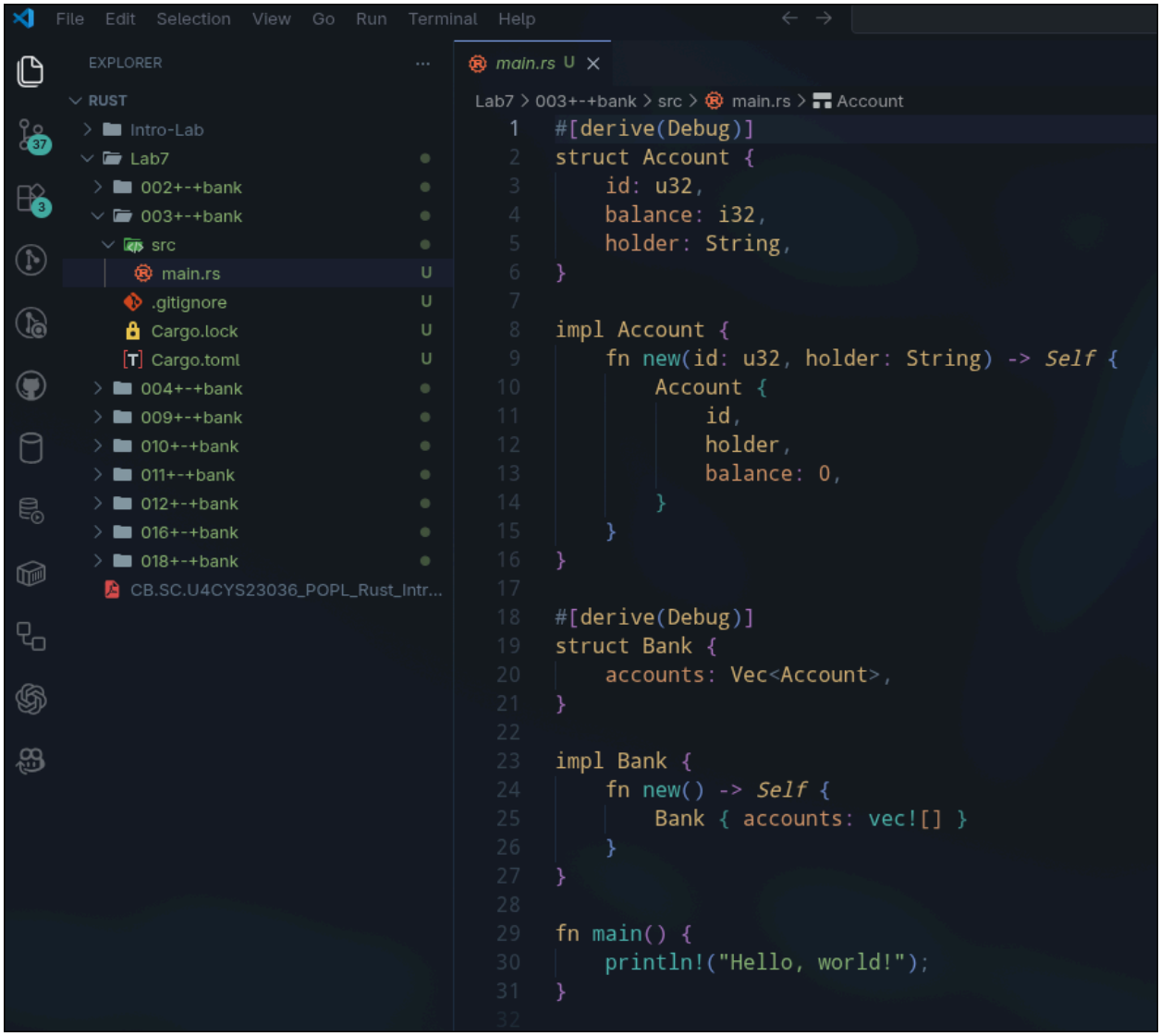
Praneesh R V
CB.SC.U4CYS23036

Lab - 7 - Ownership, References and Structures in RUST

1)
002+-+ Bank

This program defines two data structures: Account, which stores an id, balance, and account holder name, and Bank, which contains a list of accounts in a vector. Both structs derive Debug, so they can be printed easily for debugging. The main function currently does not create or use them and simply prints “Hello, world!”

003 +-+Bank




```
warning: 4 warnings emitted

praneesh@ShadowEternity ~/Praneesh/Academics/Sem6/POPL/Rust/Lab7/003+-+bank/src ‹main●›
$ ./main
Hello, world!
```

This program defines two structs: Account and Bank. Both have new constructor methods—Account::new creates an account with a zero balance, and Bank::new creates a bank with an empty list of accounts. The Debug trait allows them to be printed if needed. However, in main,

these structures are not used, and the program simply prints “Hello, World!”

004+-+ Bank

```rust
#[derive(Debug)]
struct Account {
    id: u32,
    balance: i32,
    holder: String,
}

impl Account {
    fn new(id: u32, holder: String) -> Self {
        Account {
            id,
            holder,
            balance: 0,
        }
    }
}

#[derive(Debug)]
struct Bank {
    accounts: Vec<Account>,
}

impl Bank {
    fn new() -> Self {
        Bank { accounts: vec![] }
    }
}

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

```rust
}

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

    print_account(&account);
    print_account(&account);

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

The program defines Account and Bank structs, along with constructor methods to create them. A print_account function borrows an account using a reference and prints it, so ownership is not moved. In main, an account is created and passed to the function twice, which works because borrowing allows multiple reads. The bank is created but not used further.

009 +-+ Bank

```rust
#[derive(Debug)]
struct Account {
    id: u32,
    balance: i32,
    holder: String,
}
```

```
impl Account {
    fn new(id: u32, holder: String) -> Self {
        Account {
            id,
            holder,
            balance: 0,
        }
    }
}

#[derive(Debug)]
struct Bank {
    accounts: Vec<Account>,
}

impl Bank {
    fn new() -> Self {
        Bank { accounts: vec![] }
    }
}

fn print_account(account: Account) -> Account {
    println!("{:#?}", account);
    account
}

fn print_holder(holder: String) {
    println!("{}", holder);
}

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

```
    account = print_account(account);
    account = print_account(account);

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

```
warning: 4 warnings emitted

praneesh@ShadowEternity ~/Praneesh/Academics/Sem6/POPL/Rust/Lab7/009+-+bank/src ‹main●›
$ ./main
Account {
    id: 1,
    balance: 0,
    holder: "me",
}
Account {
    id: 1,
    balance: 0,
    holder: "me",
}
Account {
    id: 1,
    balance: 0,
    holder: "me",
}
```

This Rust program demonstrates the **move-and-return** pattern to manage memory ownership. In Rust, passing a variable like `Account` to a function normally moves ownership, making the original variable unusable. To circumvent this, the `print_account` function takes ownership, performs its task, and then returns the `Account` back to the caller. By reassigning this returned value to the original variable in `main`, the program effectively "borrows" the data through a full transfer cycle, allowing the same object to be processed multiple times without being dropped from memory.

010 +-+ Bank

```
#[derive(Debug)]
struct Account {
    id: u32,
    balance: i32,
```

```
    holder: String,
}

impl Account {
    fn new(id: u32, holder: String) -> Self {
        Account {
            id,
            holder,
            balance: 0,
        }
    }
}

#[derive(Debug)]
struct Bank {
    accounts: Vec<Account>,
}

impl Bank {
    fn new() -> Self {
        Bank { accounts: vec![] }
    }
}

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);
}
```

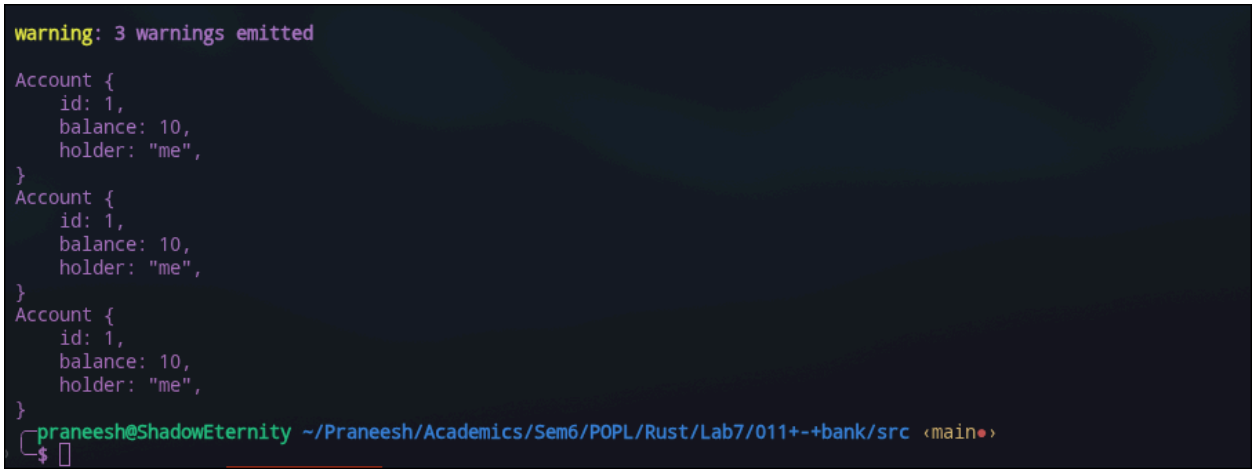
```
warning: 3 warnings emitted

praneesh@ShadowEternity ~/Praneesh/Academics/Sem6/POPL/Rust/Lab7/010+-+bank/src (main)
$ ./main
Account {
    id: 1,
    balance: 0,
    holder: "me",
}
Account {
    id: 1,
    balance: 0,
    holder: "me",
}
praneesh@ShadowEternity ~/Praneesh/Academics/Sem6/POPL/Rust/Lab7/010+-+bank/src (main)
```

The program defines Account and Bank structs, with a constructor to create a new account with a zero balance. The function print_account borrows an account by reference and prints it, so ownership is not transferred. In main, the account is borrowed, printed through the function, and then printed again directly. Because only borrowing occurs, the original value remains valid

011+-+Bank

```
#[derive(Debug)]
struct Account {
    id: u32,
    balance: i32,
    holder: String,
}

impl Account {
    fn new(id: u32, holder: String) -> Self {
        Account {
            id,
```

```
            holder,
            balance: 0,
        }
    }
}

#[derive(Debug)]
struct Bank {
    accounts: Vec<Account>,
}

impl Bank {
    fn new() -> Self {
        Bank { accounts: vec![] }
    }
}

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

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

    account.balance = 10;

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

    print_account(account_ref1);
    print_account(account_ref2);

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



The program defines Account and Bank structs and provides constructors to create them. In main, an account is created and its balance is updated before any borrowing occurs. Two immutable references to the account are then passed to print_account, which prints the details without taking ownership. Because only immutable borrows are used, the account can still be printed again afterward

012+-+bank

```rust
#[derive(Debug)]
struct Account {
    id: u32,
    balance: i32,
    holder: String,
}

impl Account {
    fn new(id: u32, holder: String) -> Self {
        Account {
            id,
            holder,
            balance: 0,
        }
```

```rust
    }
}

#[derive(Debug)]
struct Bank {
    accounts: Vec<Account>,
}

impl Bank {
    fn new() -> Self {
        Bank { accounts: vec![] }
    }
}

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

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

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

    let account_ref = &mut account;

    account_ref.balance = 10;

    change_account(account_ref);

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

```
warning: 4 warnings emitted

Account {
    id: 1,
    balance: 10,
    holder: "me",
}
┌─praneesh@ShadowEternity ~/Praneesh/Academics/Sem6/POPL/Rust/Lab7/012+-+bank/src ‹main●›
└─$
```

The program defines Account and Bank structs with constructors for creating instances. A function change_account takes a mutable reference to an account and updates its balance, while print_account can display account details using an immutable reference. In main, the account is mutably borrowed and modified through the function. After the borrow ends, the account is printed, showing the updated value.

016+-+bank

```rust
#[derive(Debug)]
struct Account {
   id: u32,
   balance: i32,
   holder: String,
}

impl Account {
   fn new(id: u32, holder: String) -> Self {
       Account {
           id,
           holder,
           balance: 0,
       }
   }
}

#[derive(Debug)]
struct Bank {
   accounts: Vec<Account>,
}
```

```rust
impl Bank {
    fn new() -> Self {
        Bank { accounts: vec![] }
    }

    fn add_account(&mut self, account: Account) {
        self.accounts.push(account);
    }
}

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

    bank.add_account(account);

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

```
warning: 1 warning emitted

Bank {
    accounts: [
        Account {
            id: 1,
            balance: 0,
            holder: "me",
        },
    ],
}
praneesh@ShadowEternity ~/Praneesh/Academics/Sem6/POPL/Rust/Lab7/016+-+bank/src ‹main●›
$
```

The program defines Account and Bank structs along with constructors to create them. The Bank implementation includes an add_account method that takes a mutable reference to the bank and moves an account into its vector. In main, a bank and an account are created, the account is added to the bank, and the updated bank is printed using the Debug formatter

018+-+bank

```rust
#[derive(Debug)]
struct Account {
    id: u32,
    balance: i32,
    holder: String,
}

impl Account {
    fn new(id: u32, holder: String) -> Self {
        Account {
            id,
            holder,
            balance: 0,
        }
    }

    fn summary(&self) -> String {
        format!("{} has a balance {}", self.holder, self.balance)
    }

    fn deposit(&mut self, amount: i32) -> i32 {
        self.balance += amount;
        self.balance
    }

    fn withdraw(&mut self, amount: i32) -> i32 {
        self.balance -= amount;
        self.balance
    }
}

#[derive(Debug)]
struct Bank {
```

```
    accounts: Vec<Account>,
}

impl Bank {
    fn new() -> Self {
        Bank { accounts: vec![] }
    }

    fn add_account(&mut self, account: Account) {
        self.accounts.push(account);
    }

    fn total_balance(&self) -> i32 {
        self.accounts.iter().map(|account| account.balance).sum()
    }

    fn summary(&self) -> Vec<String> {
        self.accounts
            .iter()
            .map(|account| account.summary())
            .collect::<Vec<String>>()
    }
}

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

    account.deposit(500);
    account.withdraw(250);

    bank.add_account(account);

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

```
    println!("{}", bank.total_balance());
}
```

```
warning: 1 warning emitted

[
    "me has a balance 250",
]
250
praneesh@ShadowEternity ~/Praneesh/Academics/Sem6/POPL/Rust/Lab7/018+-+bank/src ‹main●›
$
```

This code implements a basic banking system that leverages Rust's ownership and collection traits to manage financial data. The `Account` struct encapsulates user data and provides methods for mutating its state via `deposit` and `withdraw`, while the `Bank` struct manages a collection of these accounts using a `Vec`. A key transition occurs in `main` when `bank.add_account(account)` is called: ownership of the `account` instance is moved into the `Bank` structure, meaning the `account` variable in `main` is no longer accessible. The program then utilizes functional programming patterns, like `iter()`, `map()`, and `sum()`, to calculate the total liquidity and generate a list of account summaries across the entire bank.

2) Solve the following

1, Code

RUN ▶ ··· DEBUG STABLE

```
#[derive(Debug)]
struct Account {
id: u32,
balance: i32,
holder: String,
}
impl Account {
fn new(id: u32, holder: String) -> Self {
Account {
id,
holder,
balance: 0,
}
}
}
fn take_account(account: Account) {
println!("{:#?}", account);
}
fn main() {
let account = Account::new(1, String::from("me"));
take_account(account);
}
```

Output:

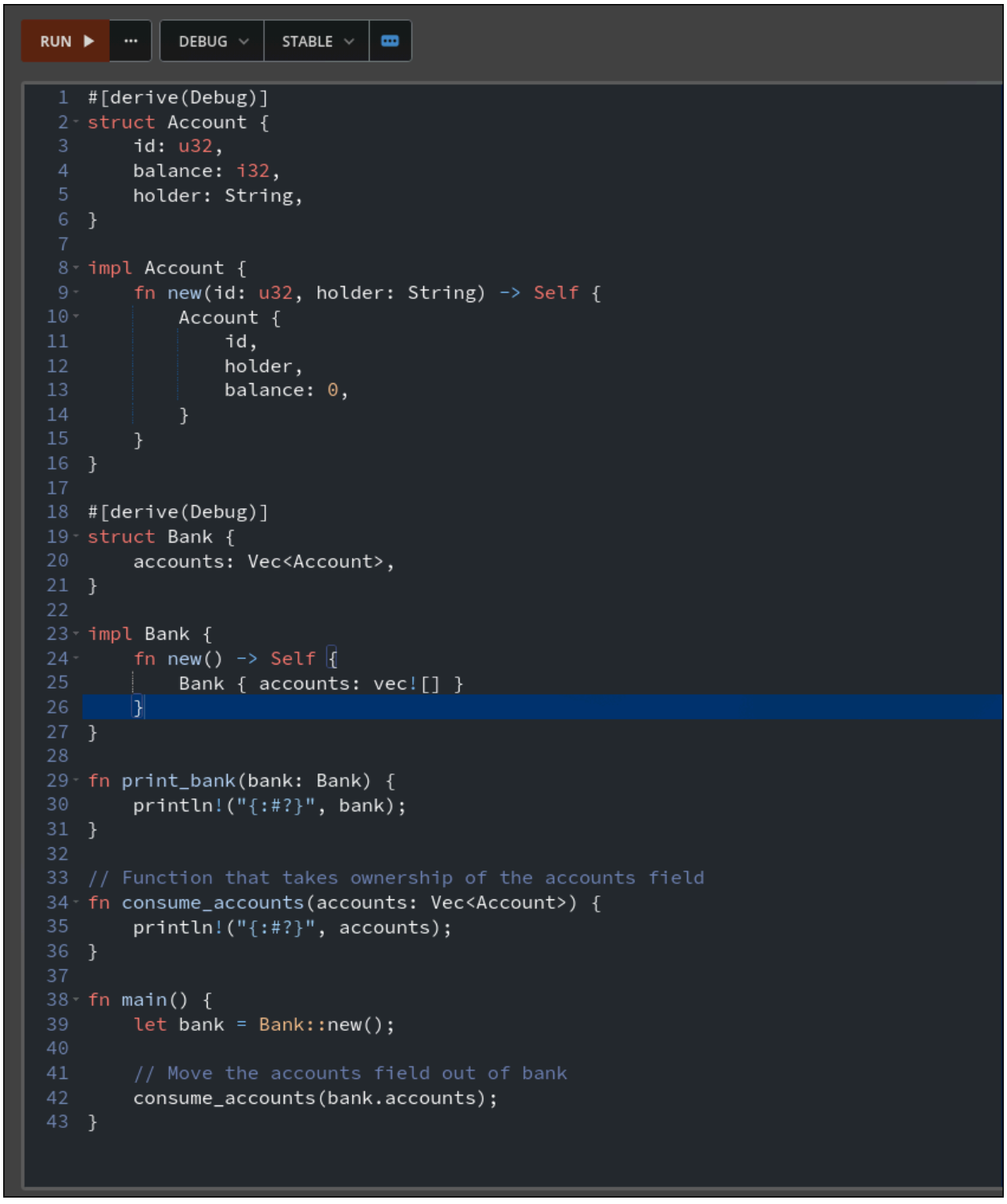
Standard Output

```
Account {
    id: 1,
    balance: 0,
    holder: "me",
}
```

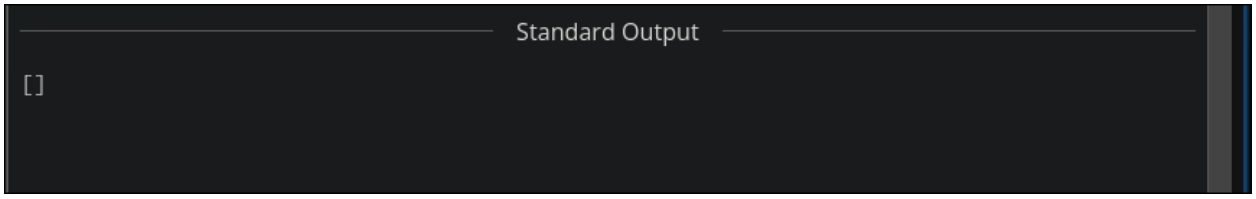
Question: Will you be able to call the function twice with the 'account' variable?

No , because ownership moved into the function on the first call

2, Code:



Output:



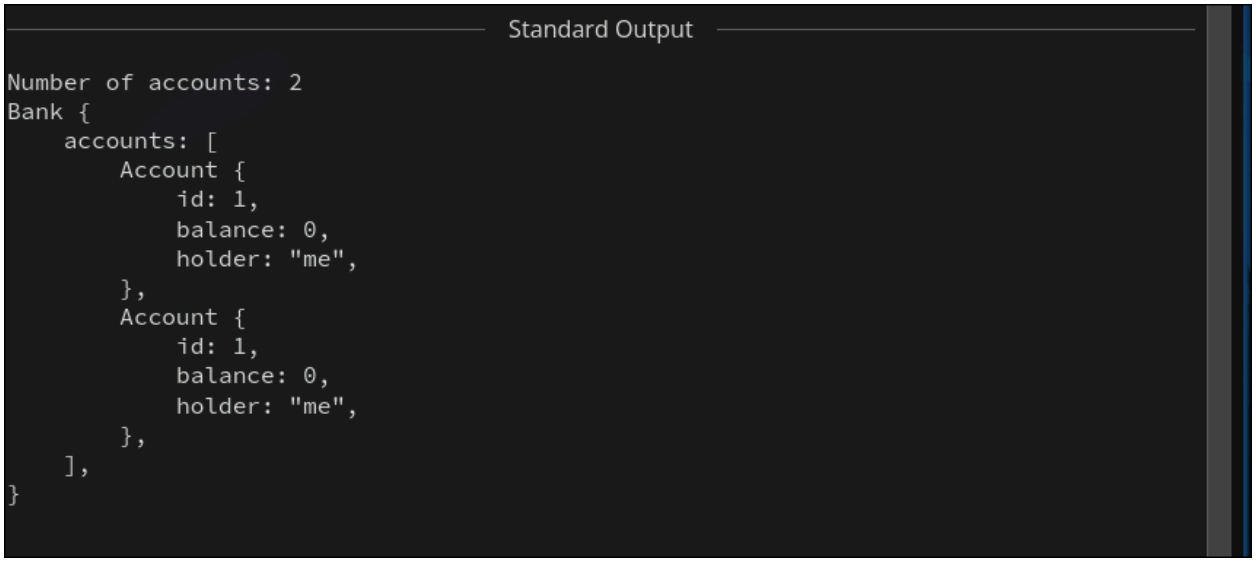
Question: When your function + print_bank run, do you think you'll end up getting an error? If so, what
error do you think you'd see?

Yes, you would get an error. When the function takes ownership of bank.accounts, that field is moved out
of bank, which makes the bank value partially moved and no longer fully usable. If you then try to call
print_bank(bank), Rust will produce an error saying that you are using a partially moved value, typically
shown as “use of partially moved value bank,” because the accounts field has already been transferred and
the original bank can’t be used anymore.

3,
Code:

```rust
#[derive(Debug)]
struct Account {
id: u32,
balance: i32,
holder: String,
}
impl Account {
fn new(id: u32, holder: String) -> Self {
Account {
id,
holder,
balance: 0,
}
}
}
#[derive(Debug)]
struct Bank {
accounts: Vec<Account>,
}
impl Bank {
fn new() -> Self {
Bank { accounts: vec![] }
}
}
fn print_num_accounts(bank: &Bank) {
println!("Number of accounts: {}", bank.accounts.len());
}

fn main() {
let mut bank = Bank::new();
let account1 = Account::new(1, String::from("me"));
let account2 = Account::new(1, String::from("me"));
bank.accounts.push(account1);
bank.accounts.push(account2);
print_num_accounts(&bank);
println!("{:#?}", bank);
}
```

Output:


Standard Output
Number of accounts: 2
Bank {
accounts: [
Account {
id: 1,
balance: 0,
holder: "me",
},
Account {
id: 1,
balance: 0,
holder: "me",
},
],
}

4,
Code:

```rust
#[derive(Debug)]
struct Account {
    id: u32,
    balance: i32,
    holder: String,
}

impl Account {
    fn new(id: u32, holder: String) -> Self {
        Account {
            id,
            holder,
            balance: 0,
        }
    }
}

#[derive(Debug)]
struct Bank {
    accounts: Vec<Account>,
}

impl Bank {
    fn new() -> Self {
        Bank { accounts: vec![] }
    }
}

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);
}
```

Output:

Standard Output

```
Bank {
    accounts: [
        Account {
            id: 1,
            balance: 0,
            holder: "me",
        },
    ],
}
```