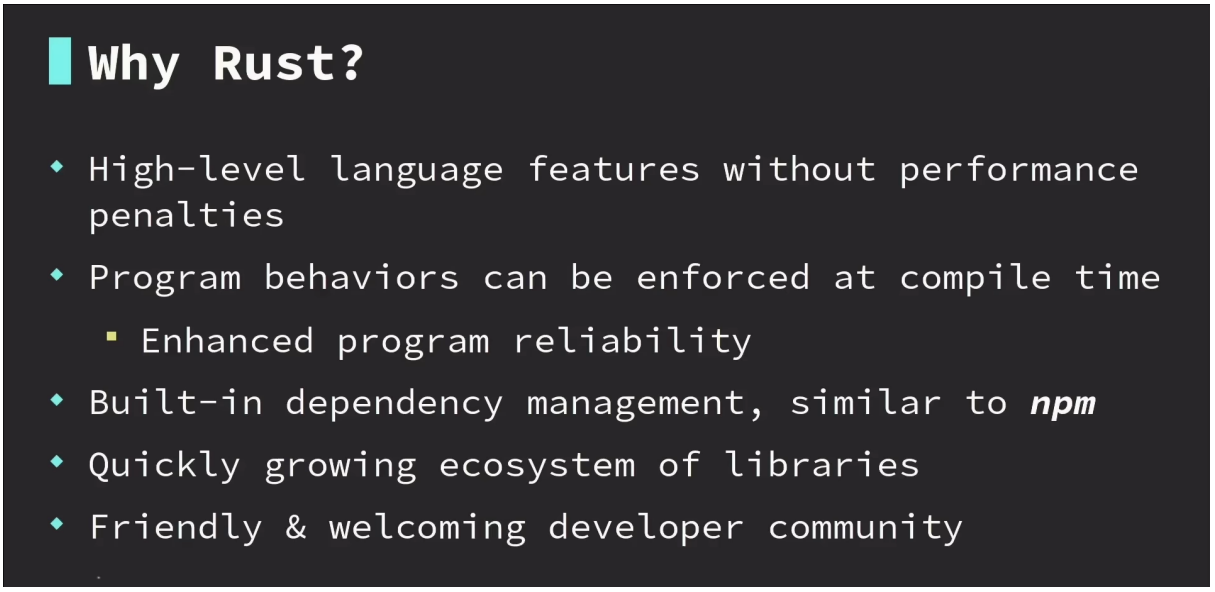
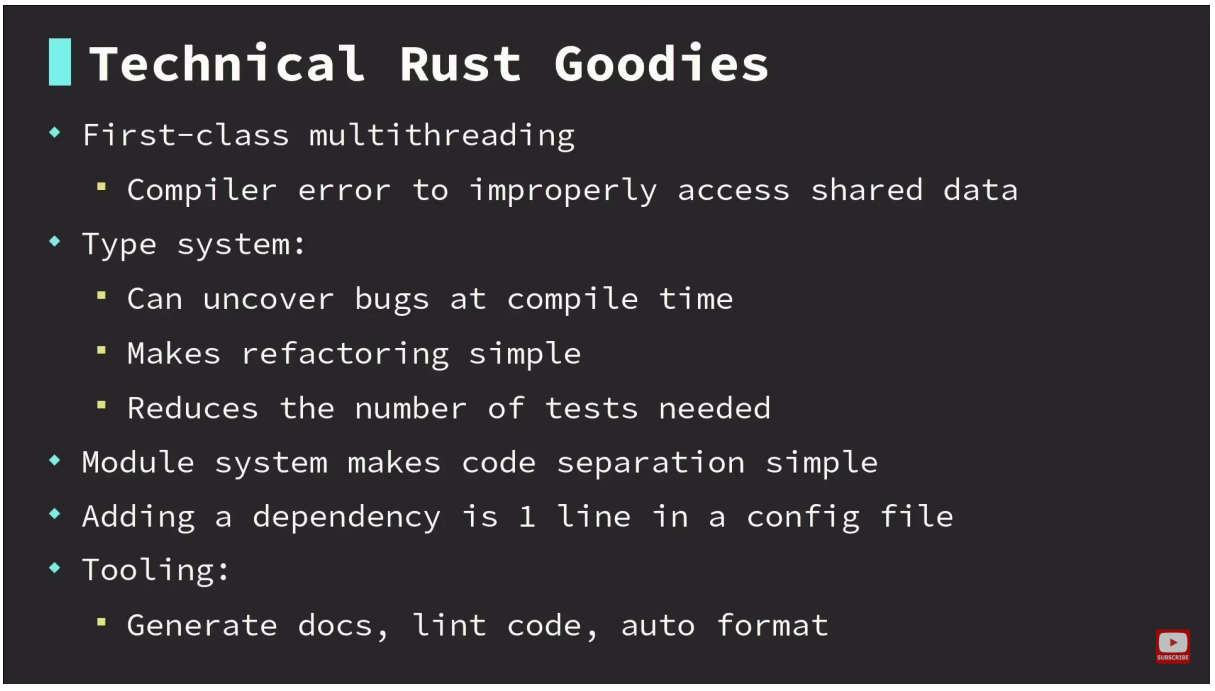
Rust Crash Course Notes





Fundamentals - Data Types

Boolean

Integer

Double / Float

Character

String

Fundamentals - Variables

Immutable by default, but can be made mutable.

```
let two = 2;
let mut name = "Benarjee Sambangi";
```

Fundamentals - Functions

A way to encapsulate program functionality. Utilized for code organization and also makes code easier to read.

```
fn add(a:i32, b:i32) -> i32 {
    a + b
}
```

Fundamentals - Loops

while loop

```
let mut a = 0;
while a !=5 {
    println("{:?}", a);
    a = a + 1;
}
```

Comments

```
//This is a comment
```

cargo run —bin file_name

cargo run -q —bin file_name

The files are in src/bin.

Fundamentals - match

All options must be accounted.

```
fn main() {
    let is_matched = false;

    match is_matched {
        true => println!("Is matched"),
        false => println!("Is not matched"),
    }
}
```

match will be checked by the compiler when an new option is added whereas in if..else it is not notified.

Fundamentals - loop

```
fn main() {
    let mut i = 1;

    loop {
        println!("{:?}", i);
        i += 1;
        if i > 4 {
            break;
        }
    }
}
```

Fundamentals - while loop

```
fn main() {
    let mut counter = 5;

    while counter >= 1 {
        println!("{:?}", counter);
        counter -= 1;
    }
```

```
    println!("done!");
}
```

Working with Data - enum

Enumeration

- Data that can be one of multiple different possibilities
  - Each possibility is called a “variant”
- Provides information about your program to the compiler
  - More robust programs

Example

```
enum Direction {
    Up,
    Down,
    Left,
    Right
}
```

```
fn which_way(go: Direction) {
    match go {
        Direction::Up => "up",
        Direction::Down => "down",
        Direction::Left => "left",
        Direction::Right => "right",
    }
}
```

```
enum Direction {
    Left,
    Right
}

fn main() {
    let go = Direction::Left;
```

```
    match go {
        Direction::Left => println!("go left"),
        Direction::Right => println!("go right"),
    }
}
```

Working with Data - struct

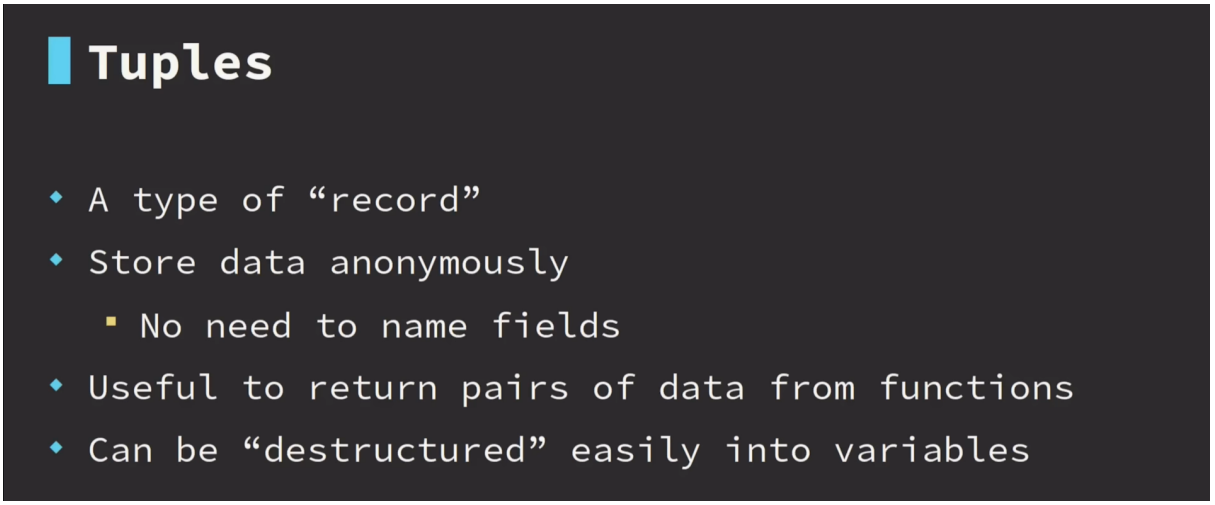
Structure

- A type that contains multiple pieces of data
  - All or nothing – cannot have some pieces of data and not others
- Each piece of data is called a “field”
- Makes working with data easier
  - Similar data can be grouped together

```
enum Flavor{
    Sparkling,
    Sweet,
    Fruity
}
struct Drink {
    flavor: Flavor,
    fluid_oz: f64
}

fn print_drink(drink:Drink){
    match drink.flavor {
        Flavor::Sparkling => println!("Flavor: Sparkling"),
        Flavor::Sweet => println!("Flavor: Sweet"),
        Flavor::Fruity => println!("Flavor: Fruity")
    }
```

```
    println!("oz: {:?}", drink.fluid_oz);
}
fn main() {
    let mydrink = Drink {
        flavor: Flavor::Sparkling,
        fluid_oz: 35.5
    };

    print_drink(mydrink);
}
```

Working with Data - Tuple




```
fn coordinates() -> (i32, i32) {
    (1, 7)
}
fn main() {
    let (x, y) = coordinates();

    if y > 5 {
        println!("greater than 5");
    }else if y < 5 {
        println!("less than 5");
    }else {
```

```
        println!("equal to 5");
    }
}
```

Fundamentals - expressions

Expressions

- Rust is an expression-based language
  - Most things are evaluated and return some value
- Expression values coalesce to a single point
  - Can be used for nesting logic

```
fn print_message(is_gt_100 : bool){
    match is_gt_100 {
        true => println!("its big"),
        false => println!("its small"),
    };
}
fn main() {
    let value = 100;

    let is_gt_100 = value > 100;

    print_message(is_gt_100);
}
```

Fundamentals - Intermediate Memory

Addresses

- All data in memory has an “address”
  - Used to locate data
  - Always the same – only data changes
- Usually don’t utilize addresses directly
  - Variables handle most of the work

Offsets

- Items can be located at an address using an “offset”
- Offsets begin at 0
- Represent the number of bytes away from the original address
  - Normally deal with indexes instead

Fundamentals - Ownership

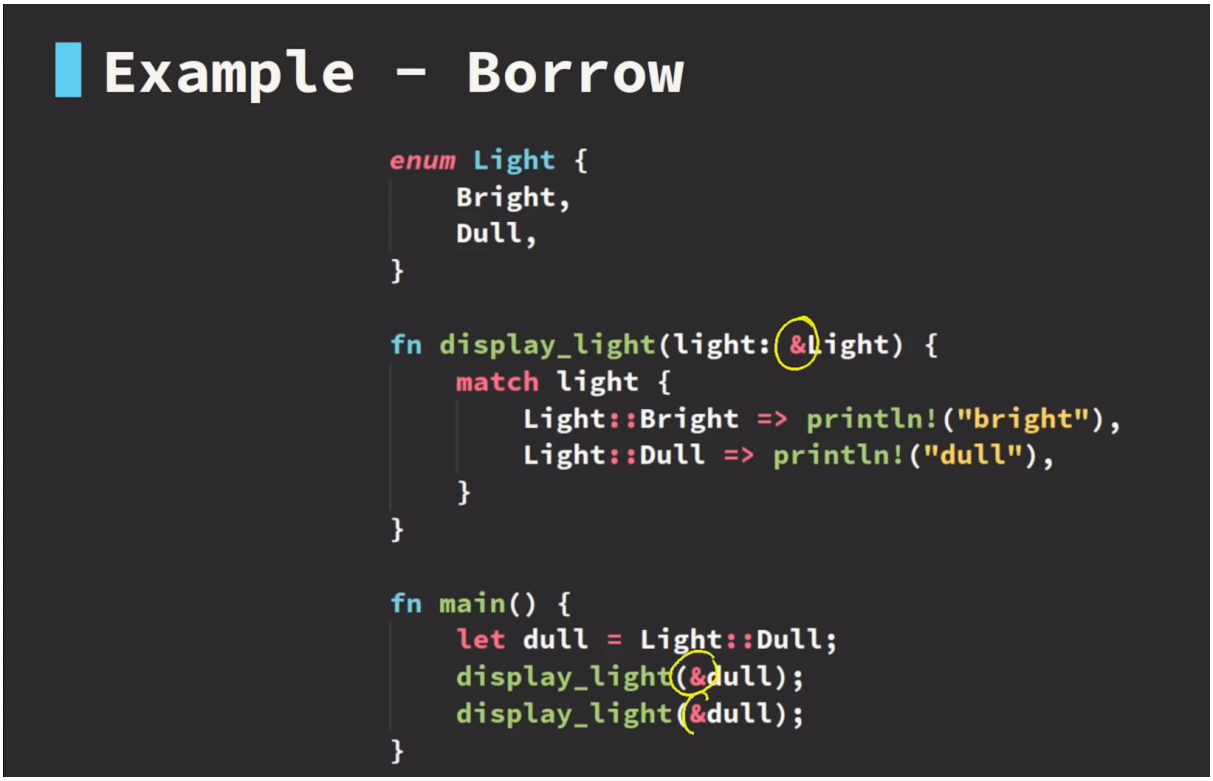
Managing memory

- Programs must track memory
  - If they fail to do so, a “leak” occurs
- Rust utilizes an “ownership” model to manage memory
  - The “owner” of memory is responsible for cleaning up the memory
- Memory can either be “moved” or “borrowed”

Example - Move

```
enum Light {
    Bright,
    Dull,
}

fn display_light(light: Light) {
    match light {
        Light::Bright => println!("bright"),
        Light::Dull => println!("dull"),
    }
}

fn main() {
    let dull = Light::Dull;
    display_light(dull);
    display_light(dull);
}
```

Example - Borrow



```
enum Light {
    Bright,
    Dull,
}

fn display_light(light: &Light) {
    match light {
        Light::Bright => println!("bright"),
        Light::Dull => println!("dull"),
    }
}

fn main() {
    let dull = Light::Dull;
    display_light(&dull);
    display_light(&dull);
}
```

```
struct Grocery {
    quantity: i32,
    id: i32
}

fn print_quantity(items : &Grocery){
    println!("{:?}", items.quantity);
}

fn print_id(items : &Grocery){
    println!("{:?}", items.id);
}
fn main() {
    let items = Grocery {
        quantity: 30,
        id: 1
    };

    print_quantity(&items);
```

```
    print_id(&items);
}
```

impl

impl allows you to implement functionality specific enumeration and structs.

```
enum Color {
    Brown,
    Red
}

impl Color {
    fn print(&self) {
        match self {
            Color::Brown => println!("brown"),
            Color::Red => println!("red"),
        }
    }
}
struct Dimensions {
    width:f64,
    height:f64,
    depth:f64
}

impl Dimensions {
    fn print(&self) {
        println!("width : {:?}", self.width);
        println!("height : {:?}", self.height);
        println!("depth: {:?}", self.depth);
    }
}
struct ShippingBox {
    color: Color,
    weight: f64,
    dimensions: Dimensions,
}
```

```
impl ShippingBox {
    fn new(weight: f64, color: Color, dimensions: Dimensions)
        Self { weight, color, dimensions}
    }

    fn print(&self){
        self.color.print();
        self.dimensions.print();
        println!("weight : {:?}", self.weight);
    }
}
fn main() {
    let small_box_dimensions = Dimensions{
        width: 1.0,
        height: 2.0,
        depth: 3.0,
    };

    let small_box = ShippingBox::new(5.0, Color::Red, small_b

    small_box.print();

}
```

The difference between self and Self is that the **self** means we have already created ShippingBox somewhere in our program, and this **Self** means we are creating a new ShippingBox.

Data Structures - vectors

Vector

- Multiple pieces of data
  - Must be the same type
- Used for lists of information
- Can add, remove, and traverse the entries

```
fn main() {
    let my_numbers = vec![10,20,30,40];

    for element in &my_numbers {
       if *element == 30 {
        println!("thirty")
       } else {
        println!("{:?}", element);
       }
    }

    println!("Length of the vector :: {:?}", my_numbers.len()
}
```

Data Types - String

String and &str

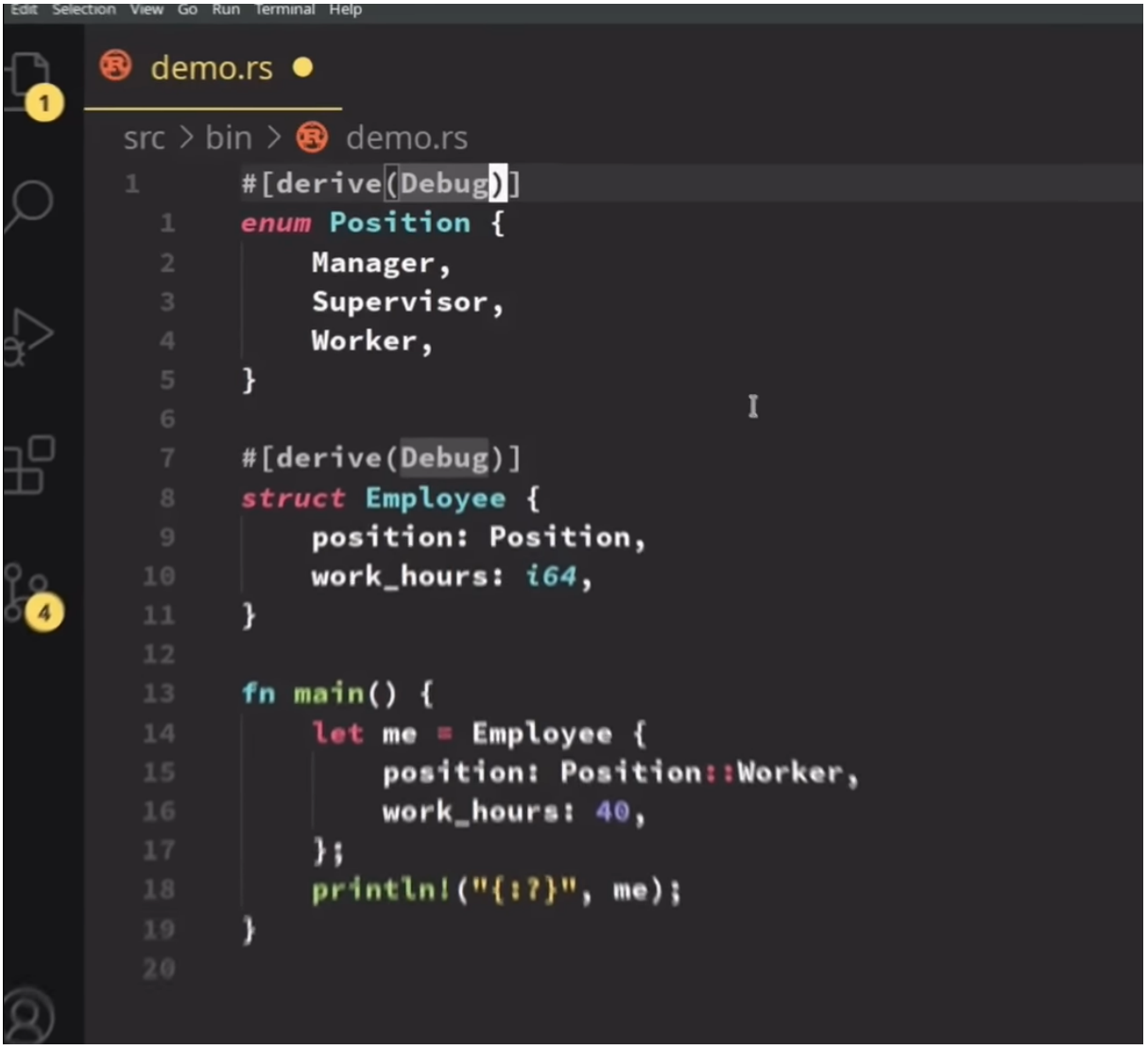
- Two commonly used types of strings
  - *String* - owned
  - *&str* - borrowed *String* slice
- Must use an owned *String* to store in a *struct*
- Use *&str* when passing to a function

Example - Pass to function

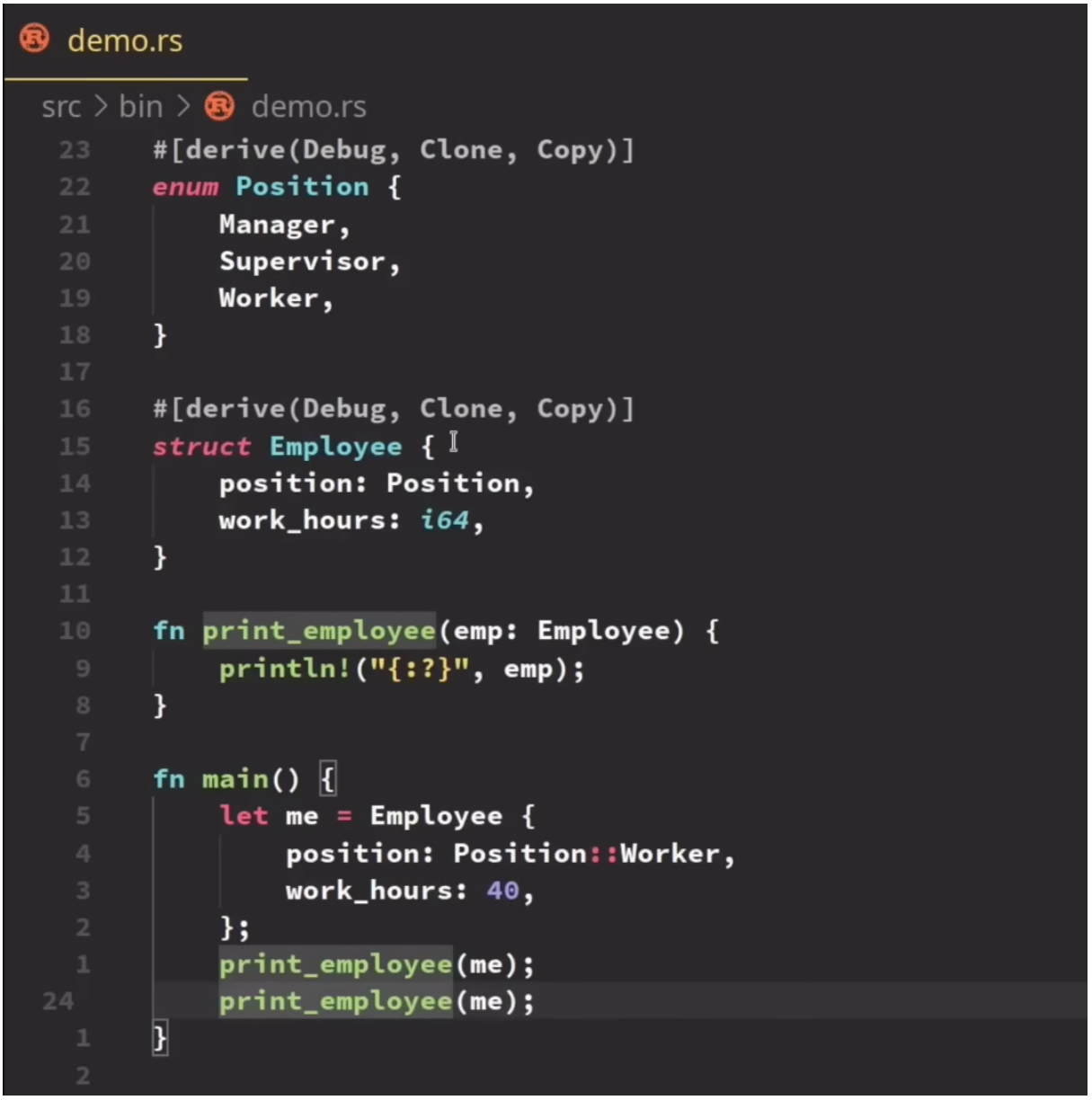
```rust
fn print_it(data: &str) {
    println!("{:?}", data);
}

fn main() {
    print_it("a string slice");
    let owned_string = "owned string".to_owned();
    let another_owned = String::from("another");
    print_it(&owned_string);
    print_it(&another_owned);
}
```

```rust
struct Person {
    age: i32,
    name: String,
    fav_color: String
}

fn print(data: &str){
    println!("{:?}", data);
}
fn main() {
    let people = vec![
        Person {age : 24, name: "Ben".to_owned(), fav_color:
        Person {age : 22, name: "Nava".to_owned(), fav_color:
        Person {age : 22, name: "Lal".to_owned(), fav_color:
    ];

    for person in people {
        if person.age == 24 {
            print(&person.name);
            print(&person.fav_color);
        }
    }
}
```

#[derive(Debug)] macro



It allows to print the structure directly, and all the items should also be derived to avoid compilation errors.



Clone and Copy traits create a copy of the original struct, that is why **me** is able to be passed twice without any error.

Fundamentals - type annotations

Example - Basic

```rust
fn print_many(msg: &str, count: i32) { }

enum Mouse {
    LeftClick,
    RightClick,
    MiddleClick,
}

let num: i32 = 15;
let a: char = 'a';
let left_click: Mouse = Mouse::LeftClick;
```

Example - Generics

```rust
let numbers: Vec<i32> = vec![1, 2, 3];
let letters: Vec<char> = vec!['a', 'b'];
let clicks: Vec<Mouse> = vec![
    Mouse::LeftClick,
    Mouse::LeftClick,
    Mouse::RightClick,
];
```

Generics allow writing flexible, reusable code without sacrificing performance. They enable us to write functions, structs, enums and methods that can operate on many different types while still being type safe.

```
//Before using Generic Function
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
```

```
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&number_list);
    println!("The largest number is {result}");

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&char_list);
    println!("The largest char is {result}");
}
```

```
//After using generic function
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
```

```
        }
    }

    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let result = largest(&number_list);
    println!("The largest number is {result}");

    let char_list = vec!['y', 'm', 'a', 'q'];

    let result = largest(&char_list);
    println!("The largest char is {result}");
}
```

Working with data - enum revisited

Enums

- *enum* is a type that can represent one item at a time
  - Each item is called a variant
- *enum* is not limited to just plain variants
  - Each variant can optionally contain additional data

```
enum Mouse {
    LeftClick,
    RightClick,
    MiddleClick,
```

```
    Scroll(i32),
    Move(i32, i32),
}
```

Advanced match

```
enum Ticket {
    Backstage(String, f64),
    Standard(f64),
    VIP(String, f64)
}
fn main() {
    let tickets = vec![
        Ticket::Backstage("Ben10".to_owned(), 50.0),
        Ticket::Standard(3.0),
        Ticket::VIP("Ben".to_owned() , 100.0)
    ];

    for ticket in tickets {
        match ticket {
            Ticket::Backstage(holder, price) => println!("Bac
            Ticket::Standard(price) => println!("Standard Tic
            Ticket::VIP(holder, price) => println!("VIP Ticke
        }
    }
}
```

Working with data - Option

Option

- A type that may be one of two things
  - Some data of a specified type
  - Nothing
- Used in scenarios where data may not be required or is unavailable
  - Unable to find something
  - Ran out of items in a list
  - Form field not filled out

```
enum Option<T> {
    Some(T),
    None
}
```

```
struct Locker {
    student_name: String,
    locker: Option<i32>
}
fn main() {
    let student_details = Locker {
        student_name: "Ben".to_owned(),
        locker: Some(115),
    };

    println!("Student Name :: {:?}", student_details.student_

    match student_details.locker {
        Some(locker_number) => println!("Locker Number :: {:?
        None => println!("No locker assigned"),
      }
}
```

Documentation

```
cargo doc --open

/// to add documentation
```

Standard Library

```
rustup doc
```

Working with Data - Result

Result

- A data type that contains one of two types of data:
  - “Successful” data
  - “Error” data
- Used in scenarios where an action needs to be taken, but has the possibility of failure
  - Copying a file
  - Connecting to a website

```
enum Result<T, E> {
    Ok(T),
    Err(E)
}
```

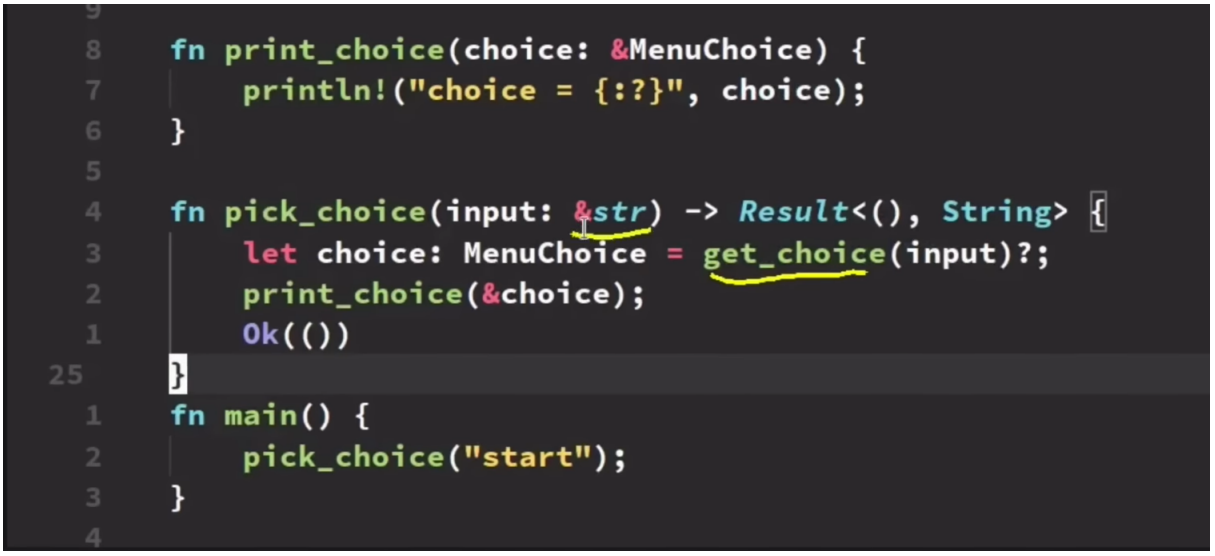
Example

```rust
fn get_sound(name: &str) -> Result<SoundData, String> {
    if name == "alert" {
        Ok(SoundData::new("alert")),
    } else {
        Err("unable to find sound data".to_owned())
    }
}

let sound = get_sound("alert");
match sound {
    Ok(_) => println!("sound data located"),
    Err(e) => println!("error: {:?}", e),
}
```

src > bin > demo.rs

```rust
#[derive(Debug)]
enum MenuChoice {
    MainMenu,
    Start,
    Quit,
}

fn get_choice(input: &str) -> Result<MenuChoice, String> {
    match input {
        "mainmenu" => Ok(MenuChoice::MainMenu),
        "start" => Ok(MenuChoice::Start),
        "quit" => Ok(MenuChoice::Quit),
        _ => Err("menu choice not found".to_owned()),
    }
}
fn main() {
    let choice = get_choice("mainmenu");
    println!("choice = {:?}", choice);
}
```

#[derive(Debug)] allows us to print directly without matching the enum.



```
fn print_choice(choice: &MenuChoice) {
    println!("choice = {:?}", choice);
}

fn pick_choice(input: &str) -> Result<(), String> {
    let choice: MenuChoice = get_choice(input)?;
    print_choice(&choice);
    Ok(())
}
fn main() {
    pick_choice("start");
}
```

? opertor allows us to check the result without using match.

() is unit type.

```
#[derive(Debug)]
struct Adult {
    name: String,
    age: u8
}
impl Adult {
    fn new(name: &str,age: u8) -> Result<Self, &str> {
        if age > 21 {
            Ok(Self{
                age,
                name: name.to_string()
            })
        } else {
            Err("Age must be 21.")
        }
    }
}
fn main() {
    let child = Adult::new("Chotu", 15);
    let adult = Adult::new("Ben", 22);

    match child {
        Ok(child) => println!("{} is {} years old", child.nam
```

```
        Err(e) => println!("{e}"),
    }

    match adult {
        Ok(adult) => println!("{} is {} years old", adult.nam
        Err(e) => println!("{e}"),
    }

}
```

Data Structures - Hashmap

Hashmap

- Collection that stores data as key-value pairs
  - Data is located using the “key”
  - The data is the “value”
- Similar to definitions in a dictionary
- Very fast to retrieve data using the key

```
use std::collections::HashMap;

fn main() {
    let mut stock = HashMap::new();
    stock.insert("Chairs", 5);
    stock.insert("Beds", 3);
    stock.insert("Tables", 2);
    stock.insert("Couches", 0);

    let mut total_items = 0;

    for (item, qty) in stock.iter() {
```

```
        total_items += qty;

        let stock_count = if qty == &0 {
            "out of stock".to_owned()
        } else {
            format!("{:?}", qty)
        };
        println!("Item= {:?}, Stock = {:?}", item, stock_coun
    }

    println!("Total Stock : {:?}", total_items);
}
```