Ownership in Rust:

the core principle for API design

Apoorv Kothari

content

- what is ownership?

- motivation for ownership

- relevant ownership rules

- designing an API

what is ownership?

what is ownership?

what is ownership?

```
let foo = 5;
```

- `value` is a property which says that **foo is equal to 5**

- `ownership` is a property which says **foo owns 5**
  - purely a compile-time property (no runtime cost)

```
let foo = String::from("data");
```

```
let foo = String::from("data");
let bar = foo;  // move ownership
```

```
let foo = String::from("data");
let bar = foo;  // move ownership
println!("{}", foo);  // try to use moved value
```

```
let foo = String::from("data");
let bar = foo;  // move ownership
println!("{}", foo);  // try to use moved value
```

```
9  | let bar = foo;
   |     --- value moved here
10 | // invalid because foo no longer owns "data"
11 | println!("{}", foo);
   |                ^^^ value used here after move
```

what is ownership?

```
let foo = String::from("data");
let bar = foo;  // move ownership
println!("{}", foo);  // try to use moved value
```

```
9  | let bar = foo;
   |     --- value moved here
10 | // invalid because foo no longer owns "data"
11 | println!("{}", foo);
   |                ^^^ value used here after move
```

what is ownership?

```
let foo = String::from("data");
let bar = foo;  // move ownership
println!("{}", foo);  // try to use moved value
```

```
9  | let bar = foo;
   |     --- value moved here
10 | // invalid because foo no longer owns "data"
11 | println!("{}", foo);
   |                ^^^ value used here after move
```

well actually... ownership/borrow/lifetimes

- these three make up the "ownership system" in Rust

- we will only cover **ownership** and **borrow**

- lifetimes:

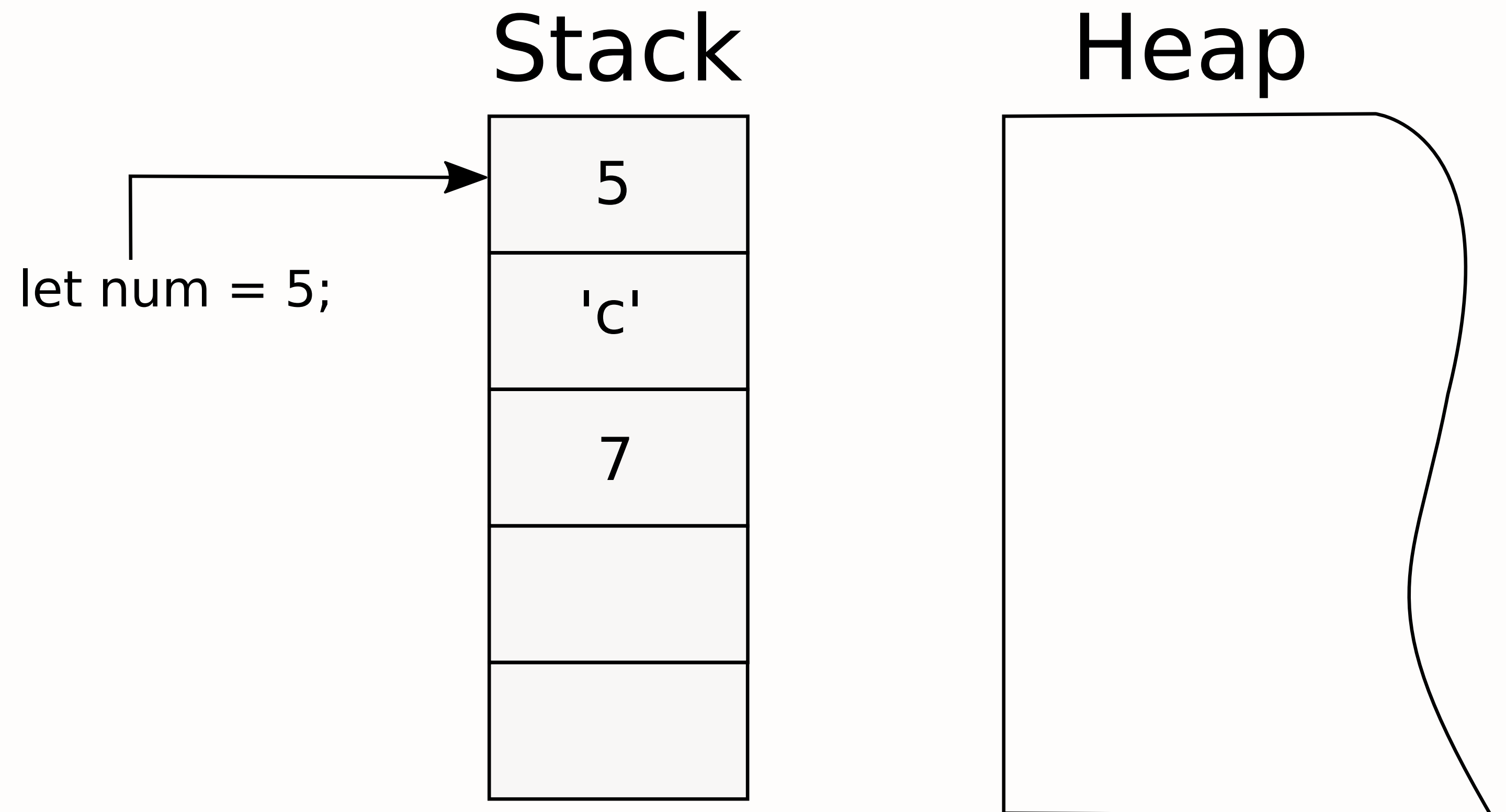   is a concept used by the compiler

    to measure when (borrowed/referenced) data goes of scope and

    recover memory once data is finished being borrowed
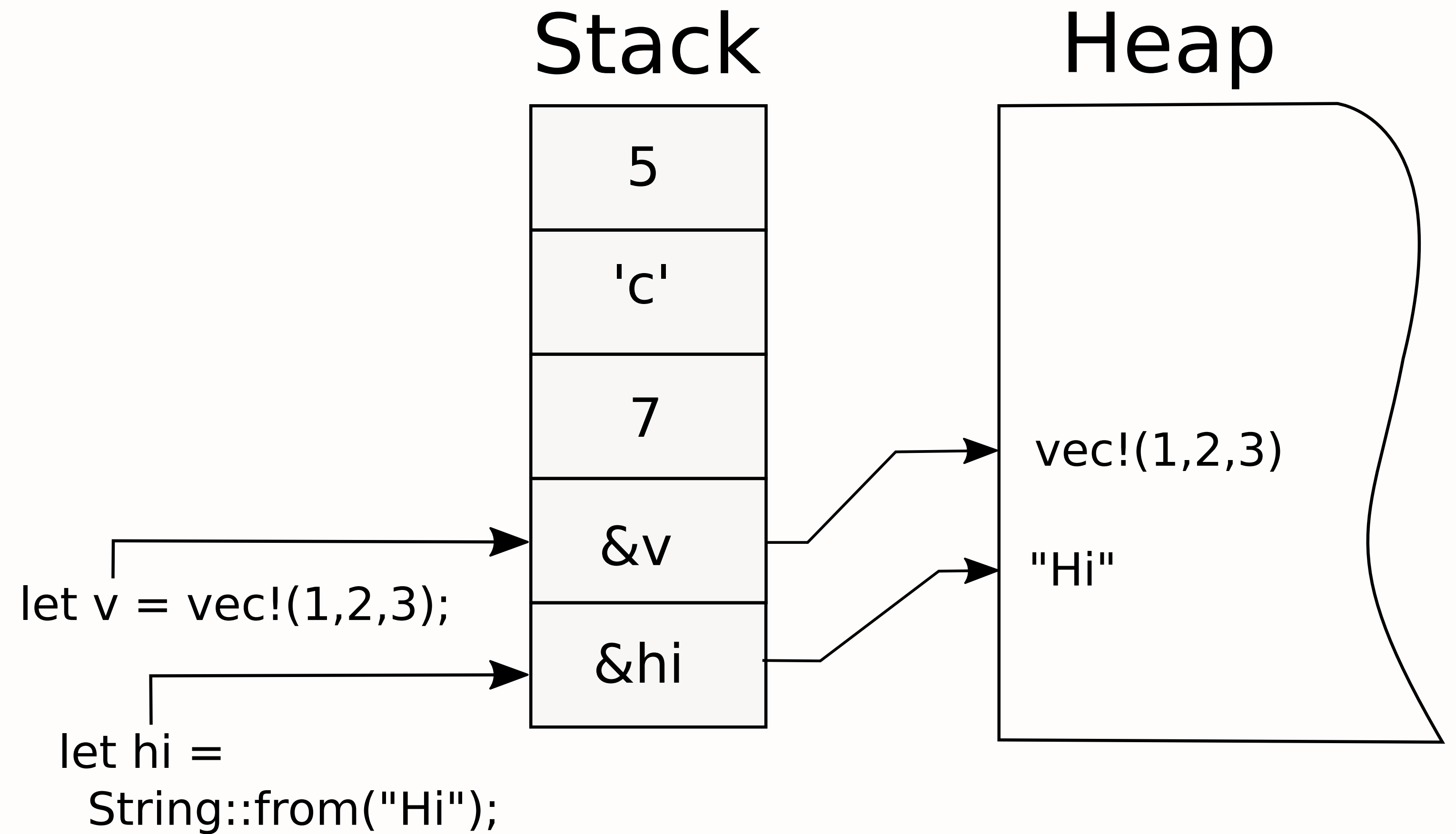
motivation for ownership

stack vs heap

- stack
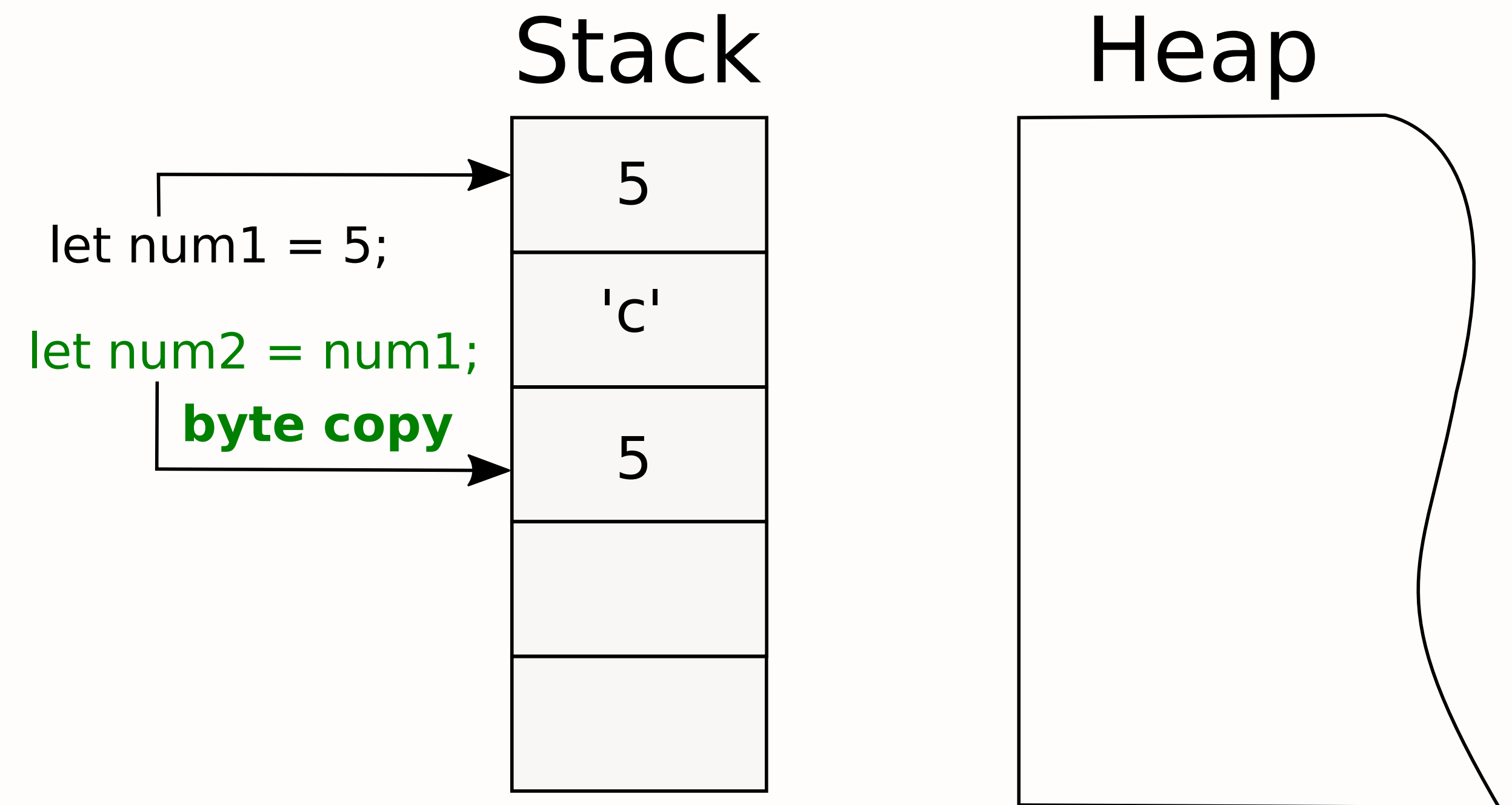 - u32, i32, bool, char, &, etc.
  - fast
  - limited space

Stack

5
'c'
7

let num = 5;

Heap

stack vs heap

- stack
  - u32, i32, bool, char, &, etc.
  - fast
  - limited space

- heap

  - String, Vector
  - slow
  - *unlimited size

Stack

5
'c'
7
&v
&hi

Heap

vec!(1,2,3)

"Hi"

let v = vec!(1,2,3);

let hi =
   String::from("Hi");

byte vs deep copy

- byte copy is the default behavior
  - copy the bytes representing 5

Stack

| 5 |
| 'c' |
| 5 |
| |
| |

Heap

let num1 = 5;
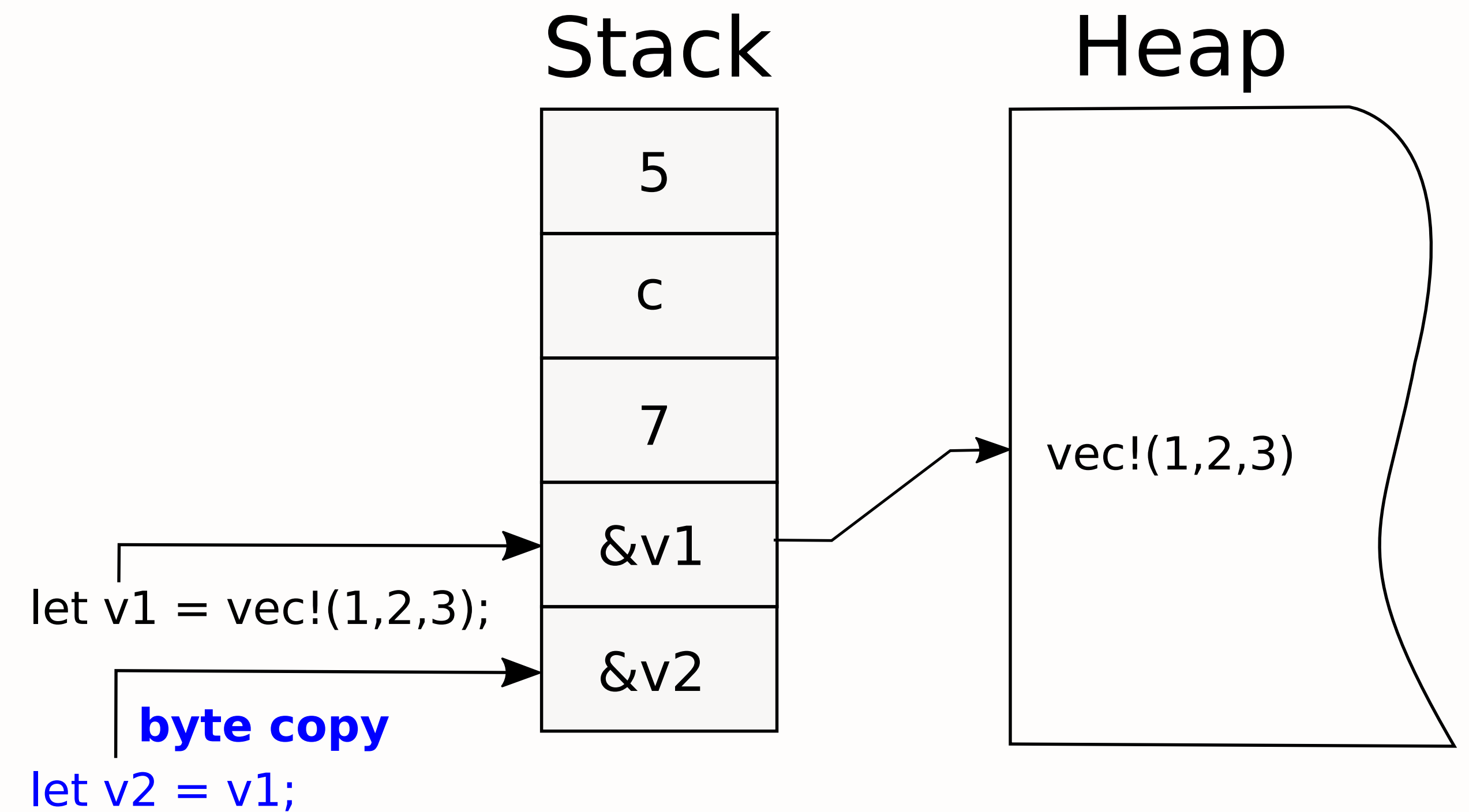
let num2 = num1;

byte copy

byte vs deep copy

- byte copy is the default behavior
  - copy the bytes representing 5

- deep copy is necessary for data living on the heap
  - follow all the pointers and copy data they point to

Stack

5
'c'
5
&v1
&v2

Heap

vec!(1,2,3)

vec!(1,2,3)

let v1 = vec!(1,2,3);

deep copy

let v2 = v1.clone();

implication of byte copy

Stack

Heap

5
c
7
&v1
&v2

vec!(1,2,3)

let v1 = vec!(1,2,3);

byte copy

let v2 = v1;
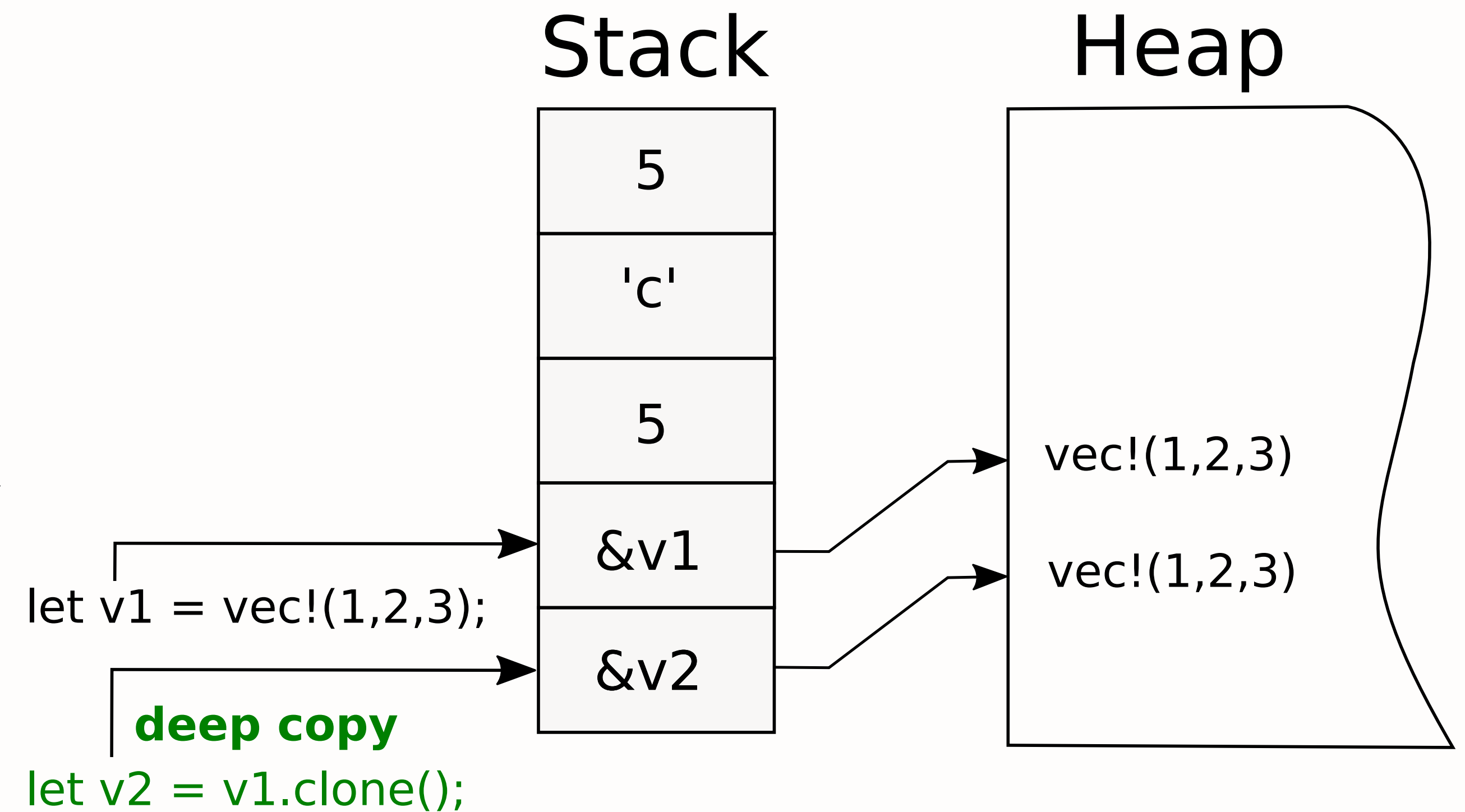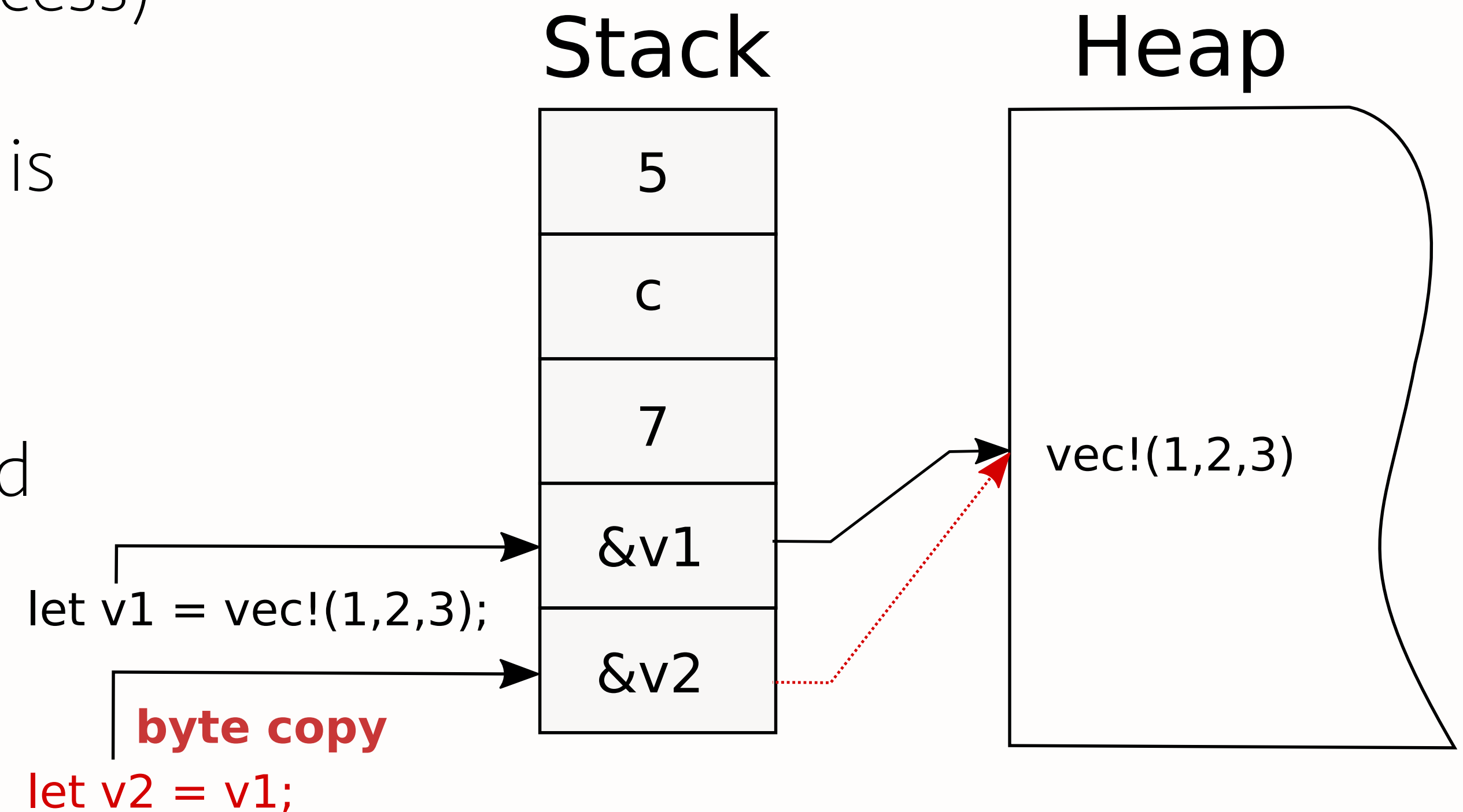
implication of byte copy

- overwriting data (multi-threaded access)

- undefined behavior (if vector length is changed)

- access neighbors data (compromised security)

- corrupting of data

Stack

5
c
7
&v1
&v2

let v1 = vec!(1,2,3);

byte copy
let v2 = v1;

Heap

vec!(1,2,3)

solutions to the 'copy' problem

Copy Options	Speed	Safety
manage memory and use a GC process (Java)	no	yes
do a byte copy let the developer handle it (C)	yes	no
always do a deep copy	no	yes
invalidate previous variable by moving ownership (Rust)	yes	yes

relevant ownership rules

ways to share data in Rust

move
(ownership semantic)

```rust
let foo = String::from("data");
let bar = foo;
// foo is invalidated
```

ways to share data in Rust

move
(ownership semantic)

```rust
let foo = String::from("data");
let bar = foo;
// foo is invalidated
```

reference (&T)
(borrow semantic)

```rust
let foo = String::from("data");
let bar = &foo;
// foo is valid
```

ways to share data in Rust

move
(ownership semantic)

```rust
let foo = String::from("data");
let bar = foo;
// foo is invalidated
```

reference (&T)
(borrow semantic)

```rust
let foo = String::from("data");
let bar = &foo;
// foo is valid
```

mutable ref (&mut T)
(borrow semantic)

```rust
let mut foo = String::from("data");
let bar = &mut foo;
// foo is valid, but only 1 ref allowed
```

ways to share data in Rust

move (ownership)

```
let foo = String::from("data");
let bar = foo;
```

NO RUNTIME COSTS

reference (&T) (borrow)

```
let foo = String::from("data");
let bar = &foo;
```

mutable ref (&mut T) (borrow)

```
let mut foo = String::from("data");
let bar = &mut foo;
```

relevant ownership rules

- each value has a variable which is its owner /

  there can only be one owner at a time   (ownership)

- you can have as many *&T* as you want   (borrow)

- you can have only one *&mut T* in scope   (borrow)

each value has a variable which is its owner /
there can only be one owner at a time

```rust
let foo: String = String::from("data");
let bar: String = foo; //ownership moved
println!("{}", foo); //moved after use
```

```
9  | let bar = foo;
   |     --- value moved here
10 | // invalid because foo no longer owns "data"
11 | println!("{}", foo);
   |                ^^^ value used here after move
```

you can have as many &T as you want

```rust
let foo = String::from("data");
let bar = &foo;  //create a reference
println!("{}", foo);  //foo still owns its data
println!("{}", foo);  //multiple reference allowed
```

you can have only one &mut T in scope

```
let mut foo = String::from("data");
let bar = &mut foo;   //create a mut reference
println!("{}", &foo);  //only 1 mut reference allowed
```

```
37 |            let bar = &mut foo;  //create a mut reference
   |                           --- mutable borrow occurs here
38 |         println!("{}", foo);  //only 1 mut reference allowed
   |                        ^^^ immutable borrow occurs here
40 |     }
   |     - mutable borrow ends here
```

designing an API

guarantees of the ownership system

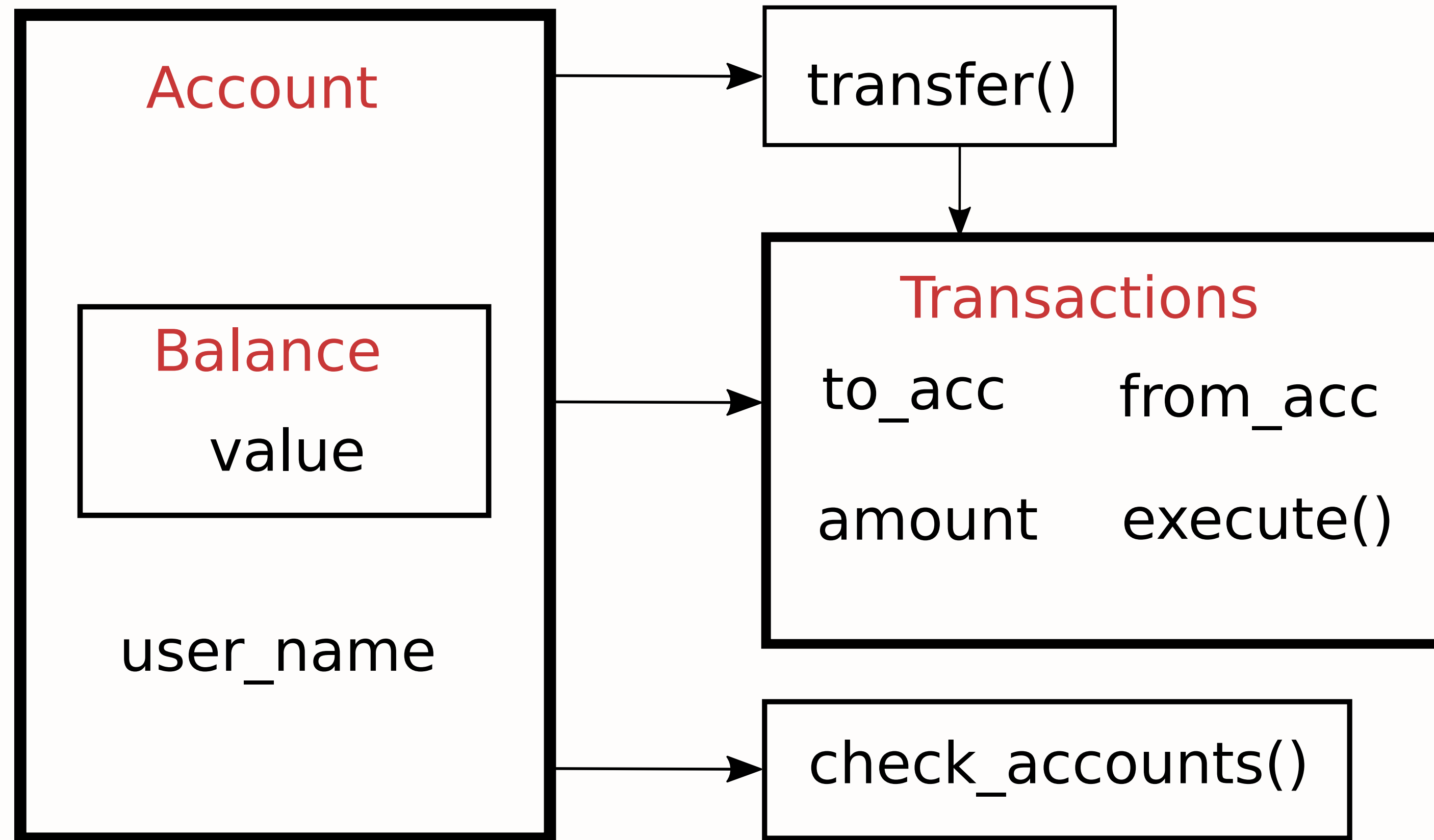- **move**: owner controls how long the data is *valid* for

guarantees of the ownership system

- move: owner controls how long the data is valid for

- &T: *multiple immutable* reads are possible

guarantees of the ownership system

- move: owner controls how long the data is valid for

- &T: multiple immutable reads are possible

- **&mut T**: *mutation* is possible and there is only *one* &mut in scope!

bank API

Account

transfer()

Balance

value

Transactions

to_acc   from_acc

amount   execute()

user_name

check_accounts()

```
struct Account {
    user_name: String,
    balance: Balance,
}
```

```
struct Balance {
    value: u64,
}
```

bank API

```
struct Account {
    user_name: String,
    balance: Balance,
}
```

```
struct Balance {
    value: u64,
}
```

```
struct Account {
    user_name: String,
    balance: Balance,
}
```

```
struct Balance {
    value: u64,
}
```

bank API

```rust
struct Transaction<'a> {
    from: &'a mut Account,
    to: &'a mut Account,
    amount: u64,
}

impl<'a> Transaction<'a> {
    fn execute(&mut self) {
        self.from.balance.value -= self.amount;
        self.to.balance.value += self.amount;
    }
}
```

bank API

```
struct Transaction<'a> {
    from: &'a mut Account,
    to: &'a mut Account,
    amount: u64,
}
```

```
impl<'a> Transaction<'a> {
    fn execute(&mut self) {
        self.from.balance.value -= self.amount;
        self.to.balance.value += self.amount;
    }
}
```

bank API

```rust
struct Transaction<'a> {
    from: &'a mut Account,
    to: &'a mut Account,
    amount: u64,
}

impl<'a> Transaction<'a> {
    fn execute(&mut self) {
        self.from.balance.value -= self.amount;
        self.to.balance.value += self.amount;
    }
}
```

bank API

```rust
struct Transaction<'a> {
    from: &'a mut Account,
    to: &'a mut Account,
    amount: u64,
}
```

```rust
impl<'a> Transaction<'a> {
    fn execute(&mut self) {
        self.from.balance.value -= self.amount;
        self.to.balance.value += self.amount;
    }
}
```

bank API

```rust
fn check_accounts(acc: Vec<&Account>) {
    for i in acc {
        println!("{:?}", i);
    }
}
```

```rust
fn transfer(
    mut from: &mut Account,
    mut to: &mut Account,
    amount: u64) {
    Transaction { from, to, amount }.execute();
}
```

bank API

```rust
fn check_accounts(acc: Vec<&Account>) {
    for i in acc {
        println!("{:?}", i);
    }
}
```

```rust
fn transfer(
    mut from: &mut Account,
    mut to: &mut Account,
    amount: u64) {
    Transaction { from, to, amount }.execute();
}
```

bank API

```rust
fn check_accounts(acc: Vec<&Account>) {
    for i in acc {
        println!("{:?}", i);
    }
}
```

```rust
fn transfer(
    mut from: &mut Account,
    mut to: &mut Account,
    amount: u64) {
    Transaction { from, to, amount }.execute();
}
```

bank API

```rust
fn main() {
    let alice_balance = Balance { value: 100 };
    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let bob_balance = Balance { value: 100 };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    check_accounts(vec![&alice_acc, &bob_acc]);

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
    }
    check_accounts(vec![&alice_acc, &bob_acc]);

    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);
}
```

bank API

```rust
fn main() {

    let alice_balance = Balance { value: 100 };
    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };

    let bob_balance = Balance { value: 100 };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };

    check_accounts(vec![&alice_acc, &bob_acc]);

    ...
}
```

bank API

```
fn main() {

    let alice_balance = Balance { value: 100 };
    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };

    let bob_balance = Balance { value: 100 };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };

    check_accounts(vec![&alice_acc, &bob_acc]);


    ...
}
```

bank API

```rust
fn main() {

    let alice_balance = Balance { value: 100 };
    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };

    let bob_balance = Balance { value: 100 };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };

    check_accounts(vec![&alice_acc, &bob_acc]);


    ...
}
```

bank API

```rust
fn main() {

    let alice_balance = Balance { value: 100 };
    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };

    println!("{:?}", alice_balance);

    let bob_balance = Balance { value: 100 };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };

    check_accounts(vec![&alice_acc, &bob_acc]);

    ...
}
```

bank API

```rust
fn main() {

    let alice_balance = Balance { value: 100 };
    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };

    println!("{:?}", alice_balance);

    let bob_balance = Balance { value: 100 };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };

    check_accounts(vec![&alice_acc, &bob_acc]);

    ...
}
```

bank API

```rust
fn main() {

    let alice_balance = Balance { value: 100 };
    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };

    let bob_balance = Balance { value: 100 };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };

    check_accounts(vec![&alice_acc, &bob_acc]);


    ...
}
```

bank API

```rust
fn main() {

    let alice_balance = Balance { value: 100 };
    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };

    let bob_balance = Balance { value: 100 };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };

    check_accounts(vec![&alice_acc, &bob_acc]);

    ...
}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
    }
    check_accounts(vec![&alice_acc, &bob_acc]);

    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
    }
    check_accounts(vec![&alice_acc, &bob_acc]);

    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
    }
    check_accounts(vec![&alice_acc, &bob_acc]);

    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
    } //`transfer` is dropped here
    check_accounts(vec![&alice_acc, &bob_acc]);

    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
    }
    check_accounts(vec![&alice_acc, &bob_acc]);

    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
        transfer.execute();
    }


    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
struct Transaction<'a> {
    from: &'a mut Account,
    to: &'a mut Account,
    amount: u64,
}

impl<'a> Transaction<'a> {
    fn execute(&mut self) {
        self.from.balance.value -= self.amount;
        self.to.balance.value += self.amount;
    }
}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute(); // scope restricted to line
        transfer.execute();
    }


    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute(); // scope restricted to line
        transfer.execute(); // scope restricted to line
    }


    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...


    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
        transfer.execute(); //Works but BAD behavior
    }


    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
struct Transaction<'a> {
    from: &'a mut Account,
    to: &'a mut Account,
    amount: u64,
}
```

```rust
impl<'a> Transaction<'a> {
    fn execute(self) {
        self.from.balance.value -= self.amount;
        self.to.balance.value += self.amount;
    }
}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute(); // Transaction moved and consumed by execute()
        transfer.execute();
        ^^^^^^^^ value used here after move
    }


    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
    }
    check_accounts(vec![&alice_acc, &bob_acc]);

    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
    }
    check_accounts(vec![&alice_acc, &bob_acc]);

    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

bank API

```rust
fn main() {

    let mut alice_acc = Account { user_name: String::from("alice"), balance: alice_balance };
    let mut bob_acc = Account { user_name: String::from("bob"), balance: bob_balance };
    ...

    { /* TRANSACTION 1: Restrict the scope to the block. */
        let mut transfer = Transaction { from: &mut alice_acc, to: &mut bob_acc, amount: 10 };
        transfer.execute();
    }
    check_accounts(vec![&alice_acc, &bob_acc]);

    /* TRANSACTION 2: Restricted scope because Transaction is not assigned. */
    transfer(&mut alice_acc, &mut bob_acc, 10);
    check_accounts(vec![&alice_acc, &bob_acc]);

}
```

productivity tips (extra)

tools to becoming productive

- compiler driven development (CDD)

  - **cargo check** is faster than **cargo run**

  - **cargo-watch** (cargo check on file change)

- clippy (linter)

- cargo fmt (formatter)

references and thanks

- coworkers at iHeart Radio

- the Rust community

- Splash photo by Andrew Branch on Unsplash

- https://rustbyexample.com/

- https://doc.rust-lang.org/stable/book/first-edition/

- https://doc.rust-lang.org/stable/book/second-edition/

- https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap

- https://stackoverflow.com/questions/373419/whats-the-difference-between-passing-by-reference-vs-passing-by-value

- https://stackoverflow.com/questions/24253344/move-vs-copy-in-rust?answertab=active#tab-top

- https://blog.rust-lang.org/2017/03/16/Rust-1.16.html

Ownership in Rust:

the core principle for API design

toidiu.com

github.com/toidiu

github.com/toidiu/talk_rust_ownership