Rust – Ownership & Structure

Week 10

SCHOOL OF SCIENCE AND TECHNOLOGY
PAN-ATLANTIC UNIVERSITY

Rust - Ownership

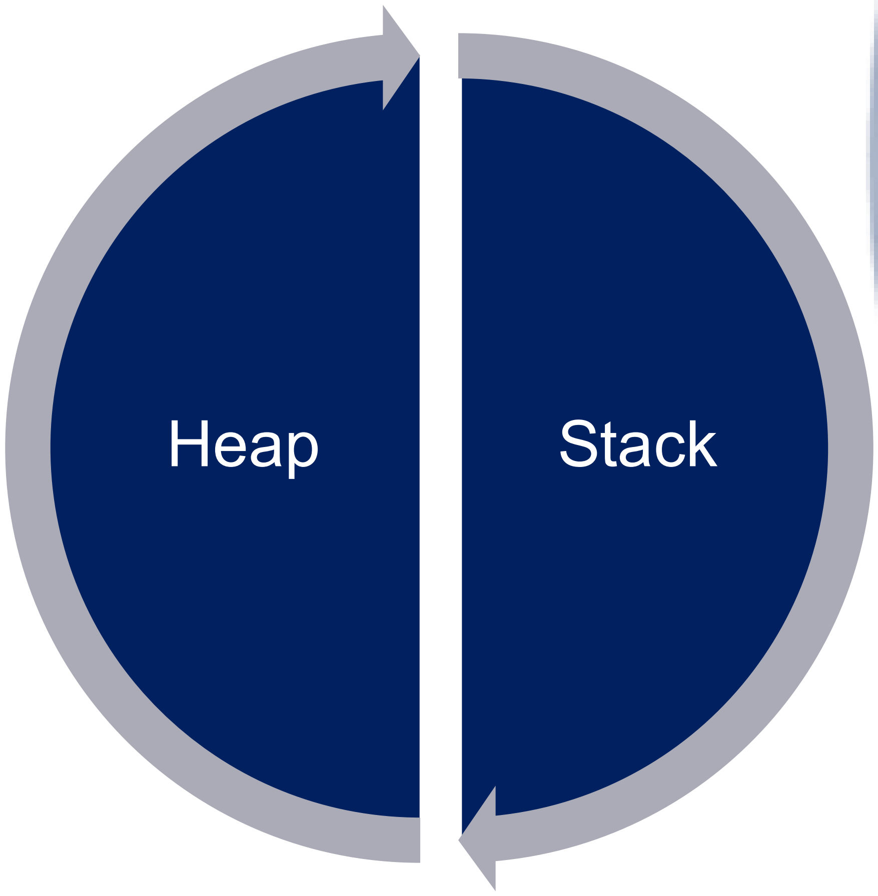
Each value in Rust has a variable that is called owner of the value. Every data stored in Rust will have an owner associated with it.

- For example, in the syntax − let age = 30, age is the owner of the value 30.

Each data can have only one owner at a time.

Two variables cannot point to the same memory location. The variables will always be pointing to different memory locations.

Memory Allocation



Stack

A stack follows a last-in-first-out (LIFO) order.

Stack stores data values for which the size is known at compile time.

- For example, a variable of fixed size ***i32*** is a candidate for stack allocation.
- Its size is known at compile time.
- All scalar types can be stored in stack as the size is fixed.

Heap

The heap memory stores data values the size of which is unknown at compile time.

It is used to store dynamic data.

Simply put, a heap memory is allocated to data values that may change throughout the life cycle of the program.

The heap is an area in the memory which is less organized when compared to stack.

Transferring Ownership

Assigning value of one variable to another variable.

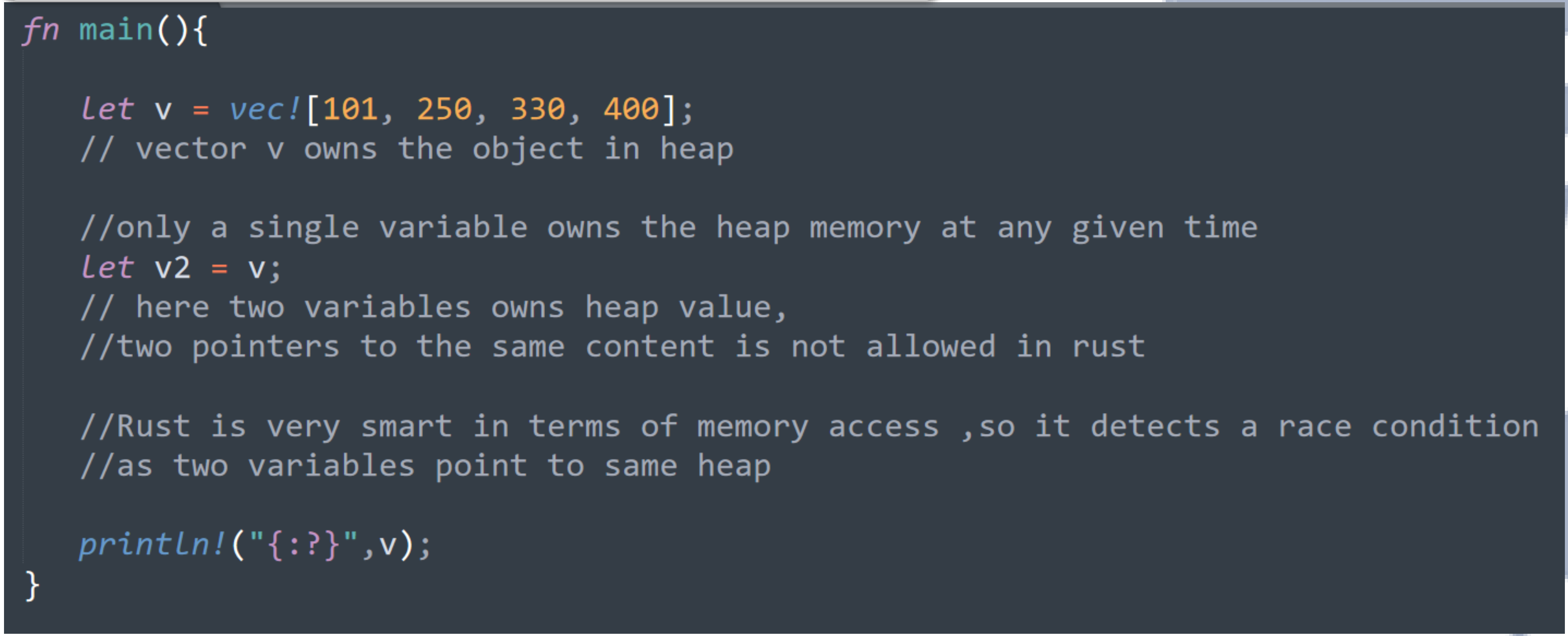
Passing value to a function.

Returning value from a function.

Assigning value of one variable to another variable

The key selling point of Rust as a language is its memory safety. Memory safety is achieved by tight control on who can use what and when restrictions.

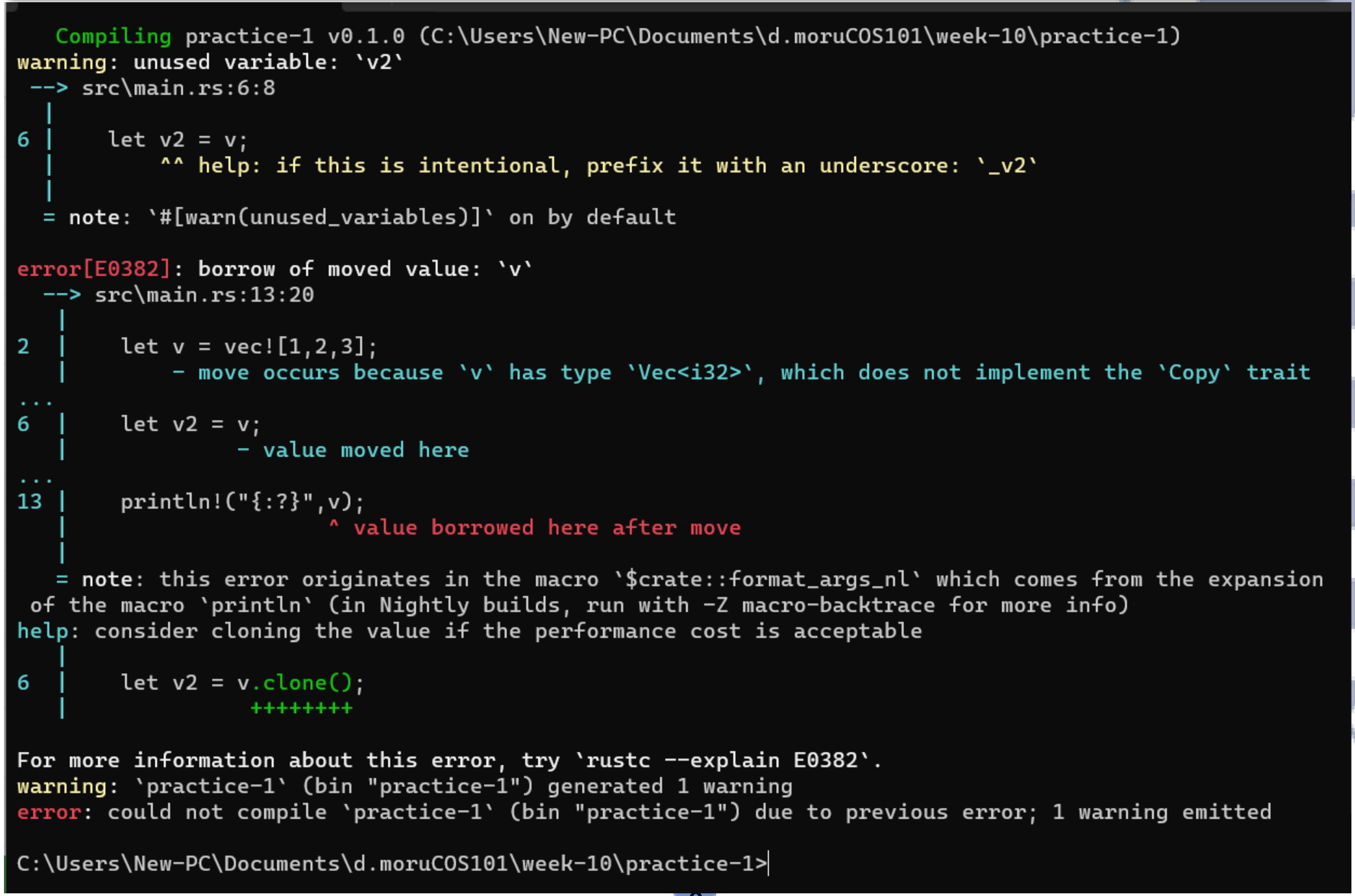
Practice 1: week-10/practice_1/src/main.rs



```
fn main(){

    let v = vec![101, 250, 330, 400];
    // vector v owns the object in heap

    //only a single variable owns the heap memory at any given time
    let v2 = v;
    // here two variables owns heap value,
    //two pointers to the same content is not allowed in rust

    //Rust is very smart in terms of memory access ,so it detects a race condition
    //as two variables point to same heap

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

Assigning value of one variable to another variable

Practice 1: week-10/practice_1/src/main.rs (ERROR MESSAGE)

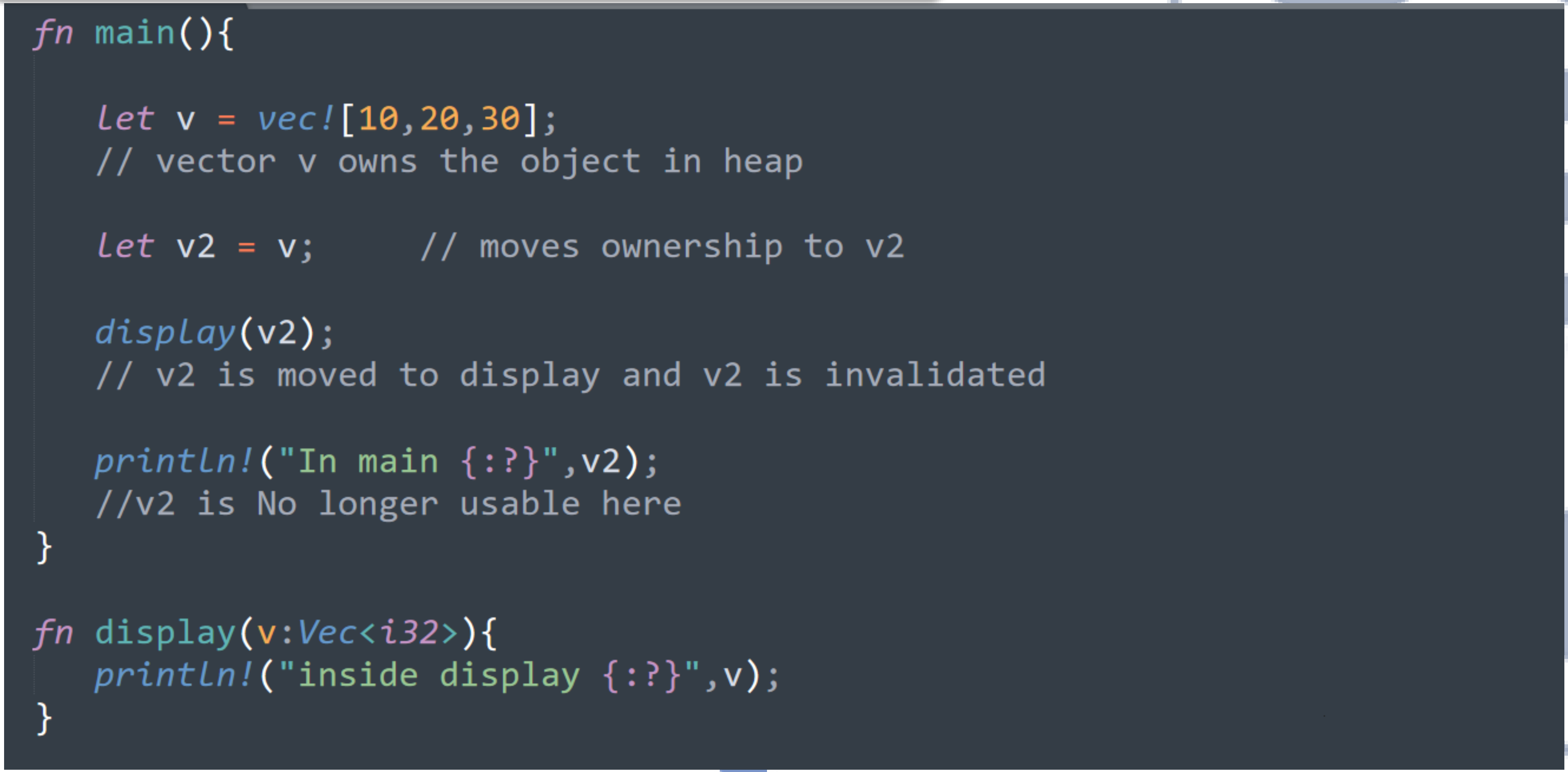


```
   Compiling practice-1 v0.1.0 (C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-1)
warning: unused variable: `v2`
 --> src\main.rs:6:8
  |
6 |    let v2 = v;
  |        ^^ help: if this is intentional, prefix it with an underscore: `_v2`
  |
  = note: `#[warn(unused_variables)]` on by default

error[E0382]: borrow of moved value: `v`
  --> src\main.rs:13:20
   |
2  |    let v = vec![1,2,3];
   |        - move occurs because `v` has type `Vec<i32>`, which does not implement the `Copy` trait
...
6  |    let v2 = v;
   |             - value moved here
...
13 |    println!("{:?}",v);
   |                    ^ value borrowed here after move
   |
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion
 of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
   |
6  |    let v2 = v.clone();
   |              ++++++++

For more information about this error, try `rustc --explain E0382`.
warning: `practice-1` (bin "practice-1") generated 1 warning
error: could not compile `practice-1` (bin "practice-1") due to previous error; 1 warning emitted

C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-1>
```

Passing value to a function.

The ownership of a value also changes when we pass an object in the heap to a function.

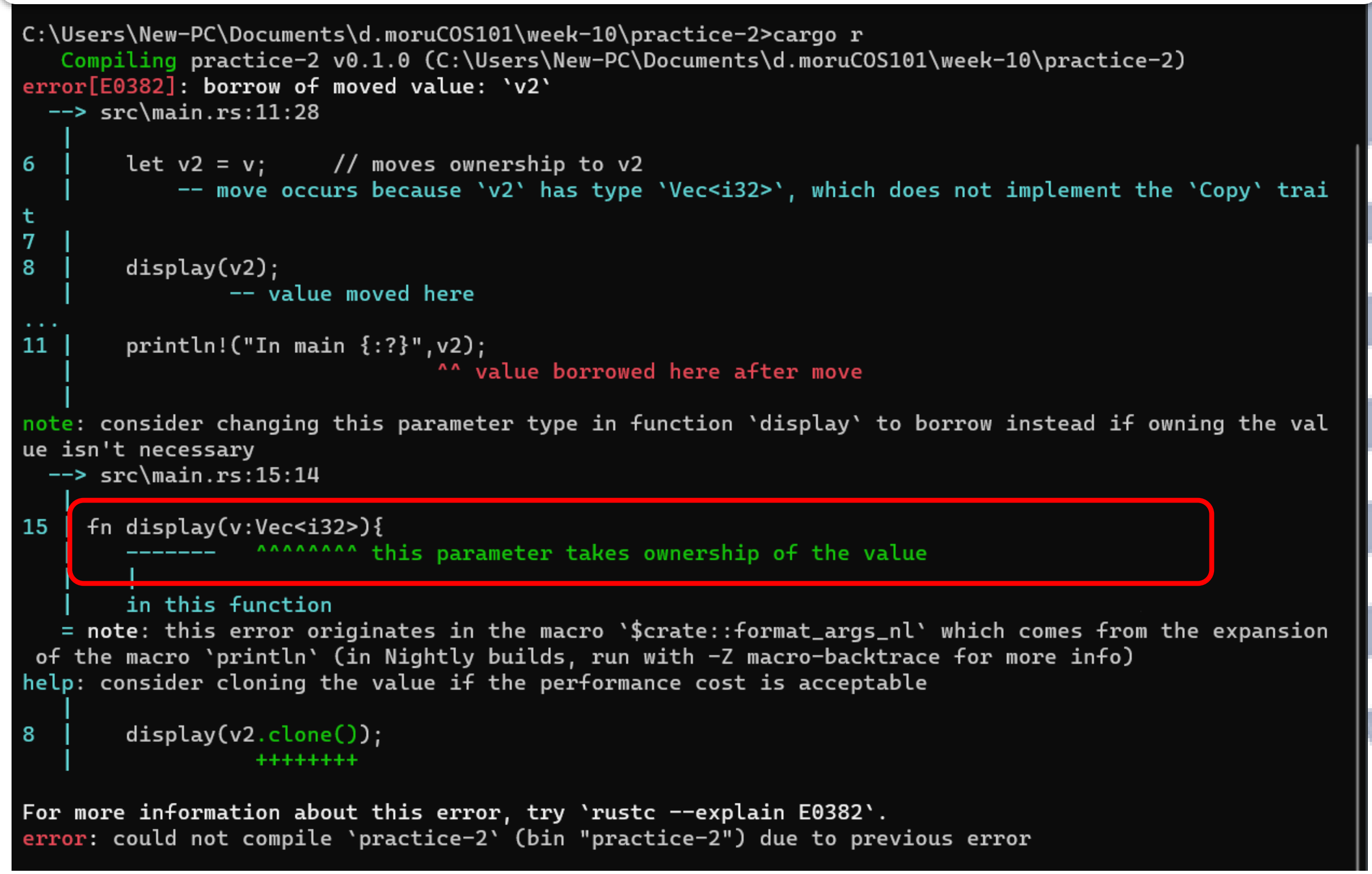
Practice 2: week-10/practice_2/src/main.rs



```rust
fn main(){

    let v = vec![10,20,30];
    // vector v owns the object in heap

    let v2 = v;     // moves ownership to v2

    display(v2);
    // v2 is moved to display and v2 is invalidated

    println!("In main {:?}",v2);
    //v2 is No longer usable here
}

fn display(v:Vec<i32>){
    println!("inside display {:?}",v);
}
```

Passing value to a function.

Practice 2: week-10/practice_2/src/main.rs (ERROR MESSAGE)

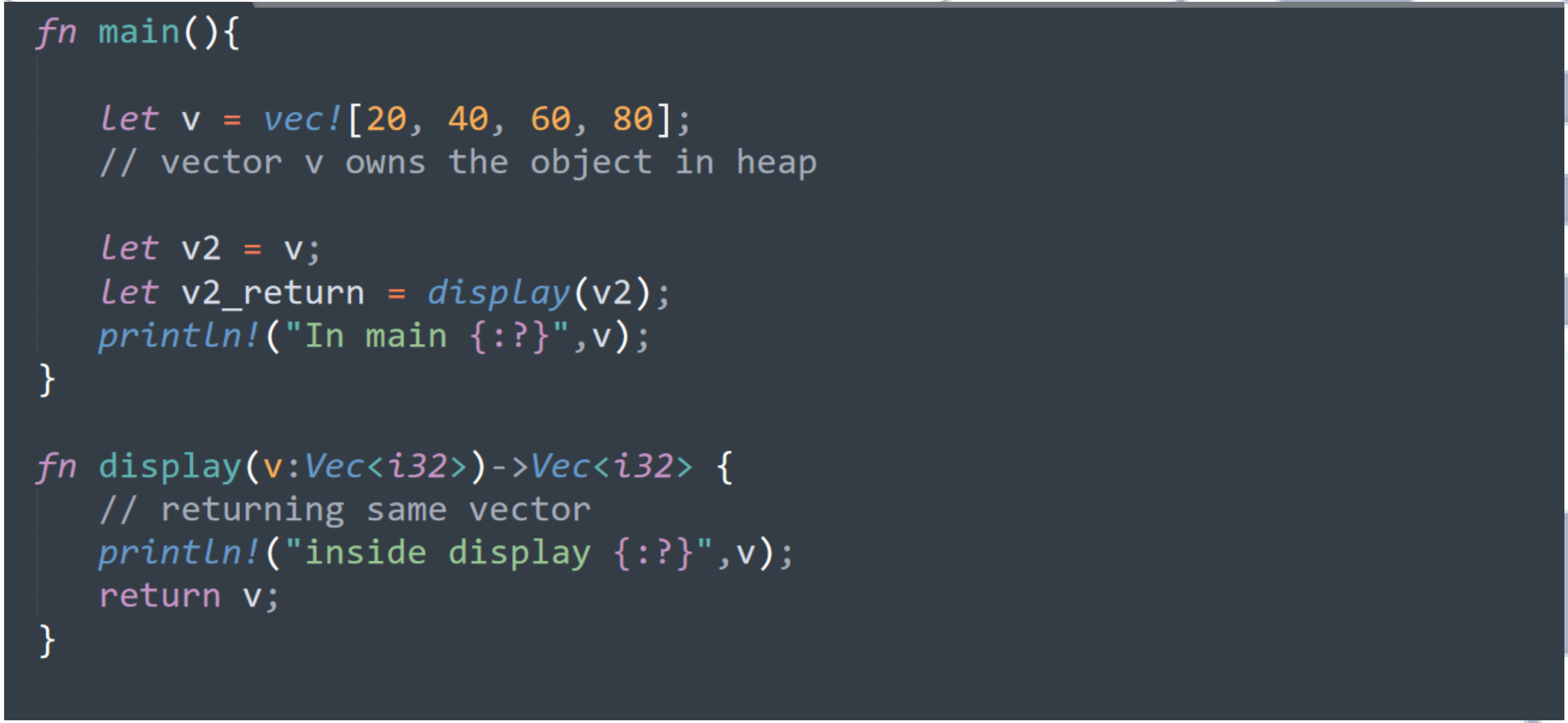


```
C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-2>cargo r
   Compiling practice-2 v0.1.0 (C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-2)
error[E0382]: borrow of moved value: `v2`
  --> src\main.rs:11:28
   |
6  |     let v2 = v;      // moves ownership to v2
   |         -- move occurs because `v2` has type `Vec<i32>`, which does not implement the `Copy` trai
t
7  |
8  |     display(v2);
   |             -- value moved here
...
11 |     println!("In main {:?}",v2);
   |                             ^^ value borrowed here after move
   |
note: consider changing this parameter type in function `display` to borrow instead if owning the val
ue isn't necessary
  --> src\main.rs:15:14
   |
15 | fn display(v:Vec<i32>){
   |    -------   ^^^^^^^^ this parameter takes ownership of the value
   |    |
   |    in this function
   = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion
 of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
   |
8  |     display(v2.clone());
   |               ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `practice-2` (bin "practice-2") due to previous error
```

Returning value from a function

Ownership passed to the function will be invalidated as function execution completes.

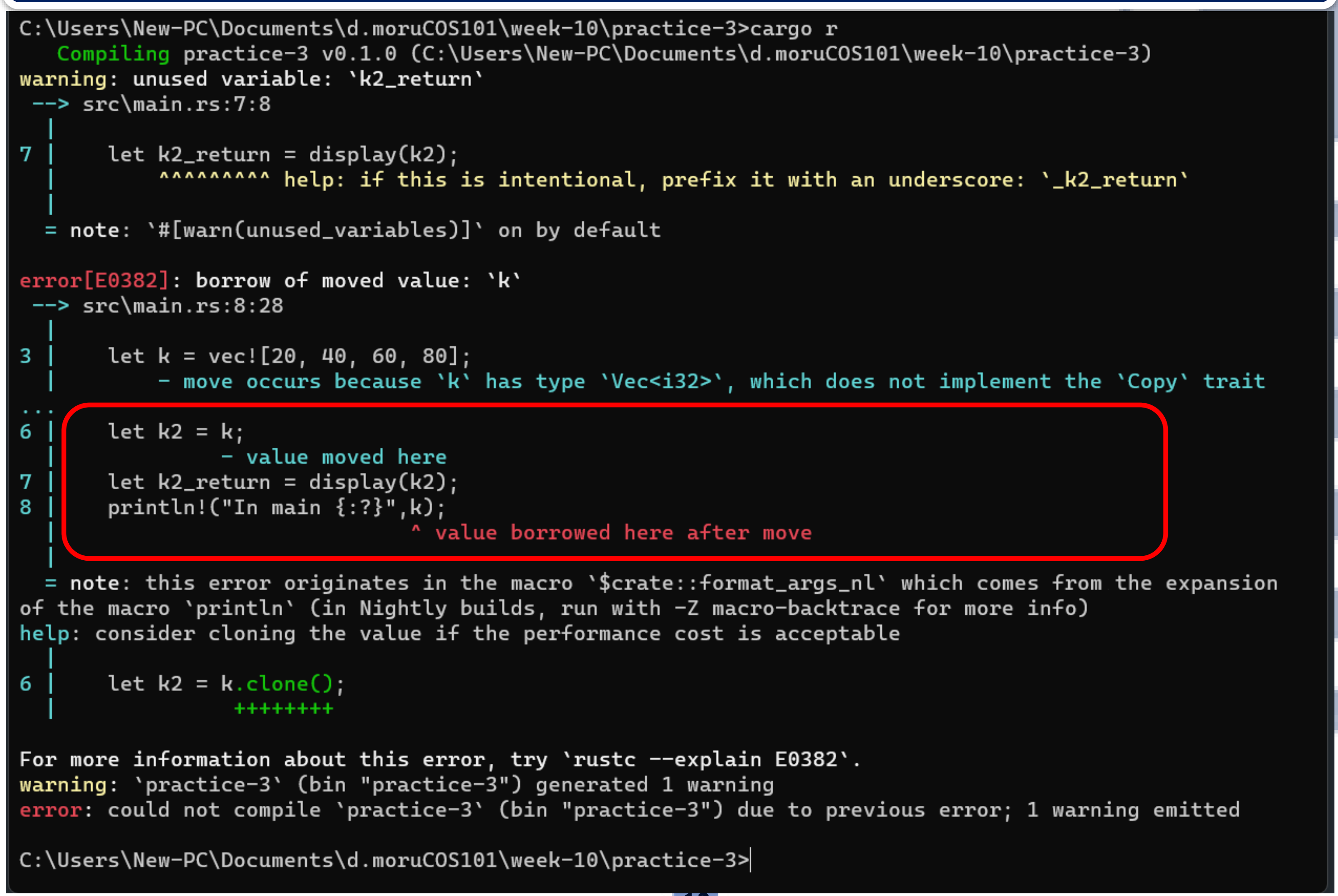
Practice 3: week-10/practice_3/src/main.rs



```rust
fn main(){

    let v = vec![20, 40, 60, 80];
    // vector v owns the object in heap

    let v2 = v;
    let v2_return = display(v2);
    println!("In main {:?}",v);
}

fn display(v:Vec<i32>)->Vec<i32> {
    // returning same vector
    println!("inside display {:?}",v);
    return v;
}
```

Returning value from a function

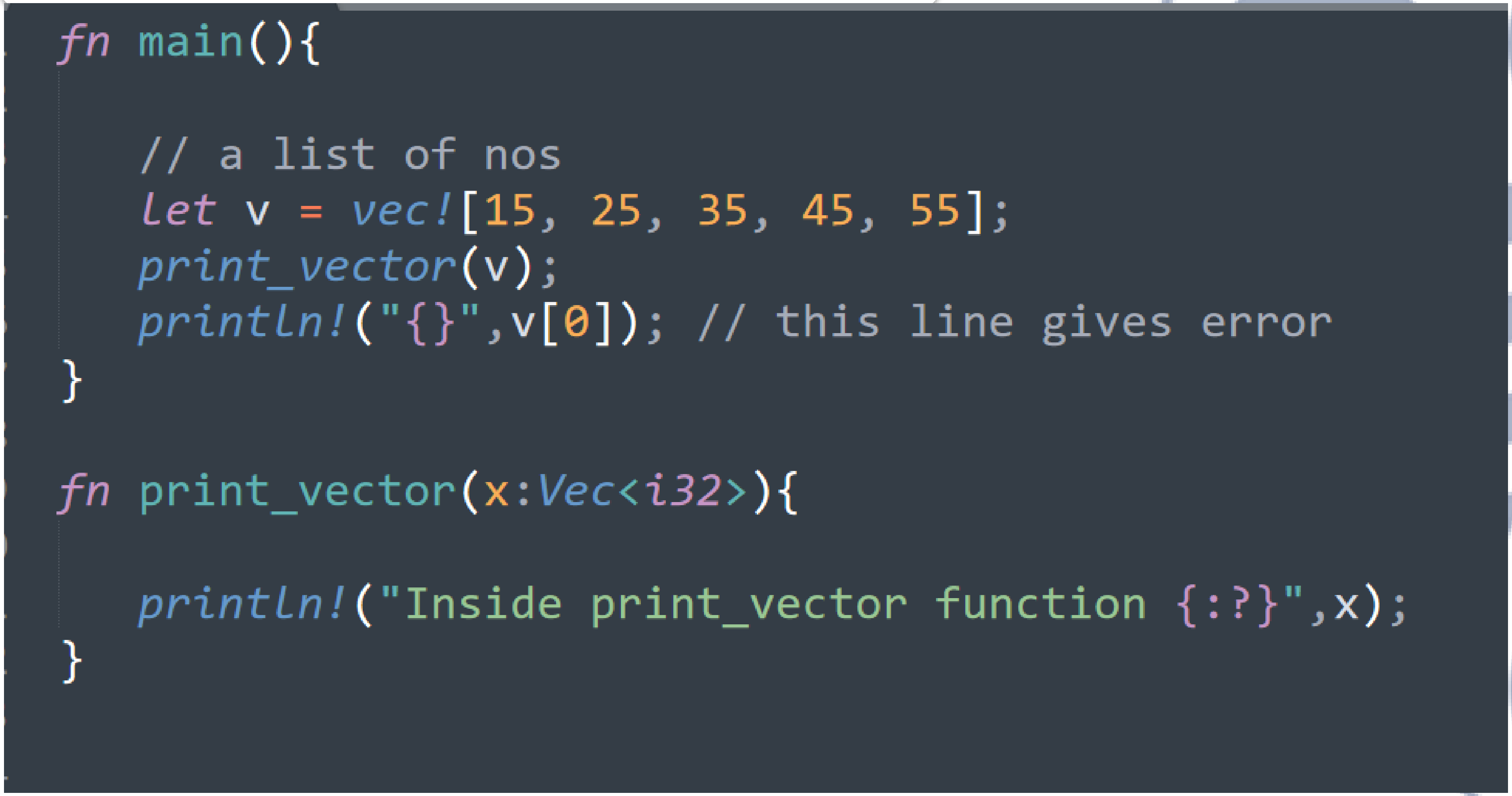
Practice 3: *week-10/practice_3/src/main.rs (ERROR MESSAGE)*



```
C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-3>cargo r
   Compiling practice-3 v0.1.0 (C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-3)
warning: unused variable: `k2_return`
 --> src\main.rs:7:8
  |
7 |    let k2_return = display(k2);
  |        ^^^^^^^^^ help: if this is intentional, prefix it with an underscore: `_k2_return`
  |
  = note: `#[warn(unused_variables)]` on by default

error[E0382]: borrow of moved value: `k`
 --> src\main.rs:8:28
  |
3 |    let k = vec![20, 40, 60, 80];
  |        - move occurs because `k` has type `Vec<i32>`, which does not implement the `Copy` trait
...
6 |    let k2 = k;
  |             - value moved here
7 |    let k2_return = display(k2);
8 |    println!("In main {:?}",k);
  |                            ^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion
of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
  |
6 |    let k2 = k.clone();
  |              ++++++++

For more information about this error, try `rustc --explain E0382`.
warning: `practice-3` (bin "practice-3") generated 1 warning
error: could not compile `practice-3` (bin "practice-3") due to previous error; 1 warning emitted

C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-3>
```

Consider the following…

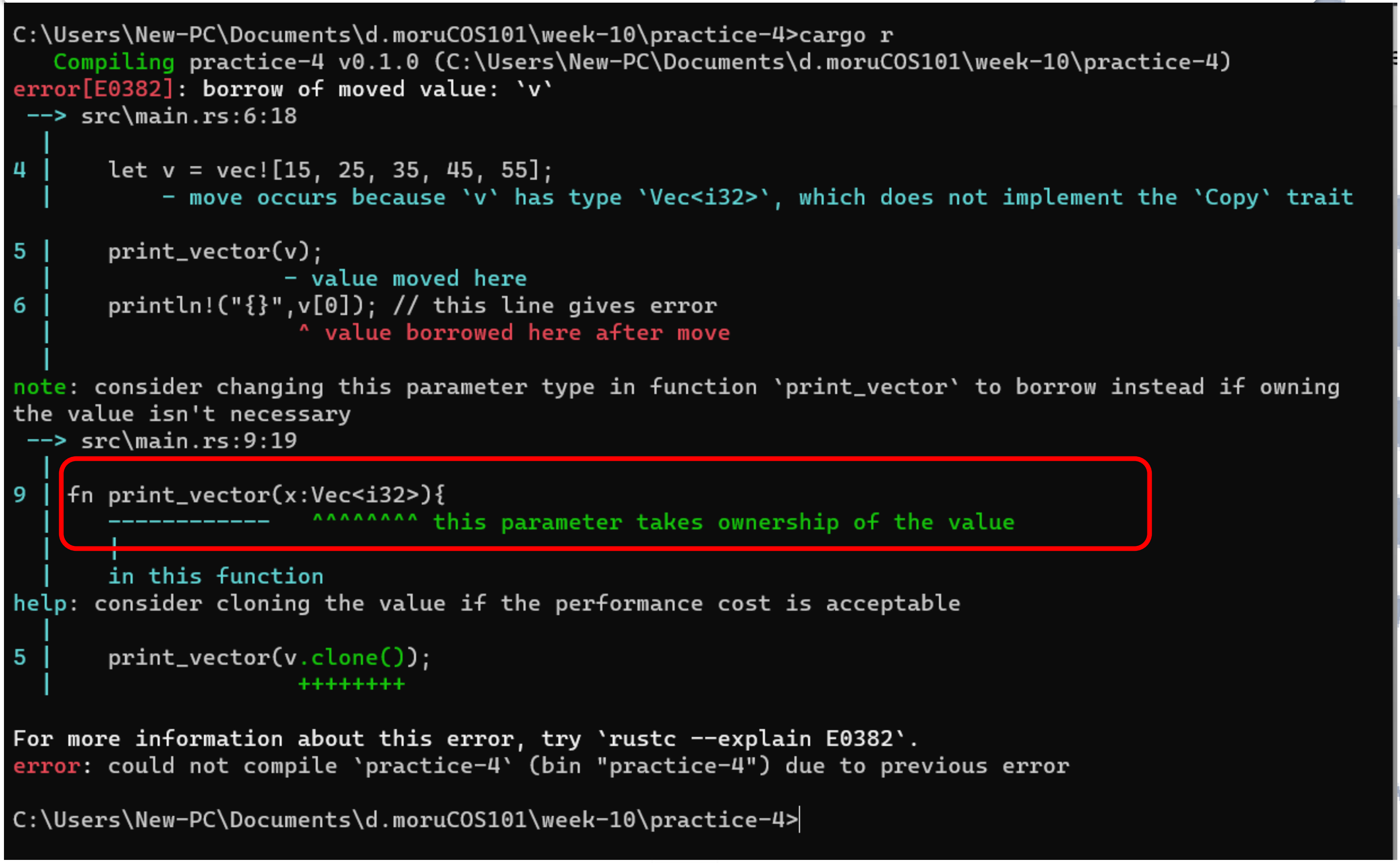
Practice 14: *week-10/practice_4/src/main.rs*



```rust
fn main(){

    // a list of nos
    let v = vec![15, 25, 35, 45, 55];
    print_vector(v);
    println!("{}",v[0]); // this line gives error
}

fn print_vector(x:Vec<i32>){

    println!("Inside print_vector function {:?}",x);
}
```

Output of the code…

Practice 4: ***week-10/practice_4/src/main.rs (ERROR MESSAGE)***



```
C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-4>cargo r
   Compiling practice-4 v0.1.0 (C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-4)
error[E0382]: borrow of moved value: `v`
 --> src\main.rs:6:18
  |
4 |     let v = vec![15, 25, 35, 45, 55];
  |         - move occurs because `v` has type `Vec<i32>`, which does not implement the `Copy` trait

5 |     print_vector(v);
  |                  - value moved here
6 |     println!("{}",v[0]); // this line gives error
  |                   ^ value borrowed here after move
  |
note: consider changing this parameter type in function `print_vector` to borrow instead if owning
the value isn't necessary
 --> src\main.rs:9:19
  |
9 | fn print_vector(x:Vec<i32>){
  |    ------------   ^^^^^^^^ this parameter takes ownership of the value
  |    |
  |    in this function
help: consider cloning the value if the performance cost is acceptable
  |
5 |     print_vector(v.clone());
  |                   ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `practice-4` (bin "practice-4") due to previous error

C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-4>
```

Rust - Borrowing

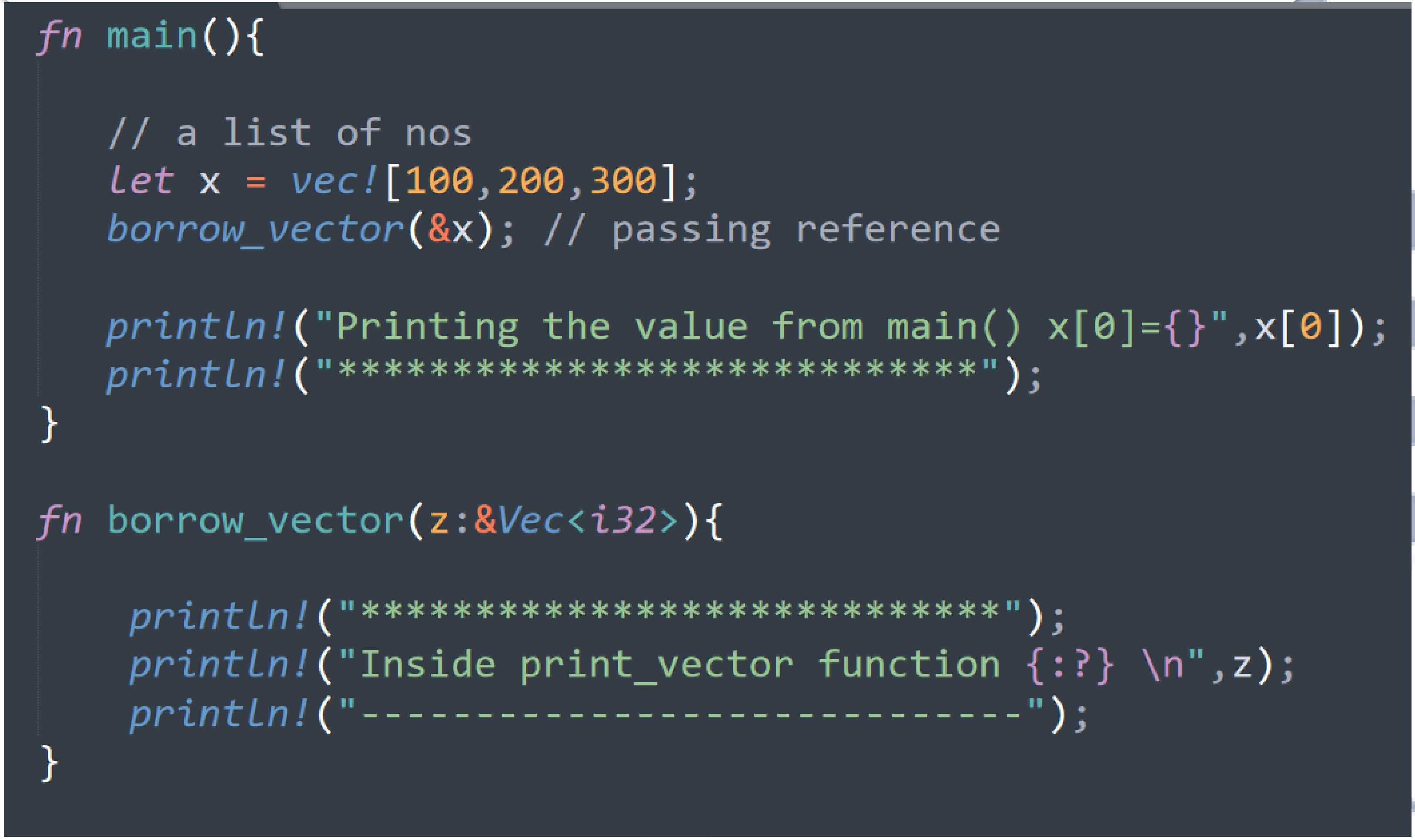
When a function transfers its control over a variable/value to another function temporarily, for a while, it is called **borrowing**.

This is achieved by passing a reference to the variable ***(& var_name)*** rather than passing the variable/value itself to the function.

The ownership of the variable/ value is transferred to the original owner of the variable after the function to which the control was passed completes execution.

Borrowing in Rust

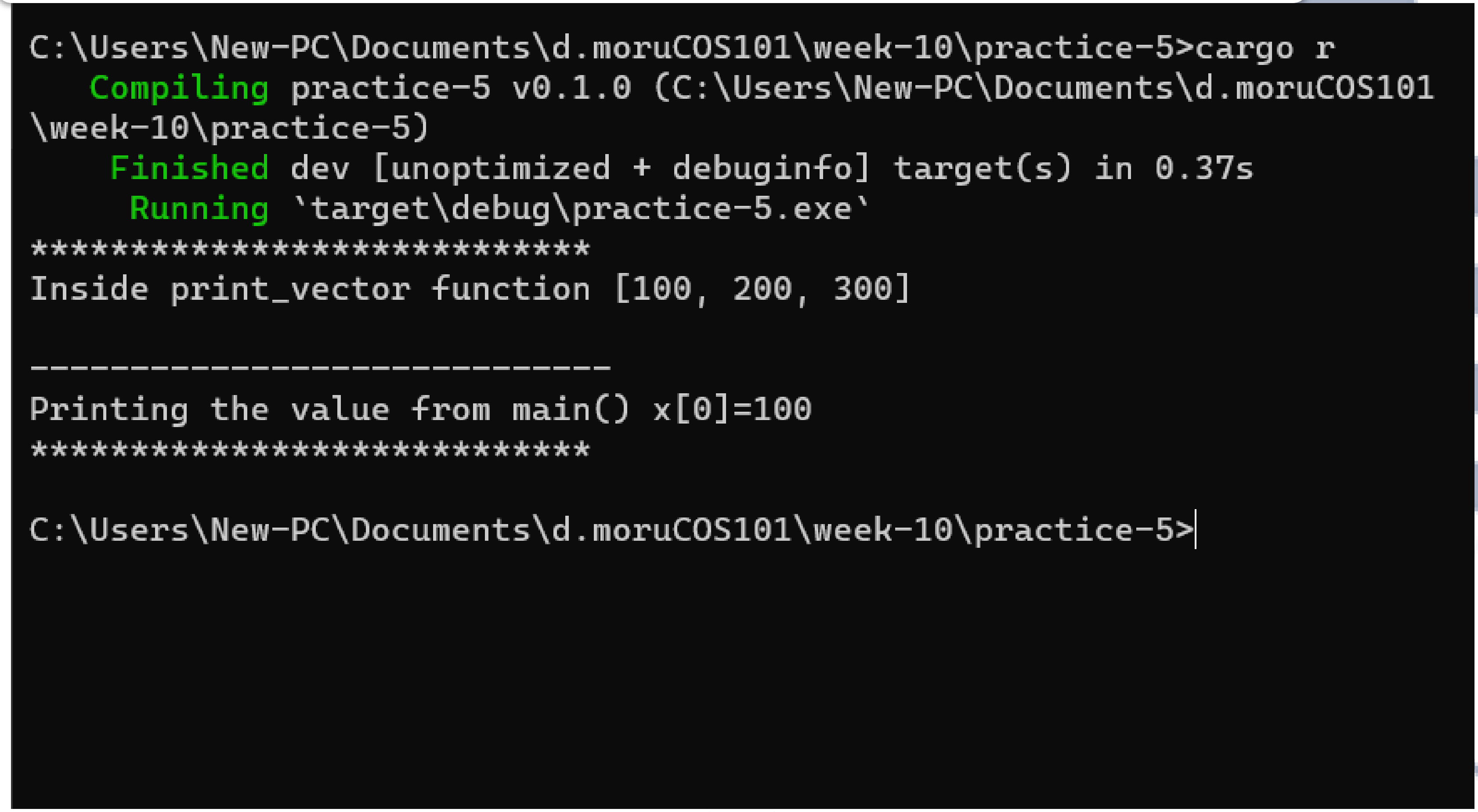
Practice 5: ***week-10/practice_5/src/main.rs***



```rust
fn main(){

   // a list of nos
   let x = vec![100,200,300];
   borrow_vector(&x); // passing reference

   println!("Printing the value from main() x[0]={}",x[0]);
   println!("****************************");
}

fn borrow_vector(z:&Vec<i32>){

    println!("****************************");
    println!("Inside print_vector function {:?} \n",z);
    println!("-----------------------------");
}
```

Rust - Borrowing

Practice 5: week-10/practice_5/src/main.rs (OUTPUT)



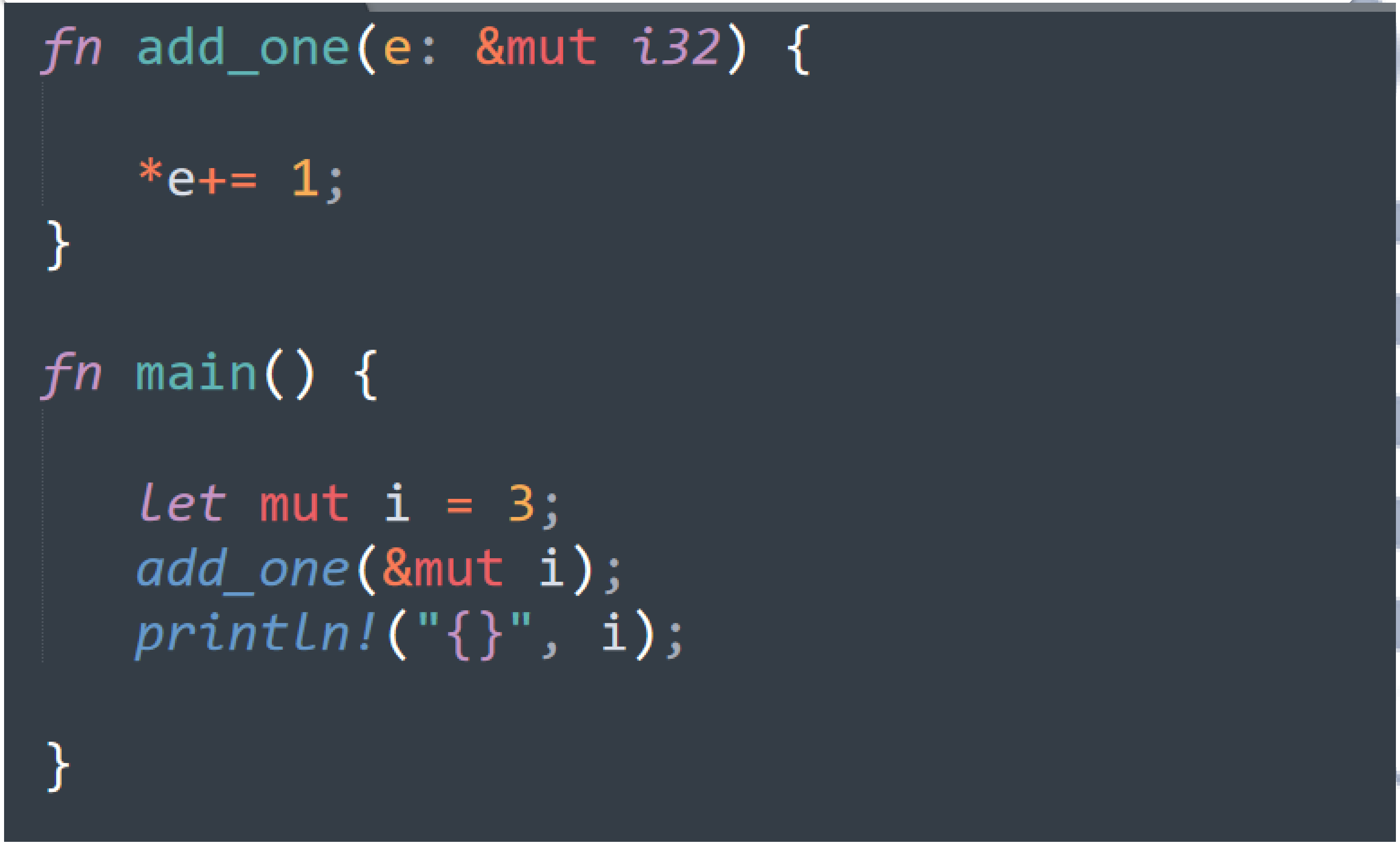
```
C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-5>cargo r
   Compiling practice-5 v0.1.0 (C:\Users\New-PC\Documents\d.moruCOS101
\week-10\practice-5)
    Finished dev [unoptimized + debuginfo] target(s) in 0.37s
     Running `target\debug\practice-5.exe`
****************************
Inside print_vector function [100, 200, 300]

-----------------------------
Printing the value from main() x[0]=100
****************************

C:\Users\New-PC\Documents\d.moruCOS101\week-10\practice-5>
```

Mutable References

- A function can modify a borrowed resource by using a *mutable reference* to such resource.
- A mutable reference is prefixed with ***&mut***.
- Mutable references can operate only on mutable variables.

Mutable References

Practice 6: week-10/practice_6/src/main.rs



```rust
fn add_one(e: &mut i32) {

    *e+= 1;
}

fn main() {

    let mut i = 3;
    add_one(&mut i);
    println!("{}", i);

}
```

Rust - Structure

A structure defines data as a key-value pair.

The *struct* keyword is used to declare a structure.

Since structures are statically typed, every field in the structure must be associated with a data type.

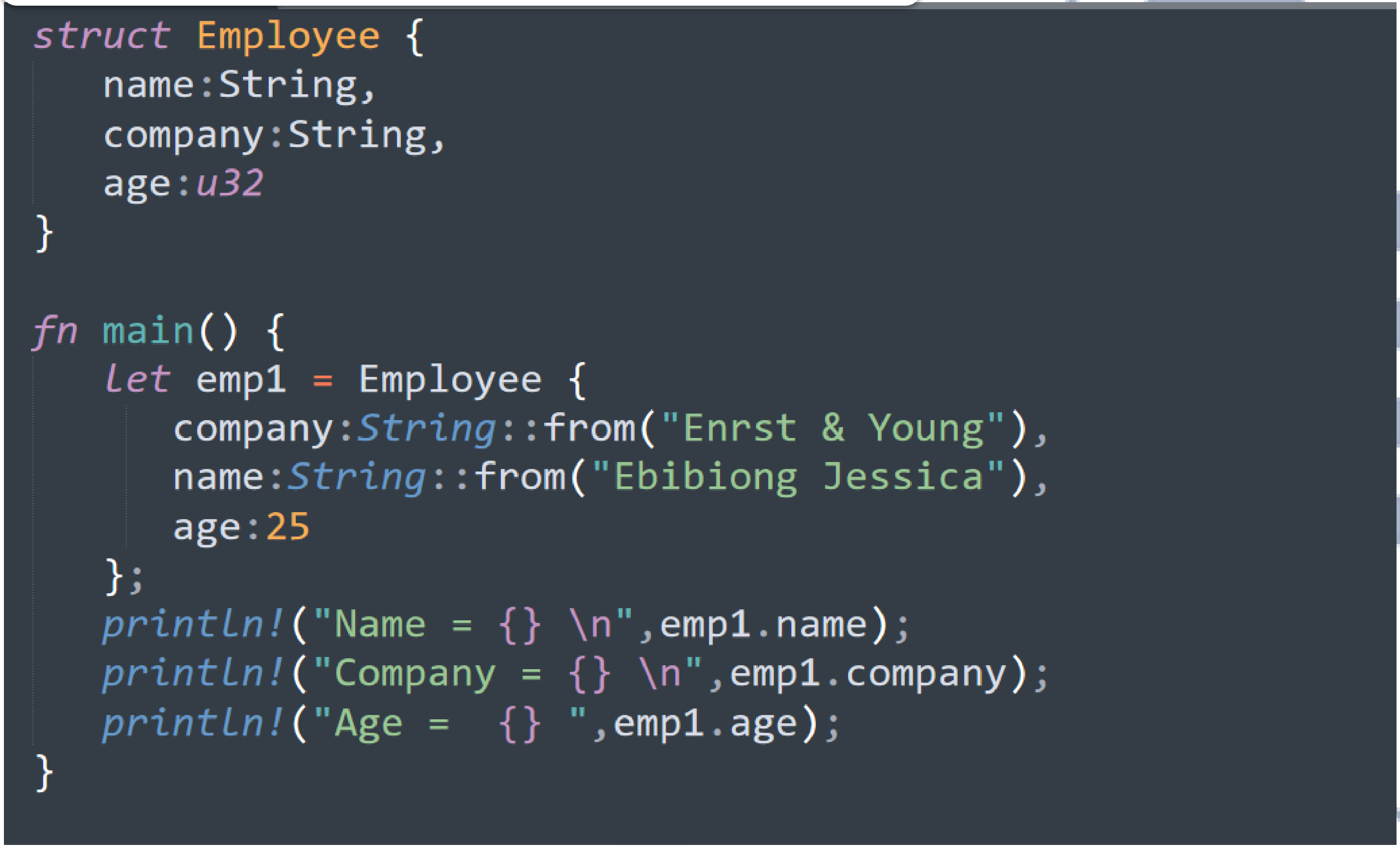
The naming rules and conventions for a structure is like that of a variable.

The structure block must end with a semicolon.

After declaring a struct, each field should be assigned a value. This is known as initialization.

Rust - Structure

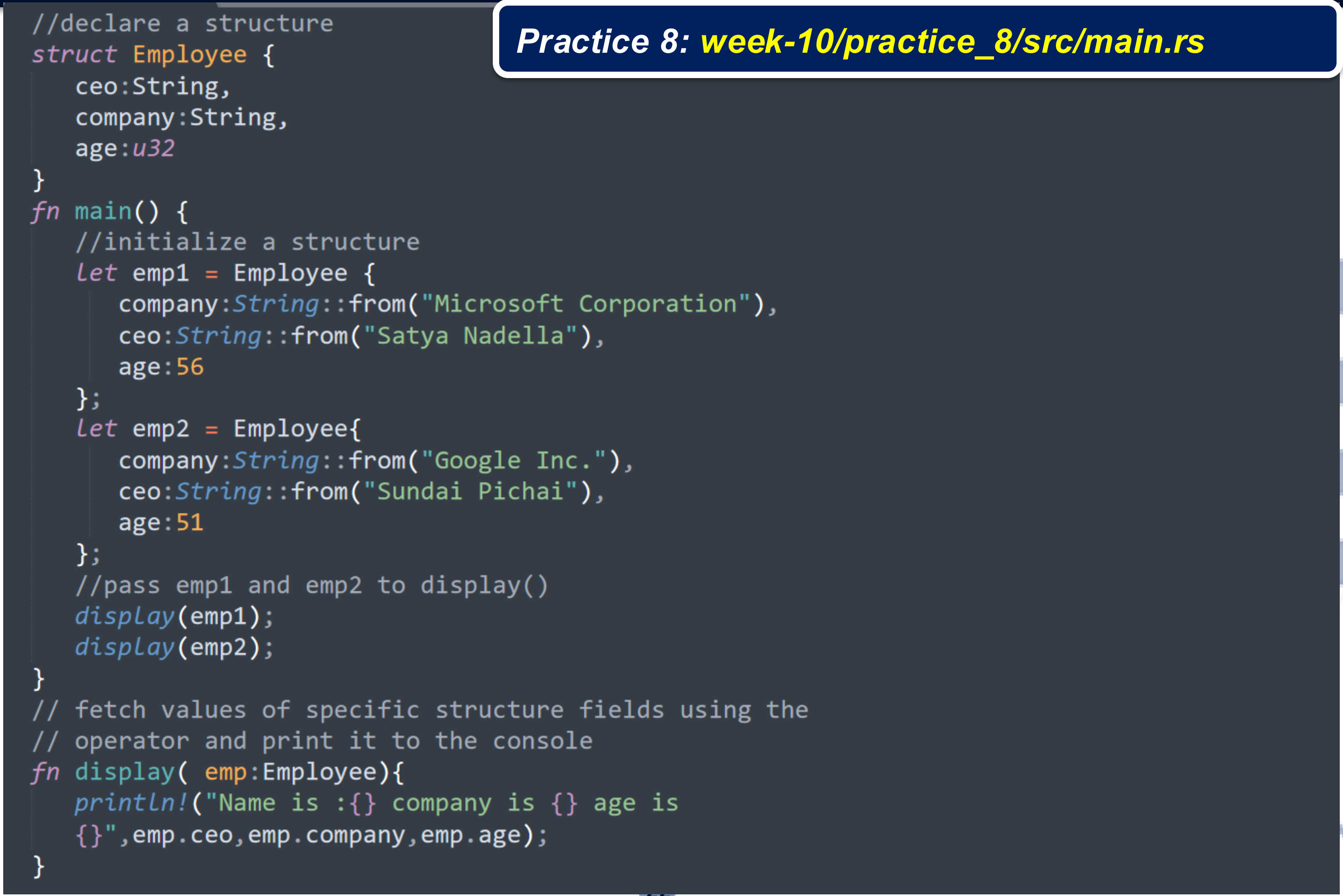
Practice 7: week-10/practice_7/src/main.rs



```rust
struct Employee {
    name:String,
    company:String,
    age:u32
}

fn main() {
    let emp1 = Employee {
        company:String::from("Enrst & Young"),
        name:String::from("Ebibiong Jessica"),
        age:25
    };
    println!("Name = {} \n",emp1.name);
    println!("Company = {} \n",emp1.company);
    println!("Age =  {} ",emp1.age);
}
```

Passing a struct to a function

Practice 8: week-10/practice_8/src/main.rs



```rust
//declare a structure
struct Employee {
   ceo:String,
   company:String,
   age:u32
}
fn main() {
   //initialize a structure
   let emp1 = Employee {
      company:String::from("Microsoft Corporation"),
      ceo:String::from("Satya Nadella"),
      age:56
   };
   let emp2 = Employee{
      company:String::from("Google Inc."),
      ceo:String::from("Sundai Pichai"),
      age:51
   };
   //pass emp1 and emp2 to display()
   display(emp1);
   display(emp2);
}
// fetch values of specific structure fields using the
// operator and print it to the console
fn display( emp:Employee){
   println!("Name is :{} company is {} age is
   {}",emp.ceo,emp.company,emp.age);
}
```

Method in Structure

Methods are like functions.

Methods are declared outside the structure block.

The ***impl*** keyword is used to define a method within the context of a structure.

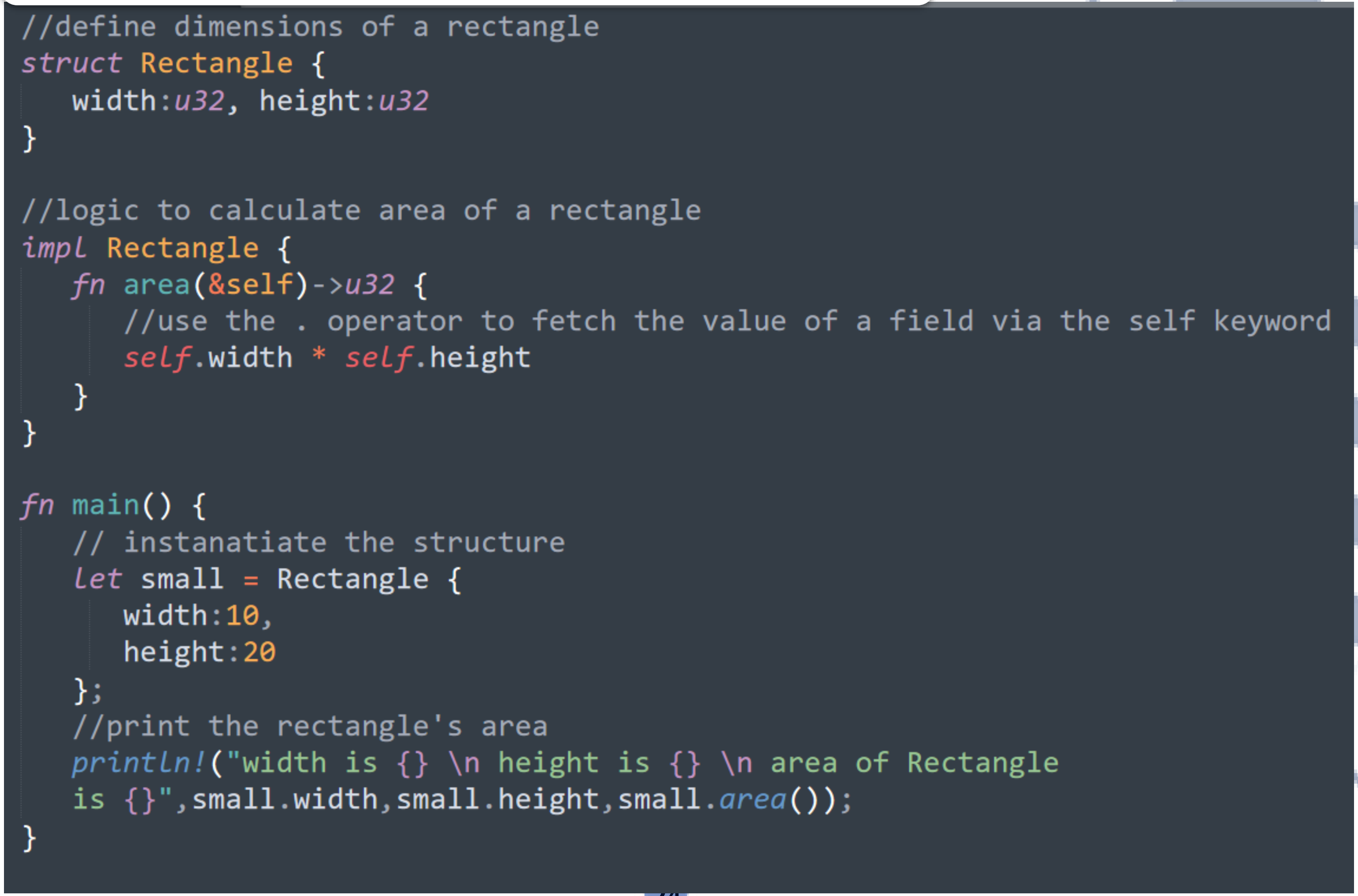
The first parameter of a method will be always ***self***, which represents the calling instance of the structure.

Methods operate on the data members of a structure.

To invoke a method, we need to first instantiate the structure. The method can be called using the structure's instance.

Method in Structure

Practice 9: week-10/practice_9/src/main.rs



```rust
//define dimensions of a rectangle
struct Rectangle {
   width:u32, height:u32
}

//logic to calculate area of a rectangle
impl Rectangle {
   fn area(&self)->u32 {
      //use the . operator to fetch the value of a field via the self keyword
      self.width * self.height
   }
}

fn main() {
   // instanatiate the structure
   let small = Rectangle {
      width:10,
      height:20
   };
   //print the rectangle's area
   println!("width is {} \n height is {} \n area of Rectangle
   is {}",small.width,small.height,small.area());
}
```

Class Project

SCHOOL OF SCIENCE AND TECHNOLOGY

PAN-ATLANTIC UNIVERSITY

Project I

Mr Ogbeifuna runs a series of electronics shops at Alaba International Market in Lagos. He has recently received a consignment of 30 laptop devices at the following cost; 10 HP laptops at 650,000 each, 6 IBM laptops at 755,000 each, 10 Toshiba laptops at 550,000 each, and 4 Dell laptops at 850,000 each. Using your knowledge of Rust struct and method calculate the total cost supposing a customer purchases 3 from each brand.