RUSTikales Rust for beginners

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Traits

1. Recap

1. Recap

- Ownership-Model
- References
- Borrow Checker

1. Recap

- Ownership-Model
- References
- Borrow Checker
- Structs
  - Declared using the keyword struct
  - Made out of fields
  - Are type definitions

1. Recap

- Ownership-Model
- References
- Borrow Checker
- Structs
- Associated functions
  - Declared using the keyword impl for a given type
  - Functions are linked to a type
  - Called via struct::fn_name()

1. Recap

- Ownership-Model
- References
- Borrow Checker
- Structs
- Associated functions
- Methods
  - Associated functions where the first parameter is either self, &self or &mut self
  - Called on instances of structs using the dot operator

1. Recap

```rust
struct Line {
    start: Point,
    end: Point
}
0 implementations
struct Point {
    x: i32,
    y: i32,
}
```

1. Recap

```rust
struct Line {
    start: Point,
    end: Point
}
```

Struct declaration

0 implementations

```rust
struct Point {
    x: i32,
    y: i32,
}
```

1. Recap

```rust
struct Line {
    start: Point,
    end: Point
}
```

Lines are made out of two fields

0 implementations

```rust
struct Point {
    x: i32,
    y: i32,
}
```

1. Recap

```
struct Line {
    start: Point,
    end: Point
}
0 implementations
struct Point {
    x: i32,
    y: i32,
}
```

As with most top-level keywords,
structs can be declared in any order

1. Recap

```rust
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

1. Recap

```rust
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

p1 and p2 are instances of type Point

1. Recap

```rust
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
```

Using impl, we can declare functions for given types

```rust
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

1. Recap

```rust
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

Associated function
→ More precise, a method

1. Recap

```rust
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

Takes a reference to an instance of type Line
→ self and Self are special identifiers

1. Recap

```rust
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

Call to an associated function of f32

1. Recap

```rust
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

Type hint suggests that this is a method

→ Methods are special associated functions and can be called as such

1. Recap

```
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

We can use the dot operator to call methods

→ line is automatically borrowed and passed as &self

1. Recap

```
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}   println!("length = {}", Line::length(&line));
```

The dot operator is syntactic sugar
→ Both versions work

1. Recap

```
impl Line {
    fn length(&self) -> f32 {
        let x: f32 = (self.end.x - self.start.x) as f32;
        let y: f32 = (self.end.y - self.start.y) as f32;
        f32::sqrt(self: x * x + y * y)
    }
}
▶ Run | Debug
fn main() {
    let p1: Point = Point { x: 3, y: 4 };
    let p2: Point = Point { x: 5, y: 10 };
    let line: Line = Line { start: p1, end: p2 };
    println!("length = {}", line.length());
}
```

Normal Ownership rules apply
→ p1 and p2 are moved into the line here

2. Traits

2. Traits

– Structs and methods are very powerful, but currently very limited

2. Traits

- Structs and methods are very powerful, but currently very limited
  - We can't print struct instances to the console
  - We can only move structs

2. Traits

```
0 implementations
struct Foo {}
fn no_copy(foo: Foo) {}
▶ Run | Debug
fn main() {
    let foo: Foo = Foo {};
    println!("{}", foo);
    println!("{:?}", foo);
    no_copy(foo);
    println!("{:#?}", foo);
}
```

2. Traits

```
0 implementations
struct Foo {}
fn no_copy(foo: Foo) {}
▶ Run | Debug
fn main() {
    let foo: Foo = Foo {};
    println!("{}", foo);
    println!("{:?}", foo);
    no_copy(foo);
    println!("{:#?}", foo);
}
```

Can‘t print normally

2. Traits

```rust
0 implementations
struct Foo {}
fn no_copy(foo: Foo) {}
▶ Run | Debug
fn main() {
    let foo: Foo = Foo {};
    println!("{}", foo);
    println!("{:?}", foo);
    no_copy(foo);
    println!("{:#?}", foo);
}
```

Can‘t debug print

2. Traits

```
0 implementations
struct Foo {}
fn no_copy(foo: Foo) {}
▶ Run | Debug
fn main() {
    let foo: Foo = Foo {};
    println!("{}", foo);
    println!("{:?}", foo);
    no_copy(foo);
    println!("{:#?}", foo);
}
```

foo is moved, we can‘t use it below

2. Traits

- Structs and methods are very powerful, but currently very limited
  - We can't print struct instances to the console
  - We can only move structs
- For this and more, Rust has the trait system

2. Traits

- Structs and methods are very powerful, but currently very limited
  - We can't print struct instances to the console
  - We can only move structs
- For this and more, Rust has the trait system
- Traits are contracts
  - A trait is made out of function declarations
  - If we want to use a trait for a struct, we need to implement those functions

2. Traits

- Structs and methods are very powerful, but currently very limited
  - We can’t print struct instances to the console
  - We can only move structs
- For this and more, Rust has the trait system
- Traits are contracts
- Traits can be defined using the keyword trait
- Traits can be implemented for a type using the keyword impl

```rust
trait Geometry {
    fn print_area(&self) {
        println!("area={}", self.area());
    }
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

```rust
trait Geometry {
    fn print_area(&self) {
        println!("area={}", self.area());
    }
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
```

Trait definition

1 implementation

```rust
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

```rust
trait Geometry {
    fn print_area(&self) {
        println!("area={}", self.area());
    }
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
```

Function declaration

```rust
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

```rust
trait Geometry {
    fn print_area(&self) {
        println!("area={}", self.area());
    }
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

Function definition

```rust
trait Geometry {
    fn print_area(&self) {
        println!("area={}", self.area());
    }
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

Traits are implemented for given types

```rust
trait Geometry {
    fn print_area(&self) {
        println!("area={}", self.area());
    }
    fn area(&self) -> f64;
    fn perimeter(&self) -> f64;
}
1 implementation
struct Rectangle {
    width: f64,
    height: f64,
}
impl Geometry for Rectangle {
    fn area(&self) -> f64 {
        self.width * self.height
    }
    fn perimeter(&self) -> f64 {
        2.0 * (self.width + self.height)
    }
}
```

Implement each function

```rust
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    println!("area = {}", rect.area());
    println!("perimeter = {}", rect.perimeter());
```

After we implemented a trait, we can call the functions like any other associated function or method

```rust
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    println!("area = {}", rect.area());
    println!("perimeter = {}", rect.perimeter());
```

After we implemented a trait, we can call the functions like any other associated function or method

```rust
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    println!("area = {}", rect.area());
    println!("perimeter = {}", rect.perimeter());
```

3/3

This seems redundant, why can‘t we simply use impl? What do we gain?

After we implemented a trait, we can call the functions like any other associated function or method

```rust
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    println!("area = {}", rect.area());
    println!("perimeter = {}", rect.perimeter());
```

3/3

This seems redundant, why can‘t we simply use impl? What do we gain?
→ We can generalize our code to take in *any* struct that implements a given trait

```rust
fn calculate_geometry(obj: &impl Geometry) {
    println!("Area: {}", obj.area());
    println!("Perimeter: {}", obj.perimeter());
}
▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    println!("area = {}", rect.area());
    println!("perimeter = {}", rect.perimeter());
    calculate_geometry(obj: &rect);
    let vector: Vec<i32> = vec![1, 2, 3];
    calculate_geometry(obj: &vector);
}
```

We accept every struct that implements the Geometry trait

```rust
fn calculate_geometry(obj: &impl Geometry) {
    println!("Area: {}", obj.area());
    println!("Perimeter: {}", obj.perimeter());
}

▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    println!("area = {}", rect.area());
    println!("perimeter = {}", rect.perimeter());
    calculate_geometry(obj: &rect);
    let vector: Vec<i32> = vec![1, 2, 3];
    calculate_geometry(obj: &vector);
}
```

```rust
fn calculate_geometry(obj: &impl Geometry) {
    println!("Area: {}", obj.area());
    println!("Perimeter: {}", obj.perimeter());
}
▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    println!("area = {}", rect.area());
    println!("perimeter = {}", rect.perimeter());
    calculate_geometry(obj: &rect);
    let vector: Vec<i32> = vec![1, 2, 3];
    calculate_geometry(obj: &vector);
}
```

Traits are contracts

→ We don‘t know the direct type of obj,

but we do know that it has those methods

```rust
fn calculate_geometry(obj: &impl Geometry) {
    println!("Area: {}", obj.area());
    println!("Perimeter: {}", obj.perimeter());
}
▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    println!("area = {}", rect.area());
    println!("perimeter = {}", rect.perimeter());
    calculate_geometry(obj: &rect);
    let vector: Vec<i32> = vec![1, 2, 3];
    calculate_geometry(obj: &vector);
}
```

Rectangle implements Geometry, so we can pass instances of type Rectangle

```rust
fn calculate_geometry(obj: &impl Geometry) {
    println!("Area: {}", obj.area());
    println!("Perimeter: {}", obj.perimeter());
}
▶ Run | Debug
fn main() {
    let rect: Rectangle = Rectangle { width: 10.0, height: 20.0 };
    println!("area = {}", rect.area());
    println!("perimeter = {}", rect.perimeter());
    calculate_geometry(obj: &rect);
    let vector: Vec<i32> = vec![1, 2, 3];
    calculate_geometry(obj: &vector);
}
```

Rectangle implements Geometry, so we can pass instances of type Rectangle

Vec does not implement Geometry, so we can‘t pass instances of type Vec

The compiler error for println!(“{}“)

```
error[E0277]: `Foo` doesn't implement `std::fmt::Display`
  --> src\main.rs:11:20
   |
11 |     println!("{}", foo);
   |                    ^^^ `Foo` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Foo`
```

The compiler error for println!(“{:?}“)

```
error[E0277]: `Foo` doesn't implement `Debug`
  --> src\main.rs:12:22
   |
12 |     println!("{:?}", foo);
   |                      ^^^ `Foo` cannot be formatted using `{:?}`
   |
   = help: the trait `Debug` is not implemented for `Foo`
   = note: add `#[derive(Debug)]` to `Foo` or manually `impl Debug for Foo`
```

The compiler error for println!("{}")

```
error[E0277]: `Foo` doesn't implement `std::fmt::Display`
  --> src\main.rs:11:20
   |
11 |     println!("{}", foo);
   |                    ^^^ `Foo` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Foo`
```

The compiler error for println!("{:?}")

```
error[E0277]: `Foo` doesn't implement `Debug`
  --> src\main.rs:12:22
   |
12 |     println!("{:?}", foo);
   |                      ^^^ `Foo` cannot be formatted using `{:?}`
   |
   = help: the trait `Debug` is not implemented for `Foo`
   = note: add `#[derive(Debug)]` to `Foo` or manually `impl Debug for Foo`
```

Aha!

The compiler error for println!(“{}“)

```
error[E0277]: `Foo` doesn't implement `std::fmt::Display`
  --> src\main.rs:11:20
   |
11 |     println!("{}", foo);
   |                    ^^^ `Foo` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Foo`
```

```rust
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
```

The compiler error for println!("{}")

```
error[E0277]: `Foo` doesn't implement `std::fmt::Display`
  --> src\main.rs:11:20
   |
11 |     println!("{}", foo);
   |                    ^^^ `Foo` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Foo`
```

```rust
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
```

All formatter traits are located in the module std::fmt

→ They control the behavior of println!() and other macros

The compiler error for println!(“{}“)

```
error[E0277]: `Foo` doesn't implement `std::fmt::Display`
  --> src\main.rs:11:20
   |
11 |     println!("{}", foo);
   |                    ^^^ `Foo` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Foo`
```

```rust
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
```

They only contain one function that we need to implement

The compiler error for println!(“{}“)

```
error[E0277]: `Foo` doesn't implement `std::fmt::Display`
  --> src\main.rs:11:20
   |
11 |     println!("{}", foo);
   |                    ^^^ `Foo` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Foo`
```

```rust
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
```

The trait requires that parameter and return type, but we‘ll just leave that to the write!() call

The compiler error for println!(“{}“)

```
error[E0277]: `Foo` doesn't implement `std::fmt::Display`
  --> src\main.rs:11:20
   |
11 |     println!("{}", foo);
   |                    ^^^ `Foo` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Foo`
```

```rust
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
```

Using write!(), we can now implement Display!

write!() is identical to println!() except that it requires a target instead of writing to the console

The compiler error for println!("{}")

```
error[E0277]: `Foo` doesn't implement `std::fmt::Display`
  --> src\main.rs:11:20
   |
11 |     println!("{}", foo);
   |                    ^^^ `Foo` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Foo`
```

```rust
impl std::fmt::Display for Foo {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        write!(f, "We have implemented the Display trait for Foo!")
    }
}
```

```
     Running `target\debug\traits.exe`
We have implemented the Display trait for Foo!
```

Intermission - Strings

Intermission - Strings

- write!, format!, print! – A lot of macros make use of Strings

Intermission - Strings

- write!, format!, print! – A lot of macros make use of Strings
- Rust has two commonly used String types
  - &str → String slices
  - String → String structs

Intermission - Strings

- write!, format!, print! – A lot of macros make use of Strings
- Rust has two commonly used String types
  - &str → String slices
  - String → String structs
- Slices are special references → They don‘t own the underlying data → &str is not resizable
- Strings are Vectors → They own the underlying data → String is resizable

Intermission - Strings

- write!, format!, print! – A lot of macros make use of Strings
- Rust has two commonly used String types
  - &str → String slices
  - String → String structs
- Slices are special references → They don't own the underlying data → &str is not resizable
- Strings are Vectors → They own the underlying data → String is resizable

```
#[derive(PartialEq, PartialOrd, Eq, Ord)]
#[stable(feature = "rust1", since = "1.0.0")]
#[cfg_attr(not(test), lang = "String")]
63 implementations
pub struct String {
    vec: Vec<u8>,
}
```

String is literally a Vector :^)

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let hello: &str = &string[0..5];
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("hello: {}", hello);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let hello: &str = &string[0..5];
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("hello: {}", hello);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

String literals are located in the data section of the executable

→ Third memory region besides Stack and Heap

→ Not resizable

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let hello: &str = &string[0..5];
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("hello: {}", hello);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

String slices are made of two fields
→ A pointer to the start of some memory
→ Length in bytes

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let hello: &str = &string[0..5];
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("hello: {}", hello);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

String slices always borrow underlying data

→ Data then can be in the data section

→ Data can also be on the heap

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let hello: &str = &string[0..5];
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("hello: {}", hello);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

We can create a String instance using String::from(&str)

→ Creates a copy of the Slice on the Heap

→ Resizable

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let hello: &str = &string[0..5];
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println!("str: {}", str);
    println!("string: {}", string);
    println!("hello: {}", hello);
    println!("emote_char: {}", emote_char);
    println!("emote: {}", emote);
    println!("emote_string: {}", emote_string);
    println!("hello_in_greek: {}", hello_in_greek);
    println!("hello_in_japanese: {}", hello_in_japanese);
}
```

Strings in Rust are always valid UTF-8

→ We can use emojis, arabic, cyrillic, ... characters

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let hello: &str = &string[0..5];
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println
    println
    println
    println
    println
    println
    println
    println                                                    e);
}
```

```
     Running `target\debug\strings.exe`
str: Hello, world!
string: Hello, world!
hello: Hello
emote_char: 👋
emote: 👋
emote_string: 👋
hello_in_greek: Γεια σου κόσμε!
hello_in_japanese: こんにちは世界!
```

```rust
fn main() {
    let str: &str = "Hello, world!";
    let string: String = String::from("Hello, world!");
    let hello: &str = &string[0..5];
    let emote_char: char = '👋';
    let emote: &str = "👋";
    let emote_string: String = String::from("👋");
    let hello_in_greek: String = String::from("Γεια σου κόσμε!");
    let hello_in_japanese: String = String::from("こんにちは世界!");
    println
    println
    println
    println
    println
    println
    println
    println                                              e);
}
```

```
     Running `target\debug\strings.exe`
str: Hello, world!
string: Hello, world!
hello: Hello
emote_char: 👋
emote: 👋
emote_string: 👋
hello_in_greek: Γεια σου κόσμε!
hello_in_japanese: こんにちは世界!
```

Slice of the first 5 bytes of the String

Intermission - Strings

```rust
let emote: &str = "👋";
let emote_slice: &str = &emote[0..1];
```

```
thread 'main' panicked at src\main.rs:7:29:
byte index 1 is not a char boundary; it is inside '👋' (bytes 0..4) of `👋`
```

UTF-8 characters can be multiple bytes long
→ Indices are byte boundaries, not glyph boundaries
→ Rust guarantees valid UTF-8 Strings → Panic at runtime

Intermission - format!()

Intermission - format!()

- The format!() macro family is an important tool to turn any value or variable into a String

Intermission - format!()

- The format!() macro family is an important tool to turn any value or variable into a String
- The first argument is always the format string
  - String literal in which we can provide placeholders for the values we want to format

Intermission - format!()

- The format!() macro family is an important tool to turn any value or variable into a String
- The first argument is always the format string
  - String literal in which we can provide placeholders for the values we want to format
- For every placeholder you specify, you need to provide an additional argument
  - In contrast to functions, macros do allow variable amount of arguments

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Call to format macro

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Format String

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Placeholder

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

One placeholder
→ We need to provide one argument

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

format!() returns a String which we can then use

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

println!() is identical to format!(), except that it prints the String instead of returning it

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

write!() is also part of the family, except that it needs a target to write to

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Format String

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Three placeholders
→ We need to provide three arguments

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

We can list different kinds of placeholders

→ {} requires std::fmt::Display

→ {:?} requires std::fmt::Debug

→ {:#?} is Debug, but nicely printed

... and many more

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person); Format Strings do allow named arguments
}
```

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

```
Name: John, Age: 32, Height: 180
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
Name: John, Age: 32, Height: 180
```

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person); Calls our Display-implementation
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

```
Name: John, Age: 32, Height: 180
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
Name: John, Age: 32, Height: 180
```

```rust
#[derive(Debug)]
```

Using macros, we can automatically implement some traits

```rust
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

```
Name: John, Age: 32, Height: 180
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
Name: John, Age: 32, Height: 180
```

```rust
#[derive(Debug)]
2 implementations
struct Person { name: String, age: u16, height: u16 }
impl std::fmt::Display for Person {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result {
        write!(f, "Name: {}, Age: {}, Height: {}", self.name, self.age, self.height)
    }
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: String::from("John"),
        age: 32,
        height: 180,
    };
    let as_str: String = format!("{}", person);
    println!("{}", as_str);
    println!("{:?}", person);
    println!("{:#?}", person);
    println!("{p}", p = person);
}
```

Debug print but with linebreaks

```
Name: John, Age: 32, Height: 180
Person { name: "John", age: 32, height: 180 }
Person {
    name: "John",
    age: 32,
    height: 180,
}
Name: John, Age: 32, Height: 180
```

2. Traits

– The trait system controls nearly everything in Rust

2. Traits

– The trait system controls nearly everything in Rust
→ Arithmetic operations → std::ops::Add, std::ops::Sub etc.
→ Comparisons → std::cmp::PartialEq, std::cmp::Eq etc.
→ for-loops → std::iter::Iterator etc.
→ Move semantics → std::marker::Copy

```rust
#[derive(Debug, PartialEq, Clone, Copy)]
6 implementations
struct Vector2D {
    x: f64,
    y: f64
}
impl std::ops::Mul<f64> for Vector2D {
    type Output = Self;
    fn mul(self, rhs: f64) -> Self::Output {
        Self { x: self.x * rhs, y: self.y * rhs }
    }
}
impl std::ops::Add for Vector2D {
    type Output = Self;
    fn add(self, rhs: Self) -> Self::Output {
        Self { x: self.x + rhs.x, y: self.y + rhs.y }
    }
}
▶ Run | Debug
fn main() {
    let v: Vector2D = Vector2D { x: 1.0, y: 2.0 };
    assert!(v * 10.0 == Vector2D { x: 10.0, y: 20.0 });
    assert!(v + v == Vector2D { x: 2.0, y: 4.0 });
}
```

```rust
#[derive(Debug, PartialEq, Clone, Copy)]
6 implementations
struct Vector2D {
    x: f64,
    y: f64
}
impl std::ops::Mul<f64> for Vector2D {
    type Output = Self;
    fn mul(self, rhs: f64) -> Self::Output {
        Self { x: self.x * rhs, y: self.y * rhs }
    }
}
impl std::ops::Add for Vector2D {
    type Output = Self;
    fn add(self, rhs: Self) -> Self::Output {
        Self { x: self.x + rhs.x, y: self.y + rhs.y }
    }
}
▶ Run | Debug
fn main() {
    let v: Vector2D = Vector2D { x: 1.0, y: 2.0 };
    assert!(v * 10.0 == Vector2D { x: 10.0, y: 20.0 });
    assert!(v + v == Vector2D { x: 2.0, y: 4.0 });
}
```

A struct representing a mathematical vector

```rust
#[derive(Debug, PartialEq, Clone, Copy)]
6 implementations
struct Vector2D {
    x: f64,
    y: f64
}
impl std::ops::Mul<f64> for Vector2D {
    type Output = Self;
    fn mul(self, rhs: f64) -> Self::Output {
        Self { x: self.x * rhs, y: self.y * rhs }
    }
}
impl std::ops::Add for Vector2D {
    type Output = Self;
    fn add(self, rhs: Self) -> Self::Output {
        Self { x: self.x + rhs.x, y: self.y + rhs.y }
    }
}
▶ Run | Debug
fn main() {
    let v: Vector2D = Vector2D { x: 1.0, y: 2.0 };
    assert!(v * 10.0 == Vector2D { x: 10.0, y: 20.0 });
    assert!(v + v == Vector2D { x: 2.0, y: 4.0 });
}
```

This implements Vector2D * f64

```rust
#[derive(Debug, PartialEq, Clone, Copy)]
6 implementations
struct Vector2D {
    x: f64,
    y: f64
}
impl std::ops::Mul<f64> for Vector2D {
    type Output = Self;
    fn mul(self, rhs: f64) -> Self::Output {
        Self { x: self.x * rhs, y: self.y * rhs }
    }
}
impl std::ops::Add for Vector2D {
    type Output = Self;
    fn add(self, rhs: Self) -> Self::Output {
        Self { x: self.x + rhs.x, y: self.y + rhs.y }
    }
}
▶ Run | Debug
fn main() {
    let v: Vector2D = Vector2D { x: 1.0, y: 2.0 };
    assert!(v * 10.0 == Vector2D { x: 10.0, y: 20.0 });
    assert!(v + v == Vector2D { x: 2.0, y: 4.0 });
}
```

This implements Vector2D + Vector2D

```rust
#[derive(Debug, PartialEq, Clone, Copy)]
6 implementations
struct Vector2D {
    x: f64,
    y: f64
}
impl std::ops::Mul<f64> for Vector2D {
    type Output = Self;
    fn mul(self, rhs: f64) -> Self::Output {
        Self { x: self.x * rhs, y: self.y * rhs }
    }
}
impl std::ops::Add for Vector2D {
    type Output = Self;
    fn add(self, rhs: Self) -> Self::Output {
        Self { x: self.x + rhs.x, y: self.y + rhs.y }
    }
}
▶ Run | Debug
fn main() {
    let v: Vector2D = Vector2D { x: 1.0, y: 2.0 };
    assert!(v * 10.0 == Vector2D { x: 10.0, y: 20.0 });
    assert!(v + v == Vector2D { x: 2.0, y: 4.0 });
}
```

This implements Vector2D == Vector2D

```rust
#[derive(Debug, PartialEq, Clone, Copy)]
6 implementations
struct Vector2D {
    x: f64,
    y: f64
}
impl std::ops::Mul<f64> for Vector2D {
    type Output = Self;
    fn mul(self, rhs: f64) -> Self::Output {
        Self { x: self.x * rhs, y: self.y * rhs }
    }
}
impl std::ops::Add for Vector2D {
    type Output = Self;
    fn add(self, rhs: Self) -> Self::Output {
        Self { x: self.x + rhs.x, y: self.y + rhs.y }
    }
}
▶ Run | Debug
fn main() {
    let v: Vector2D = Vector2D { x: 1.0, y: 2.0 };
    assert!(v * 10.0 == Vector2D { x: 10.0, y: 20.0 });
    assert!(v + v == Vector2D { x: 2.0, y: 4.0 });
}
```

This moves v, twice!

```rust
#[derive(Debug, PartialEq, Clone, Copy)]
6 implementations
struct Vector2D {
    x: f64,
    y: f64
}
impl std::ops::Mul<f64> for Vector2D {
    type Output = Self;
    fn mul(self, rhs: f64) -> Self::Output {
        Self { x: self.x * rhs, y: self.y * rhs }
    }
}
impl std::ops::Add for Vector2D {
    type Output = Self;
    fn add(self, rhs: Self) -> Self::Output {
        Self { x: self.x + rhs.x, y: self.y + rhs.y }
    }
}
▶ Run | Debug
fn main() {
    let v: Vector2D = Vector2D { x: 1.0, y: 2.0 };
    assert!(v * 10.0 == Vector2D { x: 10.0, y: 20.0 });
    assert!(v + v == Vector2D { x: 2.0, y: 4.0 });
}
```

Rust no longer moves our struct, it copies it :^)

This moves v, twice!

Intermission - Exercises

- Time for exercises!

```rust
trait Stats {
    fn get_weight(&self) -> f32;
    fn get_length(&self) -> f32;
}
1 implementation
struct Fish {
    kind: String,
    weight: f32,
    length: f32,
}
impl Stats for Fish {
    fn get_weight(&self) -> f32 {
        self.weight
    }
    fn get_length(&self) -> f32 {
        self.length
    }
}
pub fn main() {
    let fish: Fish = Fish {
        kind: String::from("Salmon"),
        weight: 10.0,
        length: 20.0,
    };
    println!("The {} is {}cm long and weighs {}kg",
             fish.kind, fish.get_length(), fish.get_weight());
}
```

2/3

```
trait Stats {
    fn get_weight(&self) -> f32;
    fn get_length(&self) -> f32;
}
1 implementation
struct Fish {
    kind: String,
    weight: f32,
    length: f32,
}
impl Stats for Fish {
    fn get_weight(&self) -> f32 {
        self.weight
    }
    fn get_length(&self) -> f32 {
        self.length
    }
}
pub fn main() {
    let fish: Fish = Fish {
        kind: String::from("Salmon"),
        weight: 10.0,
        length: 20.0,
    };
    println!("The {} is {}cm long and weighs {}kg",
            fish.kind, fish.get_length(), fish.get_weight());
}
```

Does the code compile?
If yes, what does it print?

2/3

```rust
trait Stats {
    fn get_weight(&self) -> f32;
    fn get_length(&self) -> f32;
}
1 implementation
struct Fish {
    kind: String,
    weight: f32,
    length: f32,
}
impl Stats for Fish {
    fn get_weight(&self) -> f32 {
        self.weight
    }
    fn get_length(&self) -> f32 {
        self.length
    }
}
pub fn main() {
    let fish: Fish = Fish {
        kind: String::from("Salmon"),
        weight: 10.0,
        length: 20.0,
    };
    println!("The {} is {}cm long and weighs {}kg",
            fish.kind, fish.get_length(), fish.get_weight());
}
```

Does the code compile?
If yes, what does it print?

Yes, it does compile!

2/3

```rust
trait Stats {
    fn get_weight(&self) -> f32;
    fn get_length(&self) -> f32;
}
1 implementation
struct Fish {
    kind: String,
    weight: f32,
    length: f32,
}
impl Stats for Fish {
    fn get_weight(&self) -> f32 {
        self.weight
    }
    fn get_length(&self) -> f32 {
        self.length
    }
}
pub fn main() {
    let fish: Fish = Fish {
        kind: String::from("Salmon"),
        weight: 10.0,
        length: 20.0,
    };
    println!("The {} is {}cm long and weighs {}kg",
            fish.kind, fish.get_length(), fish.get_weight());
}
```

Trait definition

Trait implementation

Does the code compile?
If yes, what does it print?

Yes, it does compile!

2/3

```rust
trait Stats {
    fn get_weight(&self) -> f32;
    fn get_length(&self) -> f32;
}
1 implementation
struct Fish {
    kind: String,
    weight: f32,
    length: f32,
}
impl Stats for Fish {
    fn get_weight(&self) -> f32 {
        self.weight
    }
    fn get_length(&self) -> f32 {
        self.length
    }
}
pub fn main() {
    let fish: Fish = Fish {
        kind: String::from("Salmon"),
        weight: 10.0,
        length: 20.0,
    };
    println!("The {} is {}cm long and weighs {}kg",
            fish.kind, fish.get_length(), fish.get_weight());
}
```

Does the code compile?
If yes, what does it print?

Yes, it does compile!

Output

```
The Salmon is 20cm long and weighs 10kg
```

3/3

```rust
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

```
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

Does the code compile?
If yes, what does it print?

3/3

```rust
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

Does the code compile?
If yes, what does it print?

Trait boundary
→ Anything that implements Printable must also implement Debug and Display

3/3

```rust
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

Does the code compile?
If yes, what does it print?

This is fine, Rust takes the default implementations given in the trait definition

3/3

```rust
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self);
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

However, our Point does not implement Debug and Display :(

Does the code compile?
If yes, what does it print?

```
use std::fmt::{Debug, Display};
1 implementation
trait Printable: Debug + Display {
    fn print_normal(&self) {
        println!("{}", self);
    }
    fn print_debug(&self) {
        println!("{:?}", self)
    }
}
1 implementation
struct Point {
    x: i32,
    y: i32,
}
impl Printable for Point {}

pub fn main() {
    let p: Point = Point { x: 10, y: 20 };
    p.print_normal();
    p.print_debug();
}
```

Does the code compile?
If yes, what does it print?

```
error[E0277]: `Point` doesn't implement `std::fmt::Display`
  --> src\exercise_2.rs:14:20
   |
14 | impl Printable for Point {}
   |                    ^^^^^ `Point` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Point`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-print) instead
note: required by a bound in `Printable`
  --> src\exercise_2.rs:2:26
   |
2  | trait Printable: Debug + Display {
   |                          ^^^^^^^ required by this bound in `Printable`

error[E0277]: `Point` doesn't implement `Debug`
  --> src\exercise_2.rs:14:20
   |
14 | impl Printable for Point {}
   |                    ^^^^^ `Point` cannot be formatted using `{:?}`
   |
   = help: the trait `Debug` is not implemented for `Point`
   = note: add `#[derive(Debug)]` to `Point` or manually `impl Debug for Point`
note: required by a bound in `Printable`
  --> src\exercise_2.rs:2:18
   |
2  | trait Printable: Debug + Display {
   |                  ^^^^^ required by this bound in `Printable`
```

```
2 implementations
trait Animal {
    fn get_name(&self) -> &String;
    fn make_sound(&self);
}
1 implementation
struct Cat { name: String }
1 implementation
struct Dog { name: String }
impl Animal for Cat {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Meow!"); }
}
impl Animal for Dog {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Woof!"); }
}
pub fn main() {
    let cat: Cat = Cat { name: String::from("Misty") };
    cat.make_sound();
    let dog: Dog = Dog { name: String::from("Rusty") };
    dog.make_sound();
}
```

```
2 implementations
trait Animal {
    fn get_name(&self) -> &String;
    fn make_sound(&self);
}
1 implementation
struct Cat { name: String }
1 implementation
struct Dog { name: String }
impl Animal for Cat {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Meow!"); }
}
impl Animal for Dog {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Woof!"); }
}
pub fn main() {
    let cat: Cat = Cat { name: String::from("Misty") };
    cat.make_sound();
    let dog: Dog = Dog { name: String::from("Rusty") };
    dog.make_sound();
}
```

Does the code compile?
If yes, what does it print?

2/3

Does the code compile?
If yes, what does it print?

```
2 implementations
trait Animal {
    fn get_name(&self) -> &String;
    fn make_sound(&self);
}
1 implementation
struct Cat { name: String }
1 implementation
struct Dog { name: String }
impl Animal for Cat {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Meow!"); }
}
impl Animal for Dog {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Woof!"); }
}
pub fn main() {
    let cat: Cat = Cat { name: String::from("Misty") };
    cat.make_sound();
    let dog: Dog = Dog { name: String::from("Rusty") };
    dog.make_sound();
}
```

Trait definition

Trait implementations

2/3

```
2 implementations
trait Animal {
    fn get_name(&self) -> &String;
    fn make_sound(&self);
}
1 implementation
struct Cat { name: String }
1 implementation
struct Dog { name: String }
impl Animal for Cat {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Meow!"); }
}
impl Animal for Dog {
    fn get_name(&self) -> &String { &self.name }
    fn make_sound(&self) { println!("Woof!"); }
}
pub fn main() {
    let cat: Cat = Cat { name: String::from("Misty") };
    cat.make_sound();
    let dog: Dog = Dog { name: String::from("Rusty") };
    dog.make_sound();
}
```

Does the code compile?
If yes, what does it print?

This code compiles :)

```
Meow!
Woof!
```

3/3

```rust
trait Forgettable {
    fn forget(&self);
}
2 implementations
struct Thing { name: String }
impl Thing {…
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
impl Forgettable for String {…
impl Forgettable for i32 {…
pub fn main() {
    let vector: Vec<Box<dyn Forgettable>> = vec![
        Box::new(String::from("hello")),
        Box::new(Thing::new("my thing")),
        Box::new(39),
    ];
    for elem: &Box<dyn Forgettable> in &vector {
        elem.forget();
    }
}
```

3/3

```
trait Forgettable {
    fn forget(&self);
}
2 implementations
struct Thing { name: String }
impl Thing {···
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
impl Forgettable for String {···
impl Forgettable for i32 {···
pub fn main() {
    let vector: Vec<Box<dyn Forgettable>> = vec![
        Box::new(String::from("hello")),
        Box::new(Thing::new("my thing")),
        Box::new(39),
    ];
    for elem: &Box<dyn Forgettable> in &vector {
        elem.forget();
    }
}
```

Does the code compile?
If yes, what does it print?

3/3

```rust
trait Forgettable {
    fn forget(&self);
}
2 implementations
struct Thing { name: String }
impl Thing {⋯
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
impl Forgettable for String {⋯
impl Forgettable for i32 {⋯
pub fn main() {
    let vector: Vec<Box<dyn Forgettable>> = vec![
        Box::new(String::from("hello")),
        Box::new(Thing::new("my thing")),
        Box::new(39),
    ];
    for elem: &Box<dyn Forgettable> in &vector {
        elem.forget();
    }
}
```

Does the code compile?
If yes, what does it print?

Rust has clear rules on what traits we can implement for which structs

→ Either you define a trait in your module, and then you can implement it for all structs

→ Or you define a struct in your module, and then you can implement all traits on it

→ You can‘t implement traits you didn‘t define for structs you didn‘t create

3/3

```rust
trait Forgettable {
    fn forget(&self);
}
```

Trait definition

```rust
2 implementations
struct Thing { name: String }
impl Thing {···
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
impl Forgettable for String {···
impl Forgettable for i32 {···
pub fn main() {
    let vector: Vec<Box<dyn Forgettable>> = vec![
        Box::new(String::from("hello")),
        Box::new(Thing::new("my thing")),
        Box::new(39),
    ];
    for elem: &Box<dyn Forgettable> in &vector {
        elem.forget();
    }
}
```

Does the code compile?
If yes, what does it print?

3/3

```rust
trait Forgettable {
    fn forget(&self);
}
2 implementations
struct Thing { name: String }
impl Thing {…
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
impl Forgettable for String {…
impl Forgettable for i32 {…
pub fn main() {
    let vector: Vec<Box<dyn Forgettable>> = vec![
        Box::new(String::from("hello")),
        Box::new(Thing::new("my thing")),
        Box::new(39),
    ];
    for elem: &Box<dyn Forgettable> in &vector {
        elem.forget();
    }
}
```

Does the code compile?
If yes, what does it print?

Trait implementations

3/3

```rust
trait Forgettable {
    fn forget(&self);
}
2 implementations
struct Thing { name: String }
impl Thing {···
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
impl Forgettable for String {···
impl Forgettable for i32 {···
pub fn main() {
    let vector: Vec<Box<dyn Forgettable>> = vec![
        Box::new(String::from("hello")),
        Box::new(Thing::new("my thing")),
        Box::new(39),
    ];
    for elem: &Box<dyn Forgettable> in &vector {
        elem.forget();
    }
}
```

Does the code compile?
If yes, what does it print?

Similar to how we can use impl Trait to make functions accept any struct, we can also use dyn Trait to make collections accept any struct that implements a trait

→ dyn means dynamic dispatch
→ We figure out at runtime where we need to jump

3/3

```rust
trait Forgettable {
    fn forget(&self);
}
2 implementations
struct Thing { name: String }
impl Thing {…
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
impl Forgettable for String {…
impl Forgettable for i32 {…
pub fn main() {
    let vector: Vec<Box<dyn Forgettable>> = vec![
        Box::new(String::from("hello")),
        Box::new(Thing::new("my thing")),
        Box::new(39),
    ];
    for elem: &Box<dyn Forgettable> in &vector {
        elem.forget();
    }
}
```

Does the code compile?
If yes, what does it print?

Because we do it dynamically and we can put any type in there, we don‘t know the size of dyn Forgettable
→ To use dyn, we need a container like Box

3/3

```rust
trait Forgettable {
    fn forget(&self);
}
2 implementations
struct Thing { name: String }
impl Thing {…
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
impl Forgettable for String {…
impl Forgettable for i32 {…
pub fn main() {
    let vector: Vec<Box<dyn Forgettable>> = vec![
        Box::new(String::from("hello")),
        Box::new(Thing::new("my thing")),
        Box::new(39),
    ];
    for elem: &Box<dyn Forgettable> in &vector {
        elem.forget();
    }
}
```

Does the code compile?
If yes, what does it print?

All of the three underlying types do implement Forgettable, so this part is fine

3/3

```rust
trait Forgettable {
    fn forget(&self);
}
2 implementations
struct Thing { name: String }
impl Thing {…
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
impl Forgettable for String {…
impl Forgettable for i32 {…
pub fn main() {
    let vector: Vec<Box<dyn Forgettable>> = vec![
        Box::new(String::from("hello")),
        Box::new(Thing::new("my thing")),
        Box::new(39),
    ];
    for elem: &Box<dyn Forgettable> in &vector {
        elem.forget();
    }
}
```

We call the correct forget implementation at runtime

Does the code compile?
If yes, what does it print?

3/3

```rust
trait Forgettable {
    fn forget(&self);
}
2 implementations
struct Thing { name: String }
impl Thing {…
impl Forgettable for Thing {
    fn forget(&self) {
        println!("I'm forgetting {}", self.name);
    }
}
impl Forgettable for String {…
impl Forgettable for i32 {…
pub fn main() {
    let vector: Vec<Box<dyn Forgettable>> = vec![
        Box::new(String::from("hello")),
        Box::new(Thing::new("my thing")),
        Box::new(39),
    ];
    for elem: &Box<dyn Forgettable> in &vector {
        elem.forget();
    }
}
```

We call the correct forget implementation at runtime

Does the code compile?
If yes, what does it print?

This code compiles :)

```
I'm forgetting hello
I'm forgetting my thing
I'm forgetting 39
```

3. Next time

- Enums
- Pattern Matching