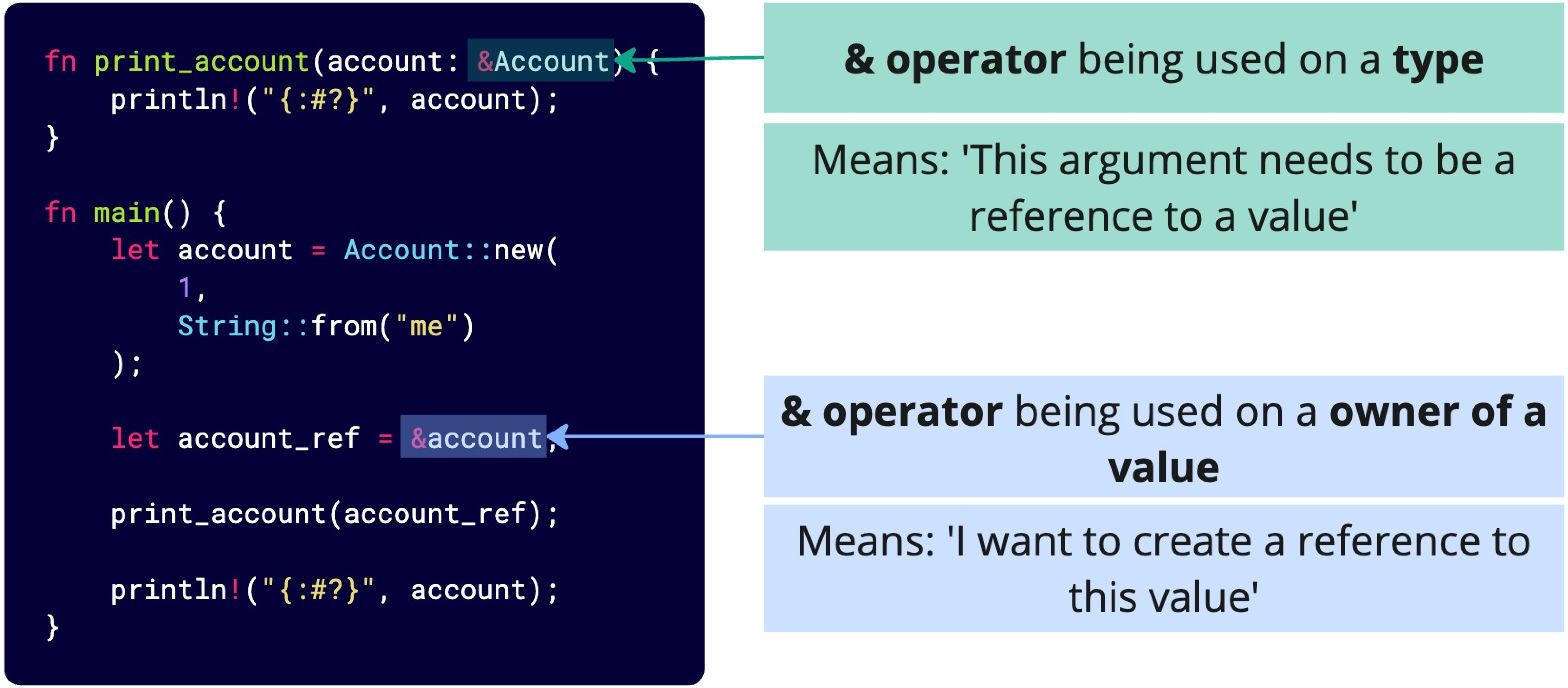
Refs allow us to look at a value without moving it

I want to make a value...

Then use that value in several locations

Changing the owner of the value manually would be tedious!

Good solution is to use a reference



```rust
fn print_account(account: &Account) {
    println!("{:#?}", account);
}

fn main() {
    let account = Account::new(
        1,
        String::from("me")
    );

    let account_ref = &account;

    print_account(account_ref);

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

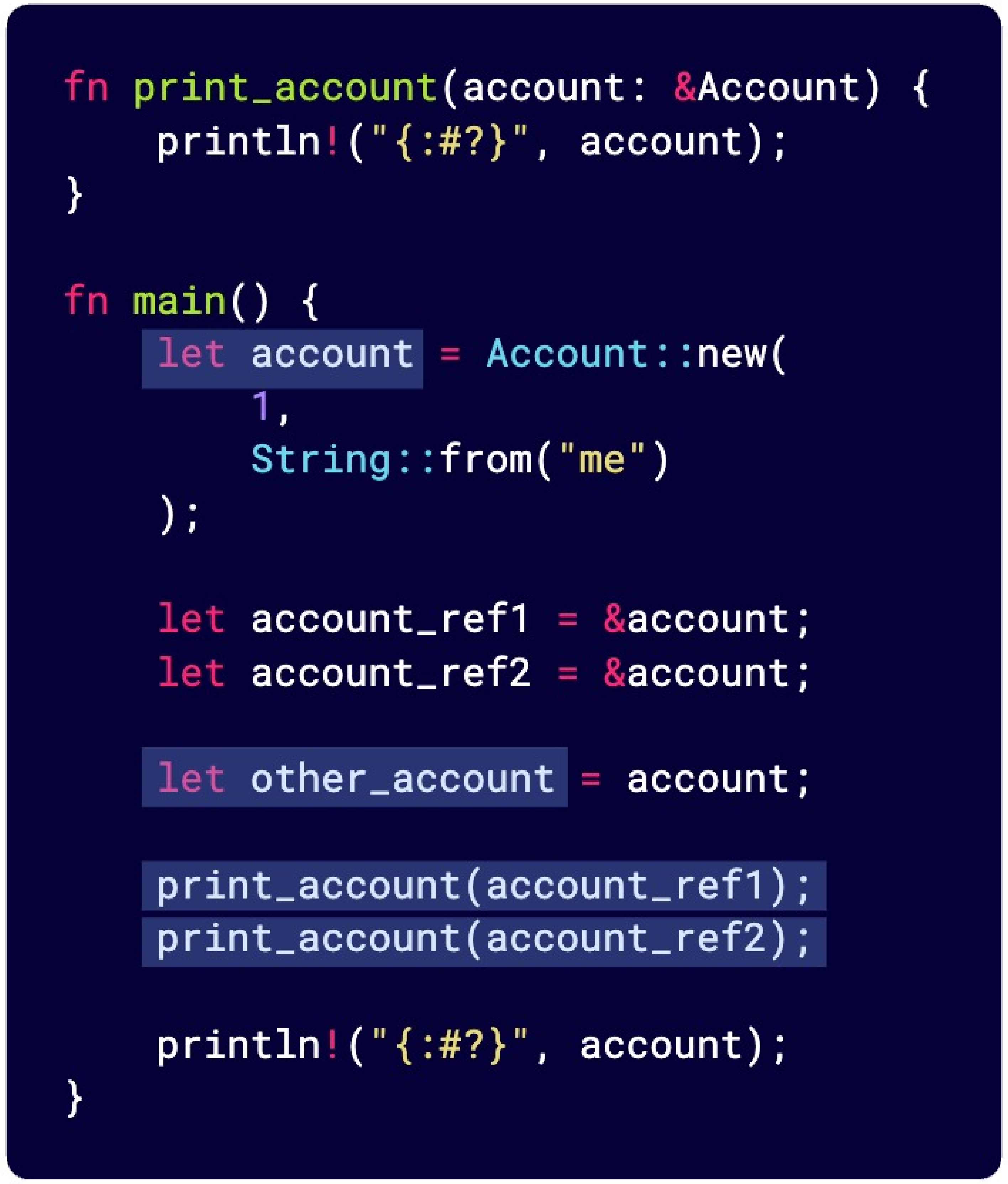
& operator being used on a **type**

Means: 'This argument needs to be a reference to a value'

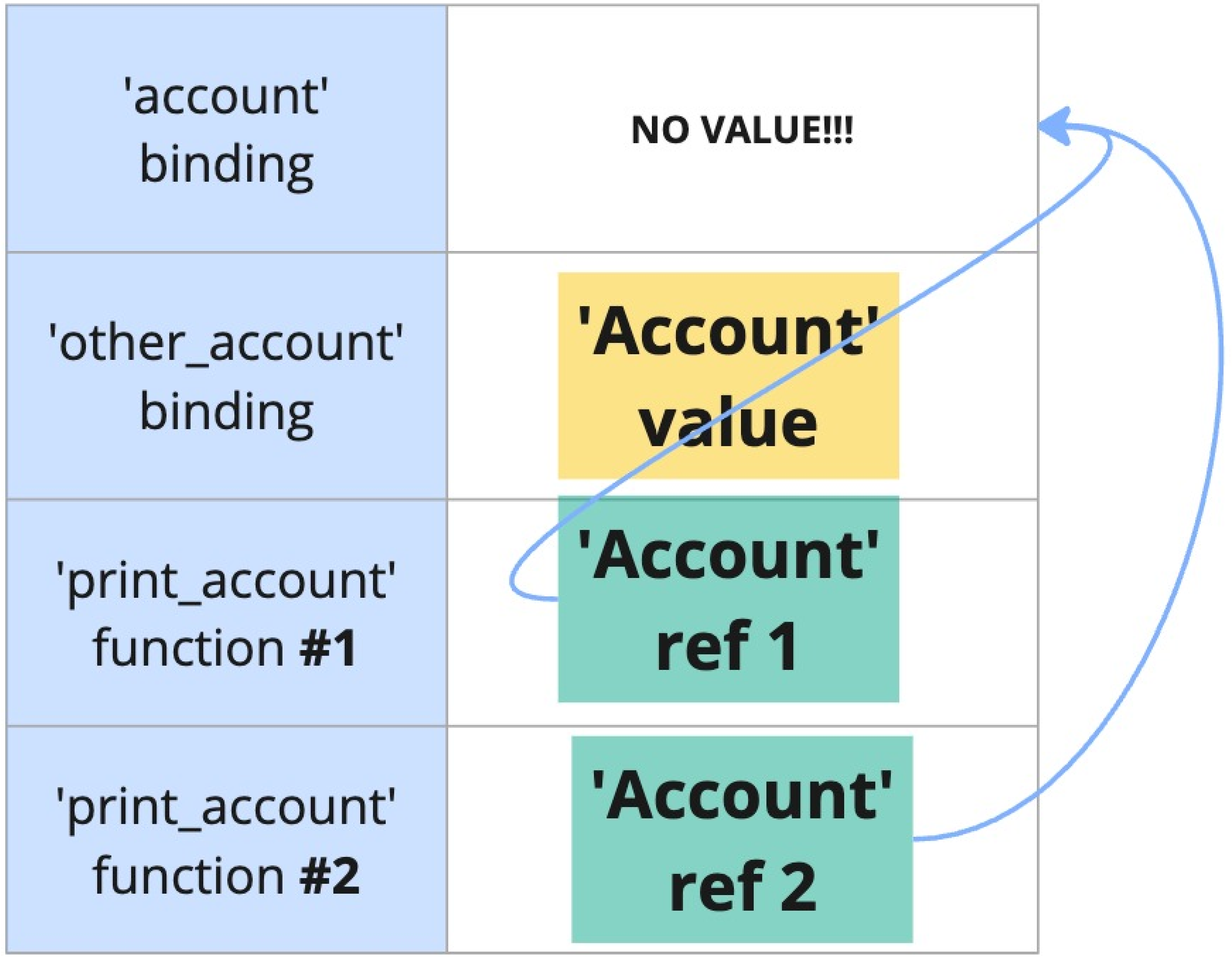
& operator being used on a **owner of a value**

Means: 'I want to create a reference to this value'

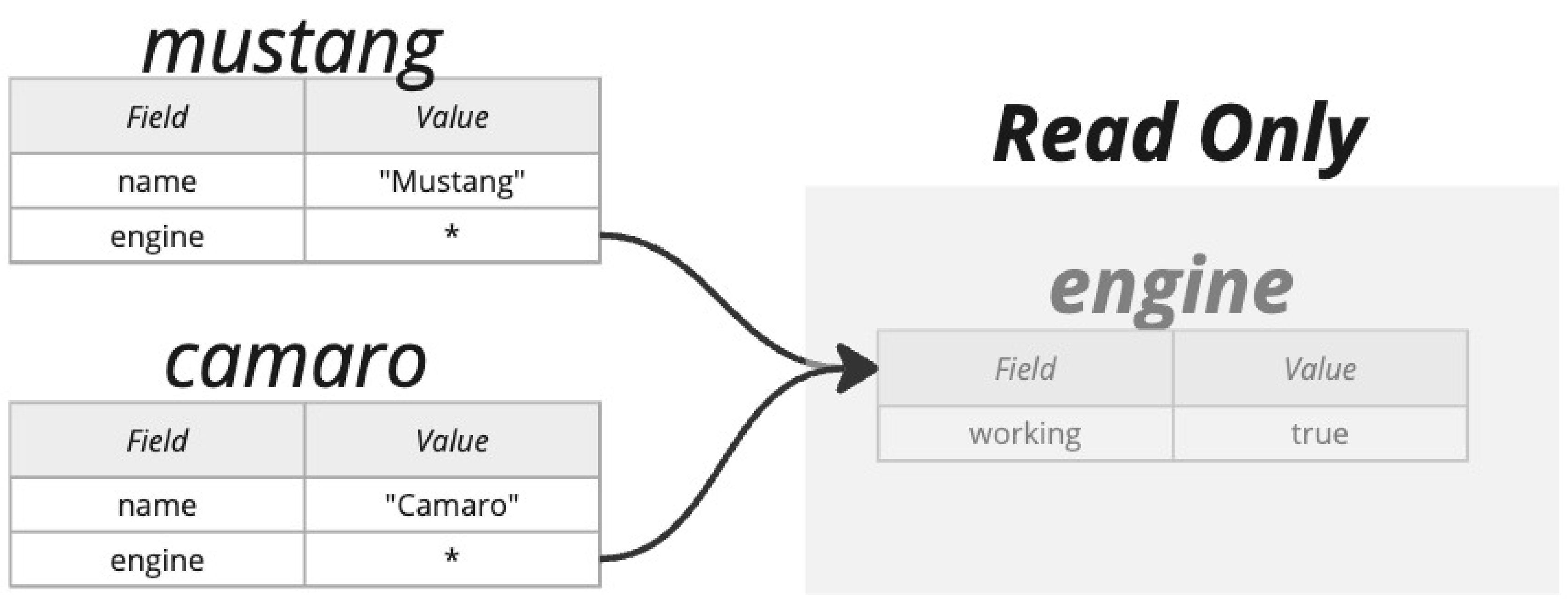
3	You can create many read-only (immutable) references to a value. These refs can all exist at the same time.
4	You can't move a value while a ref (immutable or mutable) to the value exists.



```rust
fn print_account(account: &Account) {
    println!("{:#?}", account);
}

fn main() {
    let account = Account::new(
        1,
        String::from("me")
    );

    let account_ref1 = &account;
    let account_ref2 = &account;

    let other_account = account;

    print_account(account_ref1);
    print_account(account_ref2);

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



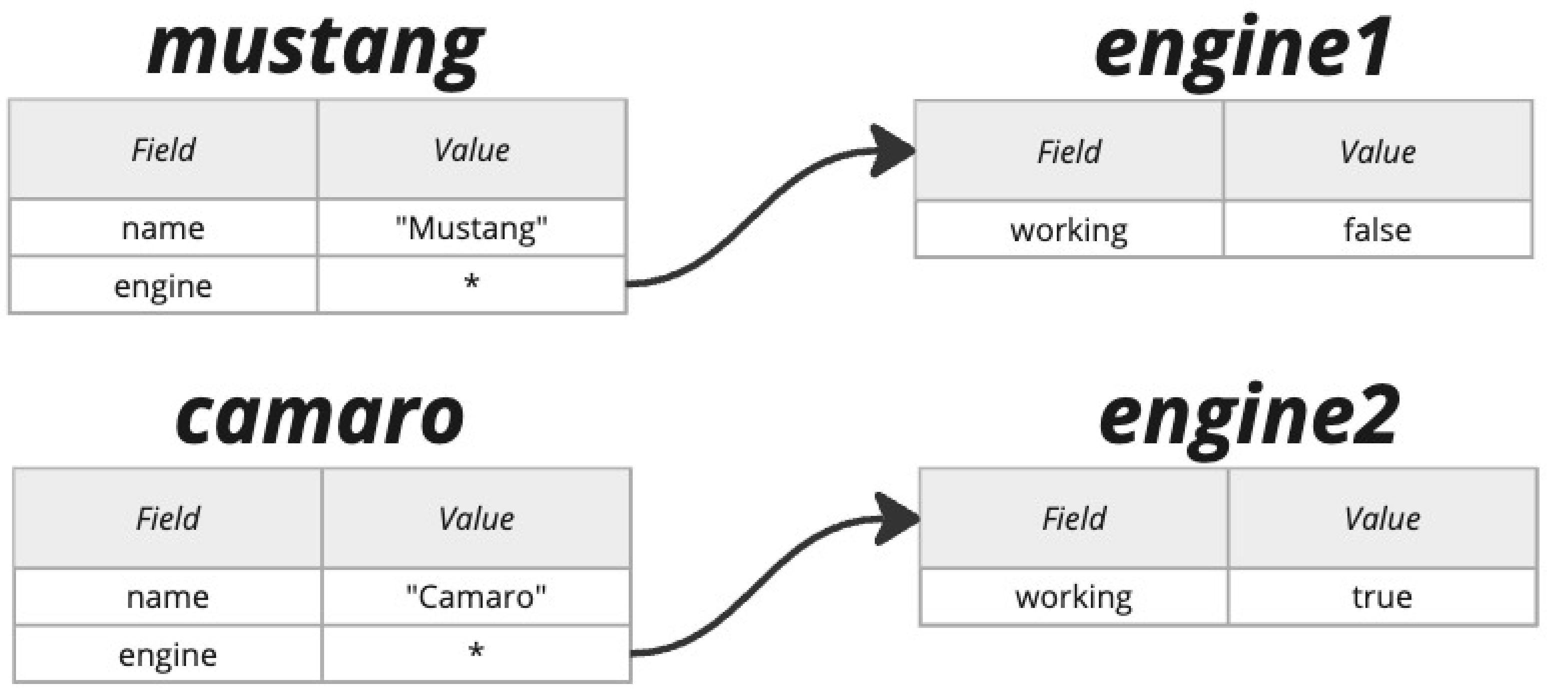
Fix #1
Each car can refer to the same engine, but can't modify it



We'd never get that bug if...

Multiple things can refer to a value at the same time, but the references have to be read-only

Fix #2
Each car "owns" a different engine



mustang

Field	*Value*
name	"Mustang"
engine	*

engine1

Field	*Value*
working	false

camaro

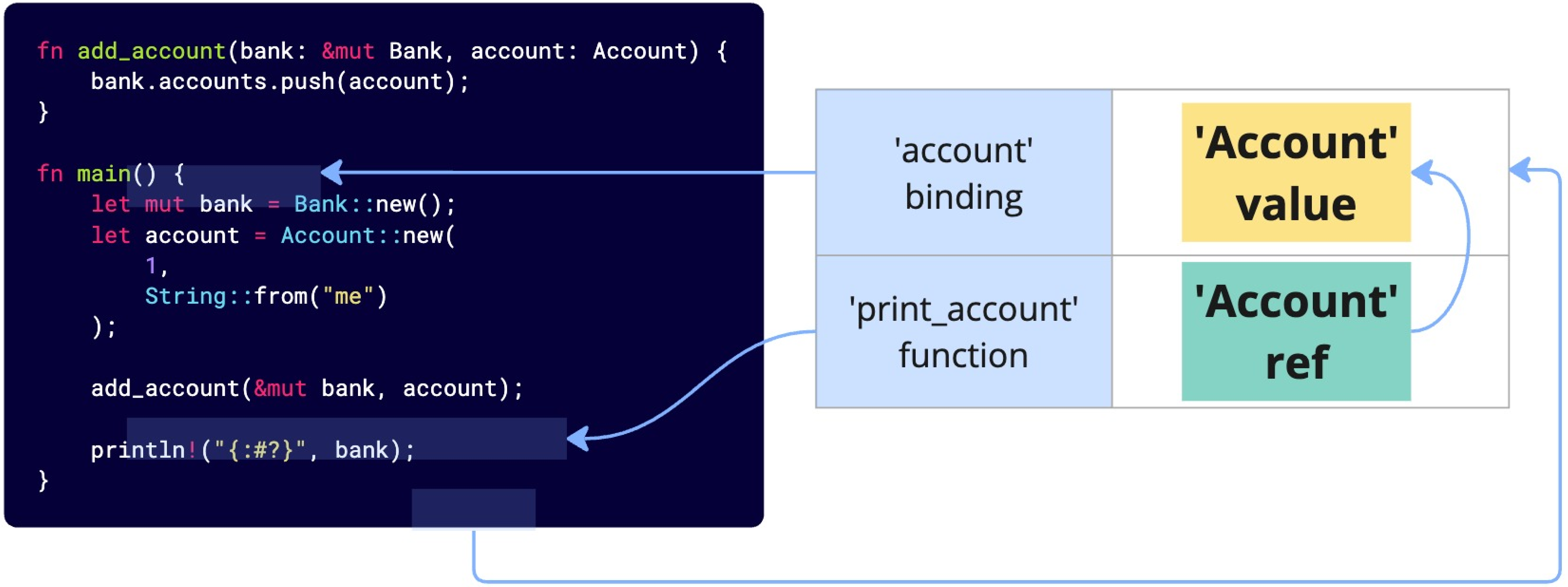
Field	*Value*
name	"Camaro"
engine	*

engine2

Field	*Value*
working	true

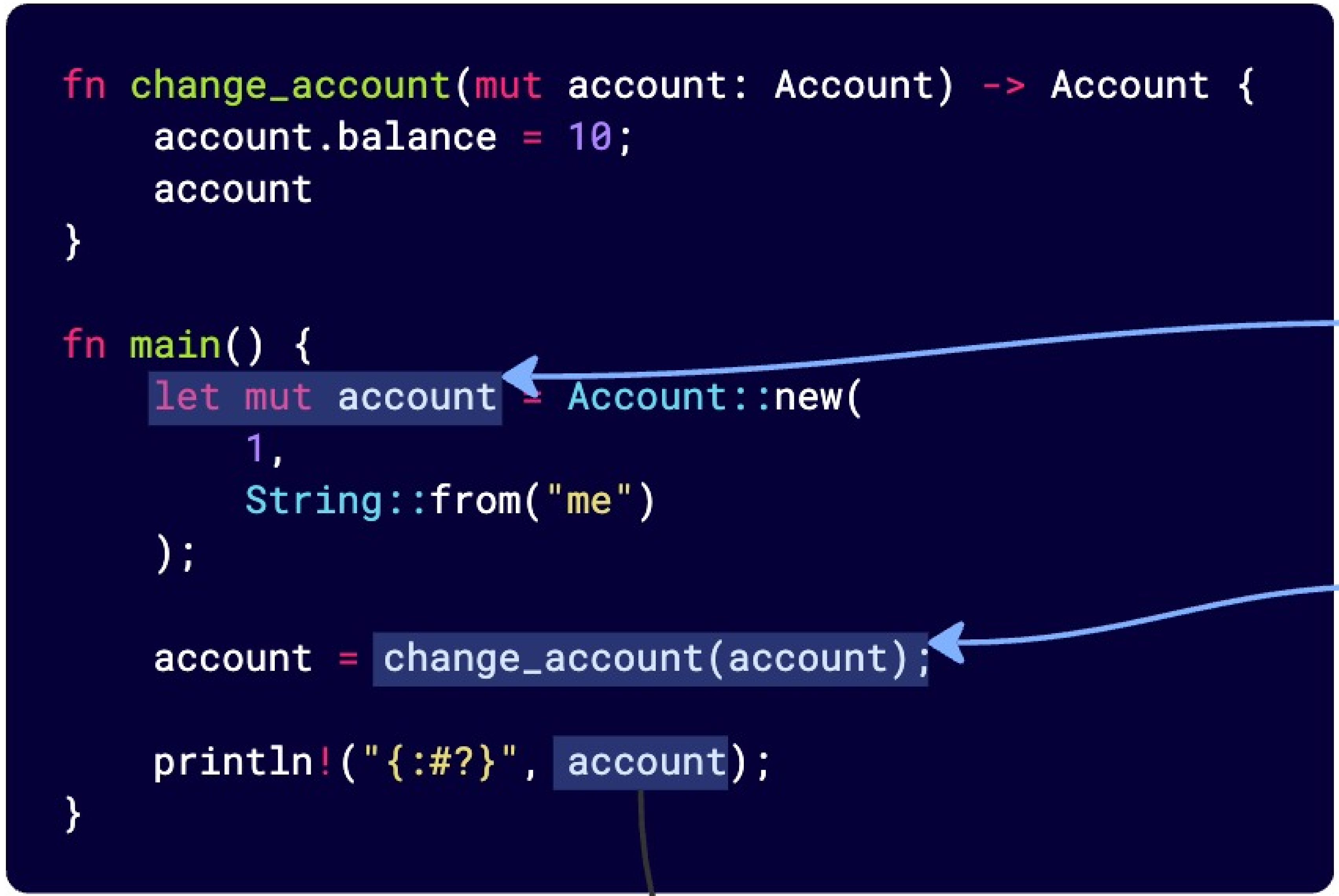
We'd never get that bug if...

A value can *only* be updated when there are no other references to it

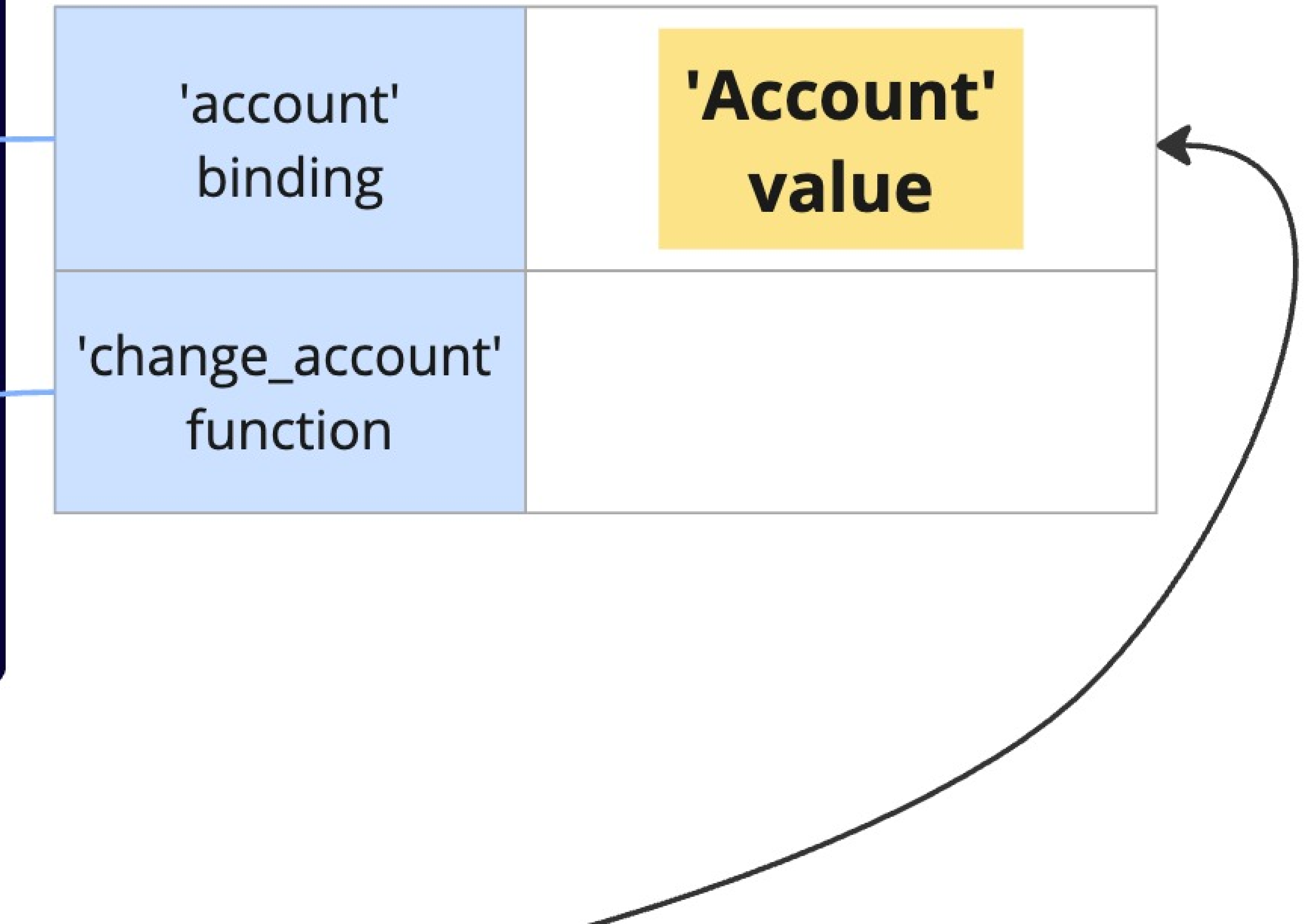


```rust
fn add_account(bank: &mut Bank, account: Account) {
    bank.accounts.push(account);
}

fn main() {
    let mut bank = Bank::new();
    let account = Account::new(
        1,
        String::from("me")
    );

    add_account(&mut bank, account);

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

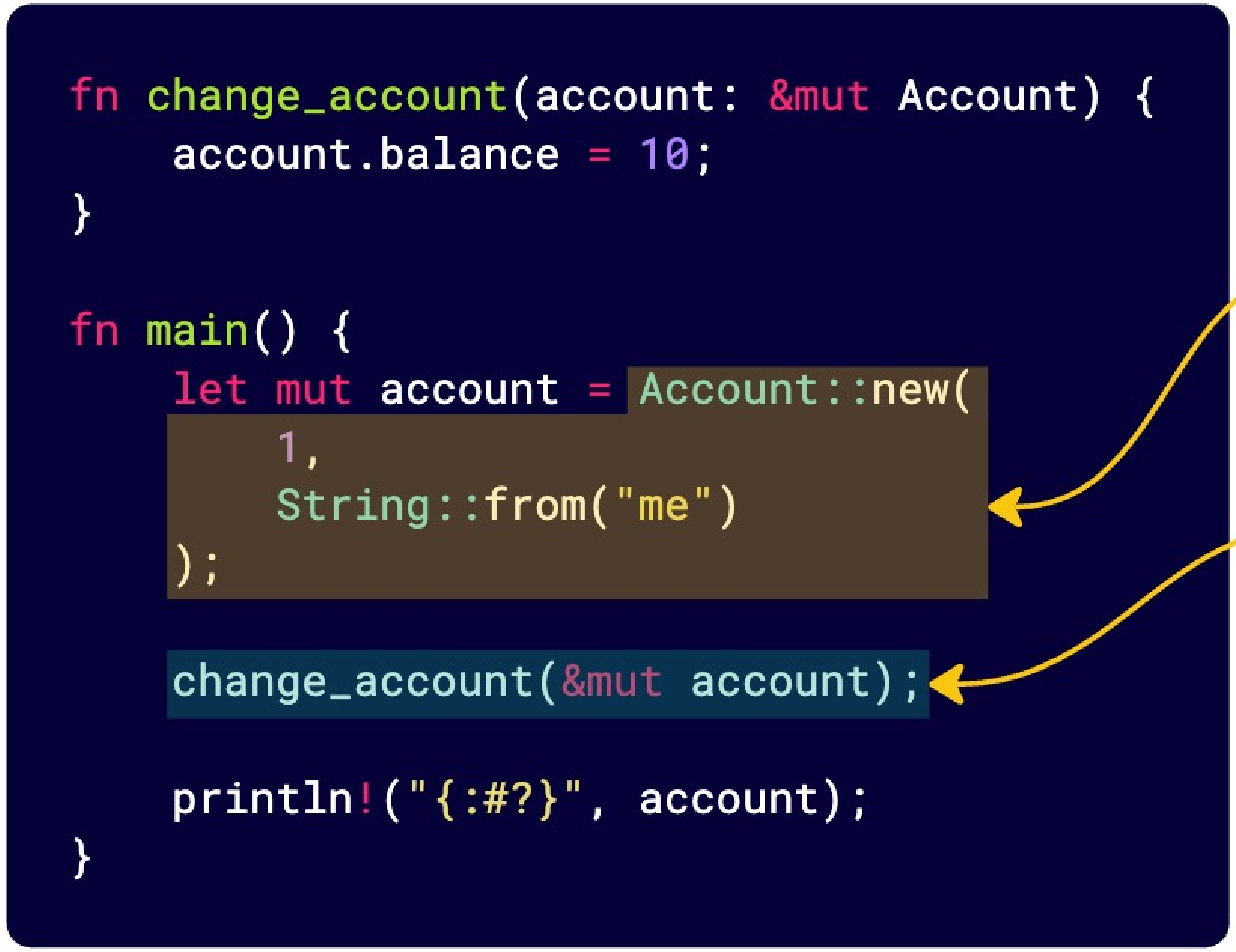
Moving ownership to update something
Really tedious



```
fn change_account(mut account: Account) -> Account {
    account.balance = 10;
    account
}

fn main() {
    let mut account = Account::new(
        1,
        String::from("me")
    );

    account = change_account(account);

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



Mutable refs allow us to read or change a value without moving it



```rust
fn change_account(account: &mut Account) {
    account.balance = 10;
}

fn main() {
    let mut account = Account::new(
        1,
        String::from("me")
    );

    change_account(&mut account);

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

I want to make a value...

Then allow that value to be changed somewhere else

Moving the value around manually would be tedious

Good solution is to use a mutable reference

5	You can make a writeable (mutable) reference to a value *only if* there are no read-only references currently in use. One mutable ref to a value can exist at a time
6	You can't mutate a value through the owner when any ref (mutable or immutable) to the value exists

7 Some types of values like numbers, booleans, etc are going to appear to **break the rules of ownership!!**

Some types of values are ***copied*** instead of moved

This means they behave more like values in other languages

$\rightarrow$

All numbers
(Examples: i32, u32, f32)

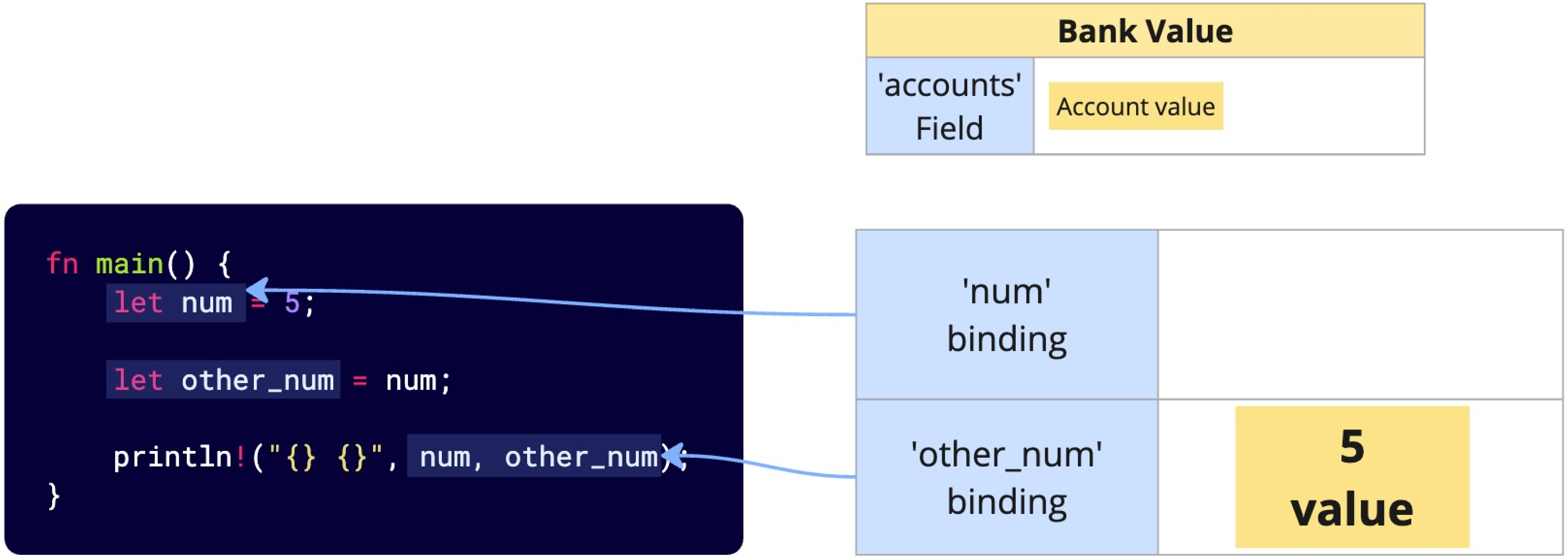
bool
(true/false)

char
(single characters)

Arrays
(if everything inside is Copy-able)

Tuples
(if everything inside is Copy-able)

References
(both readable and writable)



```rust
fn main() {
    let num = 5;

    let other_num = num;

    println!("{} {}", num, other_num);
}
```