Lecture 1: Introduction

Patterns, methods, traits and generics

Willem Vanhulle

DevLab Rust 2025

Tuesday November 4, 2025

What is the meaning of the word ‘Rust’ in this context?

What is the meaning of the word ‘Rust’ in this context?

It’s the name of a kind of organism: robust ‘funghi’

(Don’t listen to your Flemish grandma, it is **not chilling**, it’s “Roest” in Dutch.)

1.1. Participants

Round of introductions:

- What is your name?
- What would you like to learn?

1.1. Participants

Round of introductions:

- What is your name?
- What would you like to learn?
  - Microcontrollers?
  - Web servers?
  - Robotics?

1.1. Participants

Round of introductions:

- What is your name?
- What would you like to learn?
  - Microcontrollers?
  - Web servers?
  - Robotics?
- What is your experience with programming?
- What is your experience with Rust?

1.2. Me

1.2.1. Experience

Teaching experience:

- Mathematics tutor for 5 years (students 10-25 years old): analysis, statics, algebra, calculus, ...
- Part-time mathematics teacher at adult school for 1 year
- Organising internal Rust workshops at software companies (2 years)
- Organising systems programming workshops at SysGhent.be (1 year)

1.2. Me

1.2.1. Experience

Teaching experience:

- Mathematics tutor for 5 years (students 10-25 years old): analysis, statics, algebra, calculus, …
- Part-time mathematics teacher at adult school for 1 year
- Organising internal Rust workshops at software companies (2 years)
- Organising systems programming workshops at SysGhent.be (1 year)

Rust experience:

- Control small-scale fermentation robot and peripherals with Rust (1 year)
- Architect and implement remote control system for freight trains in Rust (1 year)
- Various hobby projects and conference talk

1.2. Me

1.2.1. Experience

Teaching experience:

- Mathematics tutor for 5 years (students 10-25 years old): analysis, statics, algebra, calculus, ...
- Part-time mathematics teacher at adult school for 1 year
- Organising internal Rust workshops at software companies (2 years)
- Organising systems programming workshops at SysGhent.be (1 year)

Rust experience:

- Control small-scale fermentation robot and peripherals with Rust (1 year)
- Architect and implement remote control system for freight trains in Rust (1 year)
- Various hobby projects and conference talk

Contact information:

- Questions: willemvanhulle@protonmail.com
- Website: willemvanhulle.tech, github.com/wvhulle

1.2. Me

1.2.2. Hobby projects

Latest Rust hobby projects: `nu-lint` (linter NuShell) and a `firewall-manager`

Info

Nushell is a modern shell written in Rust.

Makes pipelines easier with modern syntax and types:

```
1   date now                    # 1: today
2   | $in + 1day                # 2: tomorrow
3   | format date '%F'          # 3: Format as YYYY-MM-DD
4   | $'($in) Report'           # 4: Format the directory name
5   | mkdir $in                 # 5: Create the directory          (playground link)
```

1.2.2. Hobby projects

Latest Rust hobby projects: `nu-lint` (linter NuShell) and a `firewall-manager`

Info

Nushell is a modern shell written in Rust.

Makes pipelines easier with modern syntax and types:

```
1    date now                    # 1: today
2    | $in + 1day                # 2: tomorrow
3    | format date '%F'          # 3: Format as YYYY-MM-DD
4    | $'($in) Report'           # 4: Format the directory name
5    | mkdir $in                 # 5: Create the directory          (playground link)
```

Free Nu workshop

On Wed, 3rd of December, I will give a free workshop on NuShell in this location. Register at SysGent meetup page.

1.3. Misconceptions about Rust

It's too difficult.

1.3. Misconceptions about Rust

It's too difficult.

No, it is only hard in the first 1 month.
After that, you will experience relief. You will be able to sleep at night (no more segfaults!).

It's not used in industry.

1.3. Misconceptions about Rust

It's too difficult.

No, it is only hard in the first 1 month.
After that, you will experience relief. You will be able to sleep at night (no more segfaults!).

It's not used in industry.

No, big companies are actually using it: Google, Amazon, Microsoft.
Near Ghent: Robovision, Barco, OTIV, … . In coming years more and more: Keysight, Infrabel, …

Rust is only for systems programming.

1.3. Misconceptions about Rust

It's too difficult.

No, it is only hard in the first 1 month.
After that, you will experience relief. You will be able to sleep at night (no more segfaults!).

It's not used in industry.

No, big companies are actually using it: Google, Amazon, Microsoft.
Near Ghent: Robovision, Barco, OTIV, … . In coming years more and more: Keysight, Infrabel, …

Rust is only for systems programming.

No, you can build:

- Web applications (backend and frontend)
- Command-line tools
- Data science and machine learning

1.4. Ultimate goal

By the end of this course, you should be able to **publish your own code** to the Rust package registry crates.io.

Warning

Start thinking now what you would like to build and let me know before next session!

crates.io

Browse All Crates | Log in with GitHub

The Rust community's crate registry

Type 'S' or '/' to search

Install Cargo | Getting Started

Instantly publish your crates and install them. Use the API to interact and find out more information about available crates. Become a contributor and enhance the site with your work.

190,677,256,300
Downloads

203,226
Crates in stock

New Crates	Most Downloaded	Just Updated
rat_net_cmd v0.1.0	syn	vtcode-exec-events v0.38.0
xatlas-rs-v2 v0.1.4	hashbrown	vtcode-config v0.38.0

1.5. Learning material

Theory:

- These **slides contain extra diagrams**:
  - https://github.com/wvhulle/rust-course-ghent
  - written in Typst (a Rust based typesetting language like LaTeX)
- Book: “Programming Rust” by Jim Blandy, any edition ($\sim$ 35€, ask me for e-book)

1.5. Learning material

Theory:

- These **slides contain extra diagrams**:
  - https://github.com/wvhulle/rust-course-ghent
  - written in Typst (a Rust based typesetting language like LaTeX)
- Book: "Programming Rust" by Jim Blandy, any edition ($\sim$ 35€, ask me for e-book)

Exercises:

- Exercise from Google's Rust course
- More exercises https://exercism.org/tracks/rust
- Lots of official documentation https://doc.rust-lang.org/std/index.html

Our aim in this course: $>$ **50% exercises!**

Crate std

Since 1.0.0 · Source

The Rust Standard Library

The Rust Standard Library is the foundation of portable Rust software, a set of minimal and battle-tested shared abstractions for the broader Rust ecosystem. It offers core types, like `Vec<T>` and `Option<T>`, library-defined operations on language primitives, standard macros, I/O and multithreading, among many other things.

`std` is available to all Rust crates by default. Therefore, the standard library can be accessed in `use` statements through the path `std`, as in `use std::env`.

How to read this documentation

If you already know the name of what you are looking for, the fastest way to find it is to use the search button at the top of the page.

Otherwise, you may want to jump to one of these useful sections:

- `std::*` modules
- Primitive types
- Standard macros
- The Rust Prelude

If this is your first time, the documentation for the standard library is written to be casually perused. Clicking on interesting things

1.6. Setup for exercises

You don't have to install Rust for this first lecture:

- Solve during lectures in playground https://play.rust-lang.org
- Code samples have a "playground link" at the bottom right

1.6. Setup for exercises

You don't have to install Rust for this first lecture:

- Solve during lectures in playground https://play.rust-lang.org
- Code samples have a "playground link" at the bottom right

Do you prefer to do exercises locally?

- Find copies of exercises in the `session-N/examples/` folders
- Run them with `cargo run --example s1e1-fibonacci` (from root or session folder).

1.6. Setup for exercises

You don't have to install Rust for this first lecture:

- Solve during lectures in playground https://play.rust-lang.org
- Code samples have a "playground link" at the bottom right

Do you prefer to do exercises locally?

- Find copies of exercises in the `session-N/examples/` folders
- Run them with `cargo run --example s1e1-fibonacci` (from root or session folder).

Still need harder challenges:

- Find additional exercises on https://exercism.org/tracks/rust/exercises.
- Work on your own project!

src/lib.rs | Cargo.toml

```rust
// The code below is a stub. Just enough to satisfy the compiler.
// In order to pass the tests you can add-to or change any of this code.

#[derive(Debug)]
pub struct Duration;

impl From<u64> for Duration {
    fn from(s: u64) -> Self {
        todo!("s, measured in seconds: {s}")
    }
}

pub trait Planet {
    fn years_during(d: &Duration) -> f64 {
        todo!("convert a duration ({d:?}) to the number of years on this planet for that duration");
    }
}
```

Stuck? Get help | Run Tests | Submit

Instructions | Tests | Results | ChatGPT

```rust
use space_age::*;

fn assert_in_delta(expected: f64, actual: f64) {
    let diff: f64 = (expected - actual).abs();
    let delta: f64 = 0.01;
    if diff > delta {
        panic!("Your result of {actual} should be within {delta} of the expected result {expected}")
    }
}

#[test]
fn age_on_earth() {
    let seconds = 1_000_000_000;
    let duration = Duration::from(seconds);
    let output = Earth::years_during(&duration);
    let expected = 31.69;
    assert_in_delta(expected, output);
}
```

1.7. Course contents

Each session is 2 hours long. Longer break in the middle. Half theory, half exercises.

1. Introduction, **rust basics, patterns, methods, traits, generics** (+ homework)
2. Functional programming & standard library
3. Ownership & borrowing fundamentals
4. Smart pointers & memory management
5. Iterators, modules & testing
6. Error handling & concurrency
7. Asynchronous programming

1.8. Rust today

Part 1: Rust fundamentals:

- Types and values
- Control flow basics
- Tuples and arrays
- References
- User-defined types

1.8. Rust today

Part 1: Rust fundamentals:

- Types and values
- Control flow basics
- Tuples and arrays
- References
- User-defined types

Part 2: Important Rust concepts:

- Pattern matching
- Methods
- Traits (maybe at home)
- Generics (maybe at home)

2.1. Hello, world!

```rust
fn main() {
    println!("Hello, world!");
}
```

(playground link)

2.1. Hello, world!

```
1 fn main() {
2     println!("Hello, world!");
3 }
```
(playground link)

- `fn` defines a function
- Statements end with `;`
- `main` is the entry point
- Rust strings are UTF-8 encoded and can contain any Unicode character.

2.1. Hello, world!

```
1 fn main() {
2     println!("Hello, world!");
3 }
```
(playground link)

- `fn` defines a function
- Statements end with `;`
- `main` is the entry point
- Rust strings are UTF-8 encoded and can contain any Unicode character.

What does the `!` in `println!` signify?

2.1. Hello, world!

```
1 fn main() {
2     println!("Hello, world!");
3 }
```
(playground link)

- `fn` defines a function
- Statements end with `;`
- `main` is the entry point
- Rust strings are UTF-8 encoded and can contain any Unicode character.

What does the `!` in `println!` signify?

It's a macro, not a function. Macros expand at compile time and can take variable arguments.

2.1. Hello, world!

```
1 fn main() {
2     println!("Hello, world!");
3 }
```
(playground link)

- `fn` defines a function
- Statements end with `;`
- `main` is the entry point
- Rust strings are UTF-8 encoded and can contain any Unicode character.

What does the `!` in `println!` signify?

It's a macro, not a function. Macros expand at compile time and can take variable arguments.

Can you call `println!` without any arguments?

2.1. Hello, world!

```
1 fn main() {
2     println!("Hello, world!");
3 }
```
(playground link)

- `fn` defines a function
- Statements end with `;`
- `main` is the entry point
- Rust strings are UTF-8 encoded and can contain any Unicode character.

What does the `!` in `println!` signify?

It's a macro, not a function. Macros expand at compile time and can take variable arguments.

Can you call `println!` without any arguments?

No, it requires at least a format string: `println!("")` or use `println!()` in newer Rust versions.

2.2. Variables

Type safety with static typing:

```rust
fn main() {
    let x: i32 = 10;
    println!("x: {x}");
    // x = 20;
    // println!("x: {x}");
}
```

(playground link)

2.2. Variables

Type safety with static typing:

```
1 fn main() {
2     let x: i32 = 10;
3     println!("x: {x}");
4     // x = 20;
5     // println!("x: {x}");
6 }
```

(playground link)

What happens if we uncomment the two lines?

2.2. Variables

Type safety with static typing:

```
1 fn main() {
2     let x: i32 = 10;
3     println!("x: {x}");
4     // x = 20;
5     // println!("x: {x}");
6 }
```

(playground link)

What happens if we uncomment the two lines?

Variables are immutable by default. Use `mut` to make them mutable.

2.2. Variables

Type safety with static typing:

```
1 fn main() {
2     let x: i32 = 10;
3     println!("x: {x}");
4     // x = 20;
5     // println!("x: {x}");
6 }
```

(playground link)

What happens if we uncomment the two lines?

Variables are immutable by default. Use `mut` to make them mutable.

Are type annotations required?

2.2. Variables

Type safety with static typing:

```
1 fn main() {
2     let x: i32 = 10;
3     println!("x: {x}");
4     // x = 20;
5     // println!("x: {x}");
6 }
```

(playground link)

What happens if we uncomment the two lines?

Variables are immutable by default. Use `mut` to make them mutable.

Are type annotations required?

Not required for local variables (if the compiler **can infer**).

2.3. Values

Basic built-in types:

- Signed integers: `i8`, `i16`, `i32`, `i64`, `i128`, `isize`
- Unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128`, `usize`

2.3. Values

Basic built-in types:

- Signed integers: `i8`, `i16`, `i32`, `i64`, `i128`, `isize`
- Unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128`, `usize`

When should you use `isize` and `usize`?

2.3. Values

Basic built-in types:

- Signed integers: `i8`, `i16`, `i32`, `i64`, `i128`, `isize`
- Unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128`, `usize`

When should you use `isize` and `usize`?

Use `usize` for pointers and array indices.

2.3. Values

Basic built-in types:

- Signed integers: `i8`, `i16`, `i32`, `i64`, `i128`, `isize`
- Unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128`, `usize`

When should you use `isize` and `usize`?

Use `usize` for pointers and array indices.

- Floating point: `f32`, `f64`
- Boolean: `bool` (`true` or `false`)

2.3. Values

Basic built-in types:

- Signed integers: `i8`, `i16`, `i32`, `i64`, `i128`, `isize`
- Unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128`, `usize`

When should you use `isize` and `usize`?

Use `usize` for pointers and array indices.

- Floating point: `f32`, `f64`
- Boolean: `bool` (`true` or `false`)

2.3. Values

Does this compile?

```
1 fn main() {
2     let x: i32 = 5;
3     let y: i64 = x;
4 }
```

(playground link)

Can you compare values of different numeric types directly (e.g., `i32` and `i64`)?

2.3. Values

Does this compile?

```
1 fn main() {
2     let x: i32 = 5;
3     let y: i64 = x;
4 }
```

(playground link)

Can you compare values of different numeric types directly (e.g., `i32` and `i64`)?

No! Rust has no implicit type conversions. You must use explicit casts with `as`: `x as i64 == y`

2.3. Values

Does this compile?

```
1 fn main() {
2     let x: i32 = 5;
3     let y: i64 = x;
4 }
```

(playground link)

Can you compare values of different numeric types directly (e.g., `i32` and `i64`)?

No! Rust has no implicit type conversions. You must use explicit casts with `as`: `x as i64 == y`

- Character: `char`, written `'x'` (Unicode scalar value, 4 bytes)

Warning

Every cast that may lose precision must be explicit with `as`!

2.4. Arithmetic

```
1 fn interproduct(a: i32, b: i32, c: i32) -> i32 {
2   return a * b + b * c + c * a;
3 }
4
5 fn main() {
6     println!("result: {}", interproduct(120, 100, 248));
7 }
```

(playground link)

2.4. Arithmetic

```
1  fn interproduct(a: i32, b: i32, c: i32) -> i32 {
2      return a * b + b * c + c * a;
3  }
4
5  fn main() {
6      println!("result: {}", interproduct(120, 100, 248));
7  }
```

(playground link)

What happens when integer overflow occurs?

2.4. Arithmetic

```
1 fn interproduct(a: i32, b: i32, c: i32) -> i32 {
2   return a * b + b * c + c * a;
3 }
4
5 fn main() {
6     println!("result: {}", interproduct(120, 100, 248));
7 }
```

(playground link)

What happens when integer overflow occurs?

It is defined behavior in release mode (wrap around), but panics in debug mode.

2.5. Question

```
1 fn main() {
2     let mut x = 4;
3     --x;
4     print!("{}{}", --x, --x);
5 }
```

(playground link)

What is the output of this program?

2.5. Question

```
1 fn main() {
2     let mut x = 4;
3     --x;
4     print!("{}{}", --x, --x);
5 }
```

(playground link)

What is the output of this program?

Short answer: 4. Longer answer: Rust does not have a unary increment or decrement operator.

2.6. Type Inference

```
1    fn takes_u32(x: u32) {
2      println!("u32: {x}");
3    }
4
5    fn takes_i8(y: i8) {
6        println!("i8: {y}");
7    }
8
9    fn main() {
10       let x = 10;
11       let y = 20;
12
13       takes_u32(x);
14       takes_i8(y);
15       // takes_u32(y);
16   }                                   (playground link)
```

2.6. Type Inference

```
1   fn takes_u32(x: u32) {
2       println!("u32: {x}");
3   }
4
5   fn takes_i8(y: i8) {
6       println!("i8: {y}");
7   }
8
9   fn main() {
10      let x = 10;
11      let y = 20;
12
13      takes_u32(x);
14      takes_i8(y);
15      // takes_u32(y);
16  }                                   (playground link)
```

Does this code compile?

2.6. Type Inference

```
1   fn takes_u32(x: u32) {
2       println!("u32: {x}");
3   }
4
5   fn takes_i8(y: i8) {
6       println!("i8: {y}");
7   }
8
9   fn main() {
10      let x = 10;
11      let y = 20;
12
13      takes_u32(x);
14      takes_i8(y);
15      // takes_u32(y);
16  }                                   (playground link)
```

Does this code compile?

Yes, Rust infers `x` as `u32` and `y` as `i8`.

2.6. Type Inference

```
1   fn takes_u32(x: u32) {
2       println!("u32: {x}");
3   }
4
5   fn takes_i8(y: i8) {
6       println!("i8: {y}");
7   }
8
9   fn main() {
10      let x = 10;
11      let y = 20;
12
13      takes_u32(x);
14      takes_i8(y);
15      // takes_u32(y);
16  }
```

(playground link)

Does this code compile?

Yes, Rust infers `x` as `u32` and `y` as `i8`.

What happens if we uncomment the last line?

2.6. Type Inference

```
1   fn takes_u32(x: u32) {
2       println!("u32: {x}");
3   }
4
5   fn takes_i8(y: i8) {
6       println!("i8: {y}");
7   }
8
9   fn main() {
10      let x = 10;
11      let y = 20;
12
13      takes_u32(x);
14      takes_i8(y);
15      // takes_u32(y);
16  }                                   (playground link)
```

Does this code compile?

Yes, Rust infers `x` as `u32` and `y` as `i8`.

What happens if we uncomment the last line?

It does not compile: type mismatch.

2.6. Type Inference

```
1   fn takes_u32(x: u32) {
2       println!("u32: {x}");
3   }
4
5   fn takes_i8(y: i8) {
6       println!("i8: {y}");
7   }
8
9   fn main() {
10      let x = 10;
11      let y = 20;
12
13      takes_u32(x);
14      takes_i8(y);
15      // takes_u32(y);
16  }
```

(playground link)

Does this code compile?

Yes, Rust infers `x` as `u32` and `y` as `i8`.

What happens if we uncomment the last line?

It does not compile: type mismatch.

What is the default integer type in Rust?

2.6. Type Inference

```
1   fn takes_u32(x: u32) {
2       println!("u32: {x}");
3   }
4
5   fn takes_i8(y: i8) {
6       println!("i8: {y}");
7   }
8
9   fn main() {
10      let x = 10;
11      let y = 20;
12
13      takes_u32(x);
14      takes_i8(y);
15      // takes_u32(y);
16  }
```

(playground link)

Does this code compile?

Yes, Rust infers `x` as `u32` and `y` as `i8`.

What happens if we uncomment the last line?

It does not compile: type mismatch.

What is the default integer type in Rust?

It is `i32` unless otherwise specified.

2.6. Type Inference

Who was Pingala?

2.6. Type Inference

Who was Pingala?

An ancient Indian scholar (circa 200 BC) who described the Fibonacci sequence in his work on prosody.

Info

The Fibonacci sequence begins with [0, 1]. For n > 1, the next number is the sum of the previous two.

Complete the code in `session-1/examples/s1e1-fib.rs`.

2.6. Type Inference

2.6.1. Follow-up questions

When will this function panic?

2.6. Type Inference

2.6.1. Follow-up questions

When will this function panic?

- When `n` is too large and the result overflows `u32`.
- When recursion depth exceeds the stack size (around 10,000).

2.6. Type Inference

2.6.1. Follow-up questions

When will this function panic?

- When `n` is too large and the result overflows `u32`.
- When recursion depth exceeds the stack size (around 10,000).

How to fix these issues?

2.6. Type Inference

2.6.1. Follow-up questions

When will this function panic?

- When `n` is too large and the result overflows `u32`.
- When recursion depth exceeds the stack size (around 10,000).

How to fix these issues?

- Make it iterative.
- Use `u128` or external `bigint` crate for large Fibonacci numbers.

3.1. Blocks and scopes

```
1  fn main() {
2      let z = 13;
3      let x = {
4          let y = 10;
5          dbg!(y);
6          z - y
7      };
8      dbg!(x);
9      // dbg!(y);
10 }                          (playground link)
```

3.1. Blocks and scopes

```
1  fn main() {
2      let z = 13;
3      let x = {
4          let y = 10;
5          dbg!(y);
6          z - y
7      };
8      dbg!(x);
9      // dbg!(y);
10 }                          (playground link)
```

What is the output of this program?

3.1. Blocks and scopes

```
1  fn main() {
2      let z = 13;
3      let x = {
4          let y = 10;
5          dbg!(y);
6          z - y
7      };
8      dbg!(x);
9      // dbg!(y);
10 }                          (playground link)
```

What is the output of this program?

The output is `x = 3`.

3.1. Blocks and scopes

```
1  fn main() {
2      let z = 13;
3      let x = {
4          let y = 10;
5          dbg!(y);
6          z - y
7      };
8      dbg!(x);
9      // dbg!(y);
10 }                                (playground link)
```

What is the output of this program?

The output is `x = 3`.

What happens if we uncomment the last line?

3.1. Blocks and scopes

```
1  fn main() {
2      let z = 13;
3      let x = {
4          let y = 10;
5          dbg!(y);
6          z - y
7      };
8      dbg!(x);
9      // dbg!(y);
10 }                                (playground link)
```

What is the output of this program?

The output is `x = 3`.

What happens if we uncomment the last line?

Would not compile because `y` is out of scope: de-allocated.

3.1. Blocks and scopes

```
1  fn main() {
2      let z = 13;
3      let x = {
4          let y = 10;
5          dbg!(y);
6          z - y
7      };
8      dbg!(x);
9      // dbg!(y);
10 }                                (playground link)
```

What is the output of this program?

The output is `x = 3`.

What happens if we uncomment the last line?

Would not compile because `y` is out of scope: de-allocated.

Info

Rust is an **expression-based** language: almost everything is an expression that returns a value.

3.2. If expressions

```
1  fn main() {
2      let x = 10;
3      if x == 0 {
4          println!("zero!");
5      } else if x < 100 {
6          println!("biggish");
7      } else {
8          println!("huge");
9      }
10 }
```

(playground link)

3.2. If expressions

```
1 fn main() {
2     let x = 10;
3     let size = if x < 20 { "small" } else { "large" };
4     println!("number size: {}", size);
5 }
```

(playground link)

What happens when we put a semicolon (`;`) behind the `"small"` string?

3.2. If expressions

```
1 fn main() {
2     let x = 10;
3     let size = if x < 20 { "small" } else { "large" };
4     println!("number size: {}", size);
5 }
```

(playground link)

What happens when we put a semicolon (`;`) behind the `"small"` string?

It does not compile.

Warning

Semi-colons at the end of block **suppress block return values**.

3.3. Match expressions

```rust
fn main() {
    let val = 1;
    match val {
        1 => println!("one"),
        10 => println!("ten"),
        100 => println!("one hundred"),
        _ => {
            println!("something else");
        }
    }
}
```

(playground link)

3.3. Match expressions

```
1  fn main() {
2      let val = 1;
3      match val {
4          1 => println!("one"),
5          10 => println!("ten"),
6          100 => println!("one hundred"),
7          _ => {
8              println!("something else");
9          }
10     }
11 }                                    (playground link)
```

What happens without the _ case?

3.3. Match expressions

```
1  fn main() {
2      let val = 1;
3      match val {
4          1 => println!("one"),
5          10 => println!("ten"),
6          100 => println!("one hundred"),
7          _ => {
8              println!("something else");
9          }
10     }
11 }
```

(playground link)

What happens without the _ case?

It does not compile: non-exhaustive patterns.

Warning

All possible cases must be handled in a `match` expression (exhaustiveness is required).

3.3. Match expressions

```rust
fn main() {
    let flag = true;
    let val = match flag {
        true => 1,
        false => 0,
    };
    println!("The value of {flag} is {val}");
}
```

(playground link)

3.3. Match expressions

```rust
fn main() {
    let flag = true;
    let val = match flag {
        true => 1,
        false => 0,
    };
    println!("The value of {flag} is {val}");
}
```

(playground link)

Info

Every `match` is also an expression that returns a value.

3.3. Match expressions

```rust
fn main() {
    let flag = true;
    let val = match flag {
        true => 1,
        false => 0,
    };
    println!("The value of {flag} is {val}");
}
```

(playground link)

Info

Every `match` is also an expression that returns a value.

Warning

- All blocks are expressions! Arms may be blocks.
- Do not put semi-colons at the end of arms if you want to return a value!

3.4. Loops

```rust
fn main() {
    let mut x = 200;
    while x >= 10 {
        x = x / 2;
    }
    dbg!(x);
}
```

(playground link)

Info

`break`, `continue` and `return` work as expected. `loop` is `while true`.

3.4. Loops

```rust
fn main() {
    for x in 1..5 {
        dbg!(x);
    }

    for elem in [2, 4, 8, 16, 32] {
        dbg!(elem);
    }
}
```

(playground link)

3.4. Loops

```rust
fn main() {
    for x in 1..5 {
        dbg!(x);
    }

    for elem in [2, 4, 8, 16, 32] {
        dbg!(elem);
    }
}
```

(playground link)

What can we write inside the `in` of a `for` loop?

3.4. Loops

```
1 fn main() {
2     for x in 1..5 {
3         dbg!(x);
4     }
5
6     for elem in [2, 4, 8, 16, 32] {
7         dbg!(elem);
8     }
9 }
```

(playground link)

What can we write inside the `in` of a `for` loop?

Any iterable type. Implemented with `IntoIterator` trait (see later).

3.5. Functions

```
1  fn gcd(a: u32, b: u32) -> u32 {
2      if b > 0 { gcd(b, a % b) } else { a }
3  }
4
5  fn main() {
6      dbg!(gcd(143, 52));
7  }
```

(playground link)

What is the return type of `main`?

3.5. Functions

```
1  fn gcd(a: u32, b: u32) -> u32 {
2      if b > 0 { gcd(b, a % b) } else { a }
3  }
4
5  fn main() {
6      dbg!(gcd(143, 52));
7  }
```

(playground link)

What is the return type of `main`?

It is `()`, the **unit type** (like `void` in C/C++) and optional.

3.5. Functions

```
1  fn gcd(a: u32, b: u32) -> u32 {
2      if b > 0 { gcd(b, a % b) } else { a }
3  }
4
5  fn main() {
6      dbg!(gcd(143, 52));
7  }
```

(playground link)

What is the return type of `main`?

It is `()`, the **unit type** (like `void` in C/C++) and optional.

How would you achieve function overloading in Rust?

3.5. Functions

```
1  fn gcd(a: u32, b: u32) -> u32 {
2      if b > 0 { gcd(b, a % b) } else { a }
3  }
4
5  fn main() {
6      dbg!(gcd(143, 52));
7  }
```

(playground link)

What is the return type of `main`?

It is `()`, the **unit type** (like `void` in C/C++) and optional.

How would you achieve function overloading in Rust?

Rust doesn't support traditional overloading. Instead, use: different names, generic functions with trait bounds, or method overloading through traits like `Add` or `From`.

3.6. Macros

- `println!(format)` prints to standard output with formatting.
- `dbg!(expr)` prints the value of `expr` to standard error for debugging.
- `todo!()` panics with a “not yet implemented” message.

3.6. Macros

- `println!(format)` prints to standard output with formatting.
- `dbg!(expr)` prints the value of `expr` to standard error for debugging.
- `todo!()` panics with a "not yet implemented" message.

Which macros should you use in tests?

3.6. Macros

- `println!(format)` prints to standard output with formatting.
- `dbg!(expr)` prints the value of `expr` to standard error for debugging.
- `todo!()` panics with a "not yet implemented" message.

Which macros should you use in tests?

- `assert!(condition)` to check if a condition is true.
- `assert_eq!(a, b)` to check if two values are equal.

```
1  fn factorial(n: u32) -> u32 {
2      let mut product = 1;
3      for i in 1..=n {
4          product *= dbg!(i);
5      }
6      product
7  }
8  fn main() {
9      let n = 4;
10     println!("{n}! = {}",
       factorial(n));
11 }                          (playground link)
```

3.7. Exercise: Collatz sequence

Solve the exercise about Collatz sequences by completing the code in `session-1/examples/s1e2-collatz.rs`.

4.1. Arrays

Info

Arrays are fixed-size, **fixed-length contiguous collections** located on the stack

4.1. Arrays

Info

Arrays are fixed-size, **fixed-length contiguous collections** located on the stack

```rust
fn main() {
    let mut a: [i8; 5] = [5, 4, 3, 2, 1];
    a[2] = 0;
    println!("a: {a:?}");
}
```

(playground link)

4.1. Arrays

Info

Arrays are fixed-size, **fixed-length contiguous collections** located on the stack

```
1 fn main() {
2     let mut a: [i8; 5] = [5, 4, 3, 2, 1];
3     a[2] = 0;
4     println!("a: {a:?}");
5 }
```

(playground link)

Can you omit the length of the array in the type annotation?

4.1. Arrays

Info

Arrays are fixed-size, **fixed-length contiguous collections** located on the stack

```
1 fn main() {
2     let mut a: [i8; 5] = [5, 4, 3, 2, 1];
3     a[2] = 0;
4     println!("a: {a:?}");
5 }
```

(playground link)

Can you omit the length of the array in the type annotation?

Yes, the compiler can **infer it from the initializer**. In other cases, it is required.

4.1. Arrays

```
1 fn main() {
2     let mut a: [i8; 5] = [5, 4, 3, 2, 1];
3     a[6] = 0;
4     println!("a: {a:?}");
5 }
```

(playground link)

What happens when we run this program?

4.1. Arrays

```
1 fn main() {
2     let mut a: [i8; 5] = [5, 4, 3, 2, 1];
3     a[6] = 0;
4     println!("a: {a:?}");
5 }
```

(playground link)

What happens when we run this program?

It panics at runtime: index out of bounds. Rust performs **bounds checking** on array accesses at runtime.

4.1. Arrays

```
1 fn main() {
2     let mut a: [i8; 5] = [5, 4, 3, 2, 1];
3     a[6] = 0;
4     println!("a: {a:?}");
5 }
```

(playground link)

What happens when we run this program?

It panics at runtime: index out of bounds. Rust performs **bounds checking** on array accesses at runtime.

Warning

If possible, the compiler omits the bound checks. They may also be disabled explicitly by user.

4.1. Arrays

Have a look at this code:

```
1 fn get_index() -> usize { 6 }
2 fn main() {
3     let mut a: [i8; 5] = [5, 4, 3, 2, 1];
4     a[get_index()] = 0;
5     println!("a: {a:?}");
6 }
```

(playground link)

Will the compiler implement run-time bounds checks?

4.1. Arrays

Have a look at this code:

```
1 fn get_index() -> usize { 6 }
2 fn main() {
3     let mut a: [i8; 5] = [5, 4, 3, 2, 1];
4     a[get_index()] = 0;
5     println!("a: {a:?}");
6 }
```

(playground link)

Will the compiler implement run-time bounds checks?

Yes, because the index is not known at compile time.

4.1. Arrays

Have a look at this code:

```
1 fn get_index() -> usize { 6 }
2 fn main() {
3     let mut a: [i8; 5] = [5, 4, 3, 2, 1];
4     a[get_index()] = 0;
5     println!("a: {a:?}");
6 }
```

(playground link)

Will the compiler implement run-time bounds checks?

Yes, because the index is not known at compile time.

Info

You could write `const fn get_index() -> usize { 6 }` to make it a compile-time constant and remove the runtime bounds check.

4.1. Arrays

Info

All arrays have a fixed size known at compile time.

4.1. Arrays

Info

All arrays have a fixed size known at compile time.

Warning

All arrays are stored on the stack by default.

Info

All arrays have a fixed size known at compile time.

Warning

All arrays are stored on the stack by default.

Can the length of an array be modified at runtime?

4.1. Arrays

Info

All arrays have a fixed size known at compile time.

Warning

All arrays are stored on the stack by default.

Can the length of an array be modified at runtime?

No, but you can have arrays with mutable elements.

```
1 let mut a: [i8; 5] = [5, 4, 3, 2, 1];
2 a[2] = 0;                                    (playground link)
```

4.2. Debug output

Printing arrays for debugging:

```
1 fn main() {
2     let a: [i8; 5] = [5, 4, 3, 2, 1];
3     println!("a: {:?}", a);
4 }
```

(playground link)

4.2. Debug output

Printing arrays for debugging:

```
1 fn main() {
2     let a: [i8; 5] = [5, 4, 3, 2, 1];
3     println!("a: {:?}", a);
4 }
```

(playground link)

What does the `:?` format specifier do?

4.2. Debug output

Printing arrays for debugging:

```
1 fn main() {
2     let a: [i8; 5] = [5, 4, 3, 2, 1];
3     println!("a: {:?}", a);
4 }
```

(playground link)

What does the `:?` format specifier do?

It uses the `Debug` trait to format the value for debugging purposes. (Also used by `dbg!` and `eprintln!` macros.)

4.3. Tuples

Grouping values (of different types) without field names:

```
1 fn main() {
2     let t: (i8, bool) = (7, true);
3     dbg!(t.0);
4     dbg!(t.1);
5 }                                                    (playground link)
```

4.3. Tuples

Grouping values (of different types) without field names:

```rust
fn main() {
    let t: (i8, bool) = (7, true);
    dbg!(t.0);
    dbg!(t.1);
}
```

(playground link)

Tuples are limited

- You cannot use them in `for` loops.
- You cannot change their length.
- You cannot access elements by name.
- You cannot use array indexing syntax (e.g., `t[0]`).

4.4. Array iteration

Arrays are iterable.

```
1 fn main() {
2     let primes = [2, 3, 5, 7, 11, 13, 17, 19];
3     for prime in primes {
4         for i in 2..prime {
5             assert_ne!(prime % i, 0);
6         }
7     }
8 }
```

(playground link)

4.5. Patterns and destructuring

You can destructure tuples and arrays with **destructuring patterns**:

```
1  fn check_order(tuple: (i32, i32, i32)) -> bool {
2      let (left, middle, right) = tuple;
3      left < middle && middle < right
4  }
5
6  fn main() {
7      let tuple = (1, 5, 3);
8      println!(
9          "{tuple:?}: {}",
10         if check_order(tuple) { "ordered" } else { "unordered" }
11     );
12 }
```

(playground link)

Info

An irrefutable destructuring pattern is a pattern that **always matches**.

4.6. Exercise: nested arrays

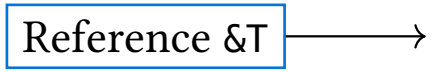
Implement a matrix transpose function that works on 3x3 matrices:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

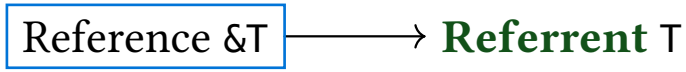
```
1 fn transpose(matrix: [[i32; 3]; 3]) -> [[i32; 3]; 3] {
2     todo!()
3 }                                                    (playground link)
```

Find the exercise here: `session-1/examples/s1e3-transpose.rs`.

5.1. Shared references



5.1. Shared references




```rust
fn main() {
    let a = 'A';
    let b = 'B';
    let mut r: &char = &a;
    dbg!(r);
}
```

(playground link)

5.1. Shared references

Reference &T ⟶ **Referrent** T

```
1 fn main() {
2     let a = 'A';
3     let b = 'B';
4     let mut r: &char = &a;
5     dbg!(r);
6 }
```

(playground link)

How is `r` displayed in the output?

5.1. Shared references

Reference `&T` ⟶ **Referrent** `T`

```
1 fn main() {
2     let a = 'A';
3     let b = 'B';
4     let mut r: &char = &a;
5     dbg!(r);
6 }
```

(playground link)

How is `r` displayed in the output?

It shows the value pointed to: `'A'`.

5.1. Shared references

Reference `&T` ⟶ **Referrent** `T`

```
1 fn main() {
2     let a = 'A';
3     let b = 'B';
4     let mut r: &char = &a;
5     dbg!(r);
6 }
```

(playground link)

How is `r` displayed in the output?

It shows the value pointed to: `'A'`.

How can we print the reference itself (the memory address)?

5.1. Shared references

Reference `&T` ⟶ **Referrent** `T`

```
1 fn main() {
2     let a = 'A';
3     let b = 'B';
4     let mut r: &char = &a;
5     dbg!(r);
6 }
```

(playground link)

How is `r` displayed in the output?

It shows the value pointed to: `'A'`.

How can we print the reference itself (the memory address)?

Use `println!("{r:p}")` or convert to raw pointer: `println!("{:p}", r as *const _)`

5.1. Shared references

Warning

Objects are shared immutably among different & through shared references.

```
1 fn main() {
2     let mut c = 'C';
3     let r: &char = &c;
4     // *r = 'D';
5     dbg!(r);
6 }
```

(playground link)

5.1. Shared references

Warning

Objects are shared immutably among different & through shared references.

```
1 fn main() {
2     let mut c = 'C';
3     let r: &char = &c;
4     // *r = 'D';
5     dbg!(r);
6 }
```

(playground link)

What happens if we uncomment the line that tries to modify *r?

5.1. Shared references

Warning

Objects are shared immutably among different & through shared references.

```
1 fn main() {
2     let mut c = 'C';
3     let r: &char = &c;
4     // *r = 'D';
5     dbg!(r);
6 }
```

(playground link)

What happens if we uncomment the line that tries to modify *r?

Objects cannot be modified through & references.

5.2. Dereferencing

References can be dereferenced automatically in certain cases.

In C++, you would dereference like this:

```cpp
char c = 'A';
char* r = &c;
std::cout << *r << std::endl;
```
(playground link)

In Rust, you can just write:

```rust
let c: char = 'A';
let r: &char = &c;
println!("{r}");
```
(playground link)

5.3. Difficult question

"What is the output of this Rust program?"

```rust
fn main() {
    let a;
    let a = a = true;
    print!("{}", std::mem::size_of_val(&a));
}
```

(playground link)

5.3. Difficult question

"What is the output of this Rust program?"

```
1 fn main() {
2     let a;
3     let a = a = true;
4     print!("{}", std::mem::size_of_val(&a));
5 }
```
(playground link)

The variable `a` is shadowed by a new variable `a` which could be renamed `b`:

```
1 let a;
2 let b = a = true;
3 print!("{}", mem::size_of_val(&b));
```
(playground link)

5.3. Difficult question

```
1  let a;
2  let b = a = true;
3  print!("{}", mem::size_of_val(&b));                    (playground link)
```

What is the output of this Rust program?

5.3. Difficult question

```
1  let a;
2  let b = a = true;
3  print!("{}", mem::size_of_val(&b));                    (playground link)
```

What is the output of this Rust program?

The value of an assignment is `()`.

We can rewrite the program as:

```
1  let a = true;
2  let b = ();
3  print!("{}", mem::size_of_val(&b));                    (playground link)
```

The `size_of_val` gives the size of `()` (the referent).

What is the size of `()`?

5.3. Difficult question

```
1  let a;
2  let b = a = true;
3  print!("{}", mem::size_of_val(&b));                    (playground link)
```

What is the output of this Rust program?

The value of an assignment is `()`.

We can rewrite the program as:

```
1  let a = true;
2  let b = ();
3  print!("{}", mem::size_of_val(&b));                    (playground link)
```

The `size_of_val` gives the size of `()` (the referent).

What is the size of `()`?

0 bytes.

5.4. Pointers

Every reference can be converted to a raw pointer, using a cast with `as`:

```
1 let x: i32 = 10;
2 let r: *const i32 = &x as *const i32;                    (playground link)
```

Info

Raw pointers are unsafe to dereference and require `unsafe` blocks.

```
1 unsafe {
2     println!("Value: {}", *r);
3 }                                                        (playground link)
```

Is every pointer a reference?

5.4. Pointers

Every reference can be converted to a raw pointer, using a cast with `as`:

```
1 let x: i32 = 10;
2 let r: *const i32 = &x as *const i32;                    (playground link)
```

Info

Raw pointers are unsafe to dereference and require `unsafe` blocks.

```
1 unsafe {
2     println!("Value: {}", *r);
3 }                                                        (playground link)
```

Is every pointer a reference?

No. References have extra compile-time checks to **prevent dangling** pointers.

Is every reference a pointer?

5.4. Pointers

Every reference can be converted to a raw pointer, using a cast with `as`:

```
1 let x: i32 = 10;
2 let r: *const i32 = &x as *const i32;                    (playground link)
```

Info

Raw pointers are unsafe to dereference and require `unsafe` blocks.

```
1 unsafe {
2     println!("Value: {}", *r);
3 }                                                        (playground link)
```

Is every pointer a reference?

No. References have extra compile-time checks to **prevent dangling** pointers.

Is every reference a pointer?

Yes.

5.5. Exclusive references

Also called mutable references.

Info

Exclusive references (`&mut T`) allow mutation of the referenced object.

```
1 fn main() {
2     let mut point = (1, 2);
3     let x_coord = &mut point.0;
4     // let y_coord = &mut point.1;
5     *x_coord = 20;
6     println!("point: {point:?}");
7 }
```

(playground link)

What happens if we uncomment?

5.5. Exclusive references

Also called mutable references.

Info

Exclusive references (`&mut T`) allow mutation of the referenced object.

```
1 fn main() {
2     let mut point = (1, 2);
3     let x_coord = &mut point.0;
4     // let y_coord = &mut point.1;
5     *x_coord = 20;
6     println!("point: {point:?}");
7 }
```

(playground link)

What happens if we uncomment?

Illegal: cannot borrow `point` as mutable more than once at a time.

Warning

One exclusive reference can exist at a time!

5.6. Exclusive vs. shared

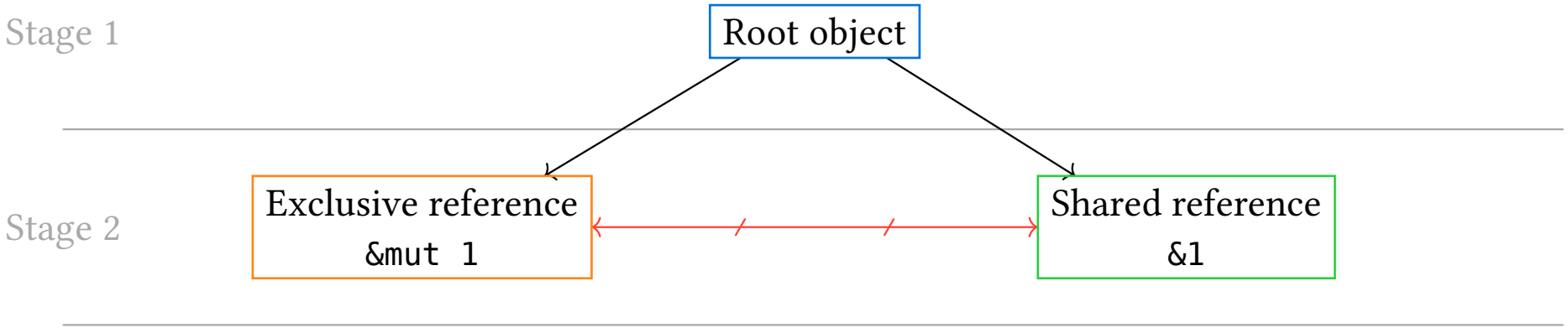
Exclusive references cannot coexist with shared references.

Stage 1

Root object

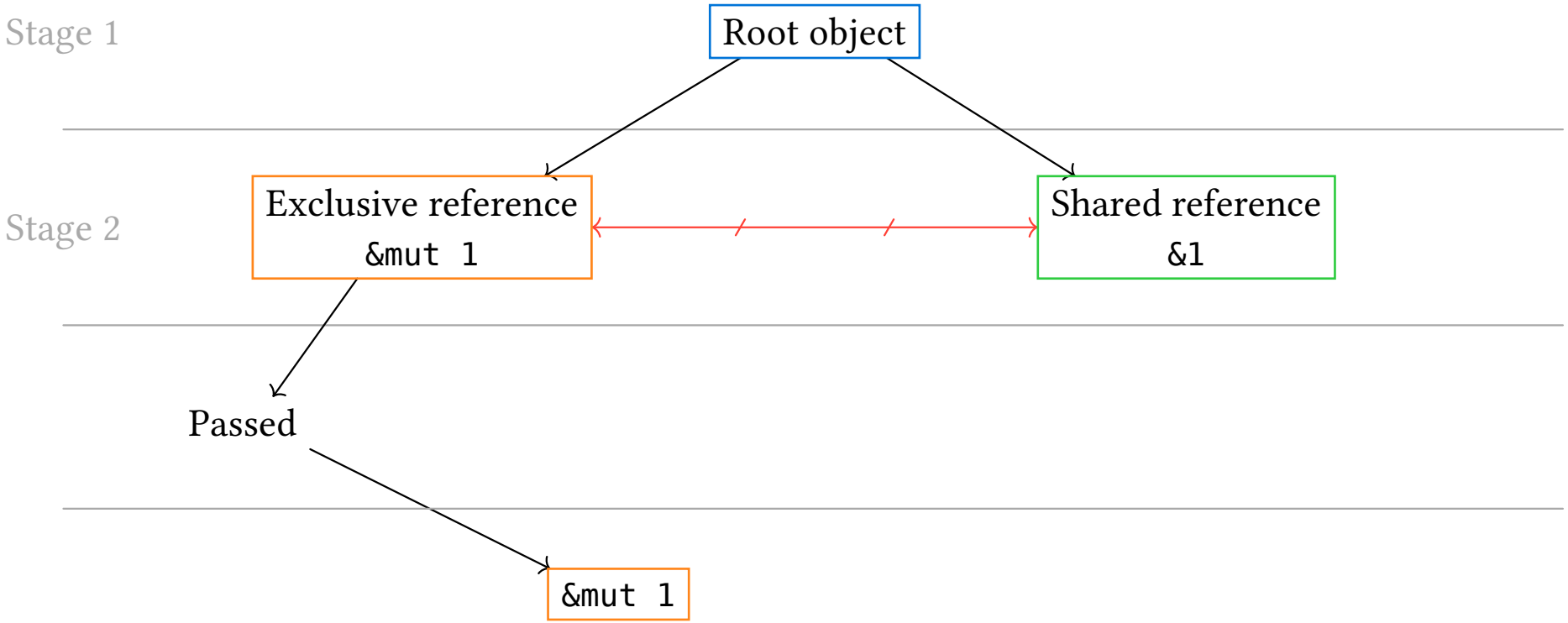
5.6. Exclusive vs. shared

Exclusive references cannot coexist with shared references.



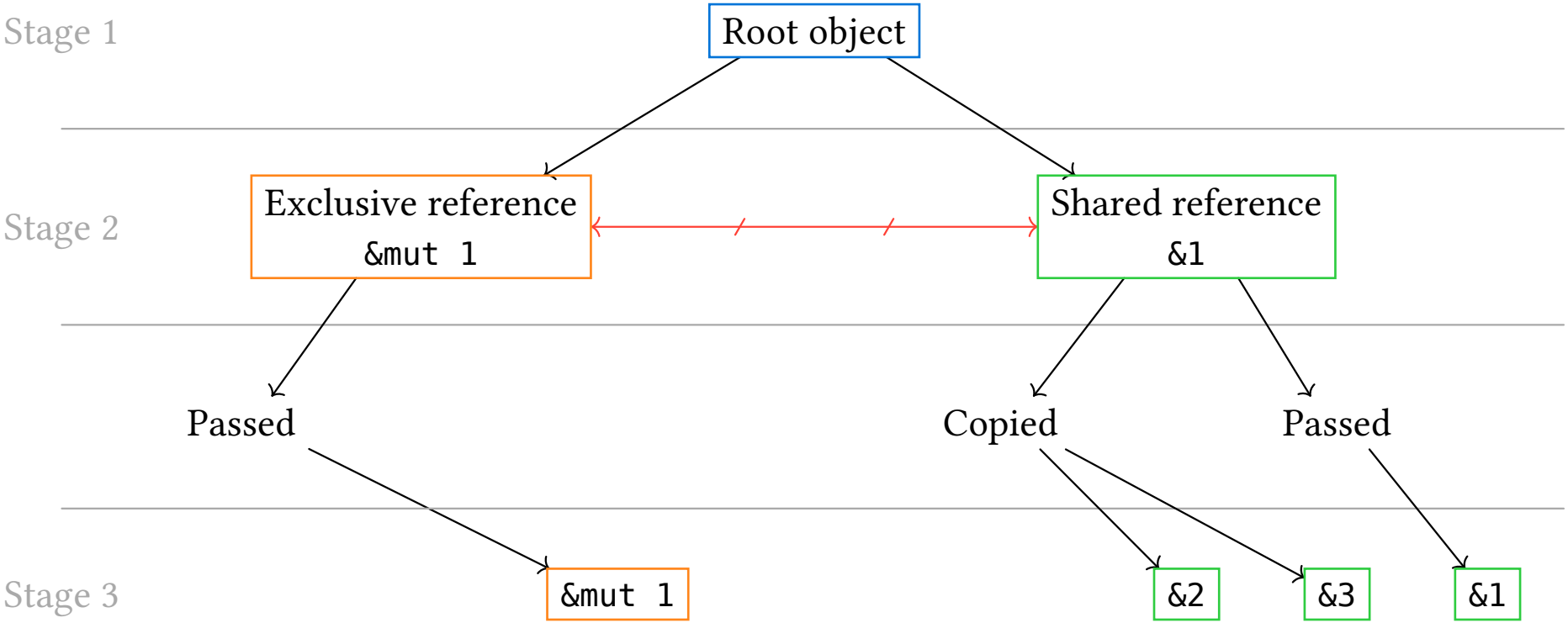
5.6. Exclusive vs. shared

Exclusive references cannot coexist with shared references.



5.6. Exclusive vs. shared

Exclusive references cannot coexist with shared references.



5.7. Kinds of `mut`

```
1  let mut x_coord: &i32;
2  let x_coord: &mut i32;                                    (playground link)
```

What is the difference between these statements?

5.7. Kinds of `mut`

```
1 let mut x_coord: &i32;
2 let x_coord: &mut i32;                    (playground link)
```

What is the difference between these statements?

- `let mut x_coord: &i32;`: An immutable reference that can be reassigned to point to different `i32` values.
- `let x_coord: &mut i32;`: An exclusive reference that allows modifying the `i32` value it points to.

Info

Mutable references don't have to be assigned to `mut` variables to be able to mutate the referenced value.

5.7. Kinds of `mut`

There used to be discussion about having two different names for what is now both called `mut`:

5.7. Kinds of `mut`

There used to be discussion about having two different names for what is now both called `mut`:

Re-assignment `mut`

```
1  let mut a = 1;
2  a = 2; // You can re-assign the
   variable
3
4  impl {
5    fn some_method(mut self) {
6      // You can re-assign to self.
7      self = some_thing(); // not very
       useful
8      self.field = 2; // may be useful
9    }
10 }                          (playground link)
```

5.7. Kinds of `mut`

There used to be discussion about having two different names for what is now both called `mut`:

Re-assignment `mut`

```
1  let mut a = 1;
2  a = 2; // You can re-assign the
   variable
3
4  impl {
5    fn some_method(mut self) {
6      // You can re-assign to self.
7      self = some_thing(); // not very
       useful
8      self.field = 2; // may be useful
9    }
10 }                          (playground link)
```

Pointer `mut`

```
1  let mut a = 1;
2  let b = &mut a; // Need assignment `mut`
   on `a`
3  *b = 2; // You can assign through the
   pointer                    (playground link)
```

Both are related to mutability, but are completely separate.

5.8. Slices

Info

A slice is a **view** into a contiguous sequence. (Like a horizontal continuous window)

```rust
fn main() {
    let a: [i32; 6] = [10, 20, 30, 40, 50, 60];
    println!("a: {a:?}");
    let s: &[i32] = &a[2..4];
    println!("s: {s:?}");
}
```

(playground link)

5.8. Slices

Info

A slice is a **view** into a contiguous sequence. (Like a horizontal continuous window)

```
1 fn main() {
2     let a: [i32; 6] = [10, 20, 30, 40, 50, 60];
3     println!("a: {a:?}");
4     let s: &[i32] = &a[2..4];
5     println!("s: {s:?}");
6 }
```

(playground link)

Can we drop the end index in the slice?

5.8. Slices

Info

A slice is a **view** into a contiguous sequence. (Like a horizontal continuous window)

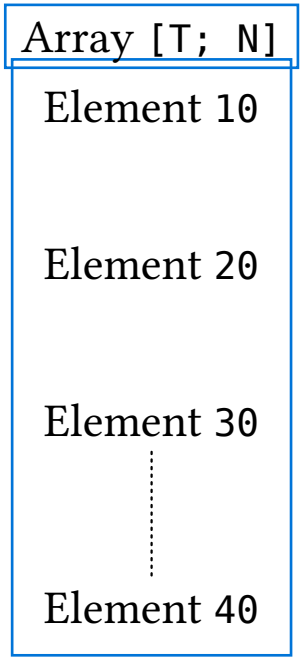
```
1 fn main() {
2     let a: [i32; 6] = [10, 20, 30, 40, 50, 60];
3     println!("a: {a:?}");
4     let s: &[i32] = &a[2..4];
5     println!("s: {s:?}");
6 }
```

(playground link)

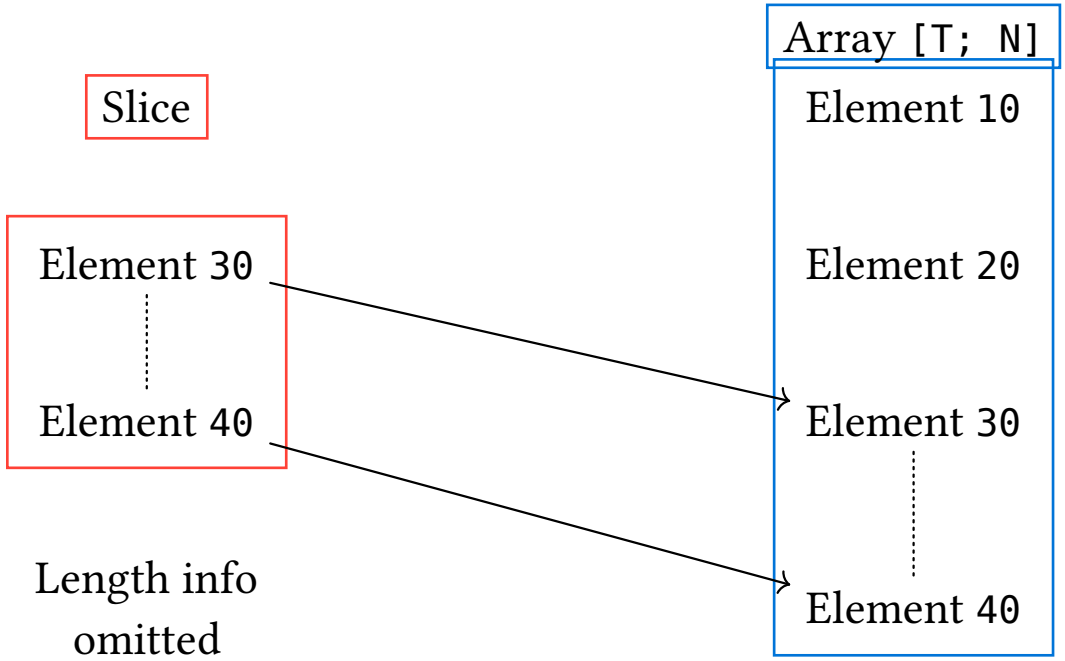
Can we drop the end index in the slice?

Yes, it defaults to the length of the array.

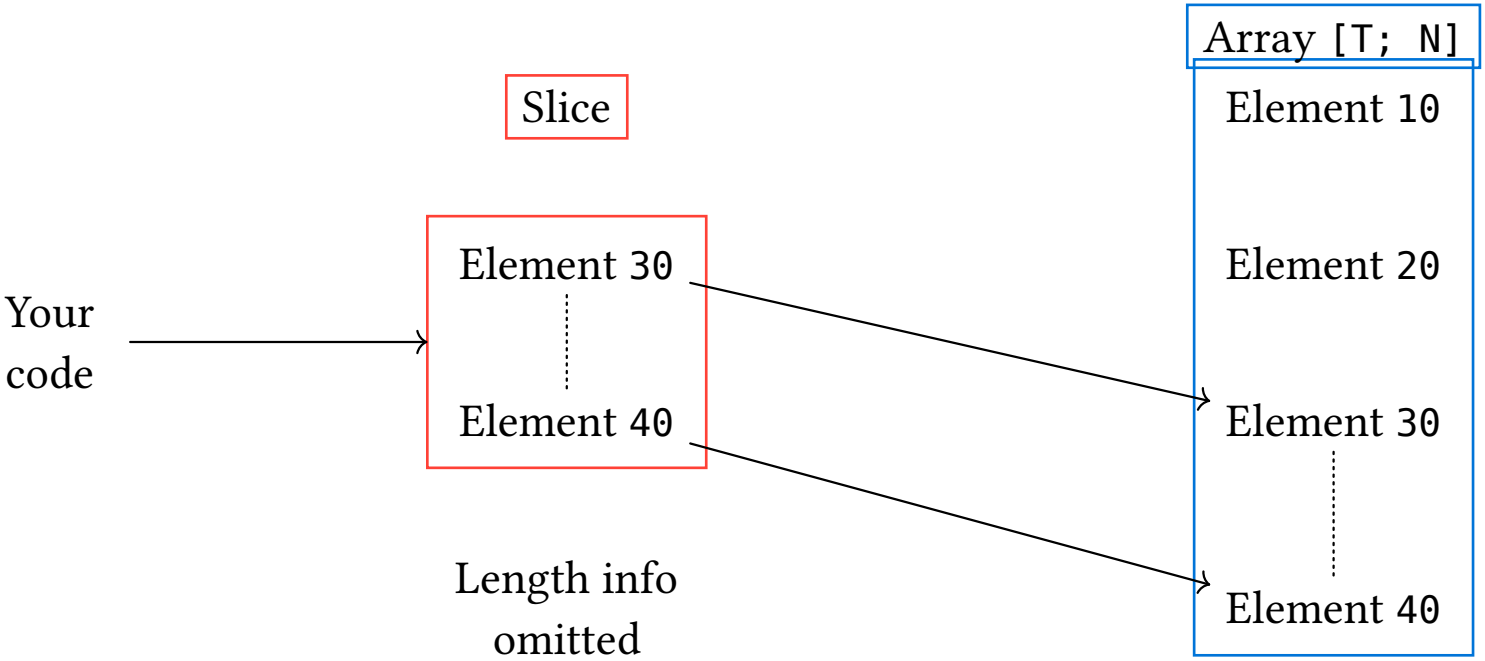
5.8. Slices



5.8. Slices

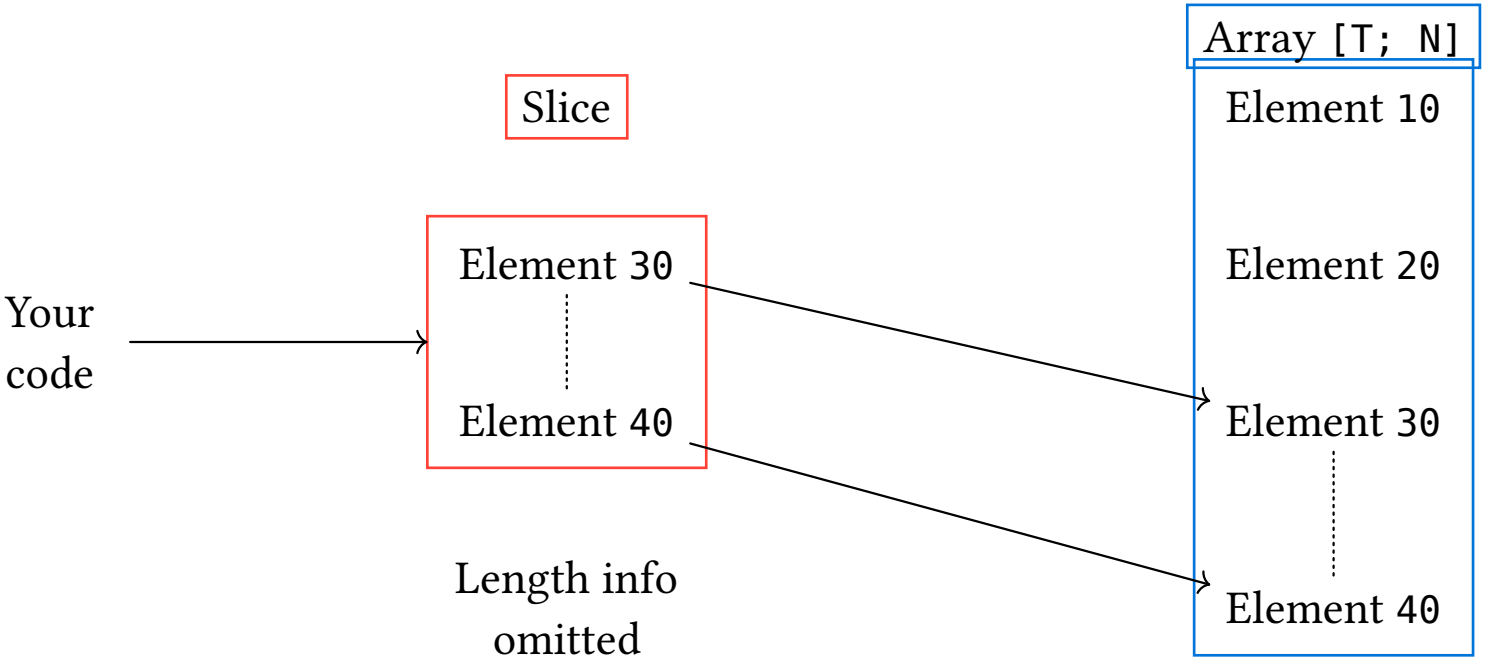

Array [T; N]
Slice
Element 10
Element 30
Element 20
Element 40
Element 30
Element 40
Length info
omitted

5.8. Slices



What happens when you have the slice, but destroy the original array?

5.8. Slices



What happens when you have the slice, but destroy the original array?

Not possible in normal Rust. (See later about lifetimes)

5.8. Slices

More limitations of slices:

- You cannot append elements to a slice.
- You cannot grow a slice.
- You cannot shrink a slice.

However, slices have unique functionality defined.

Warning

Data carriers like arrays may refer you to the methods of the associated slices.

(see next slides about strings)

5.9. Strings

```
1  fn main() {
2      let s1: &str = "World";
3      println!("s1: {s1}");
4
5      let mut s2: String = String::from("Hello
       ");
6      println!("s2: {s2}");
7
8      s2.push_str(s1);
9      println!("s2: {s2}");
10
11     let s3: &str = &s2[2..9];
12     println!("s3: {s3}");
13 }                              (playground link)
```

Two new types:

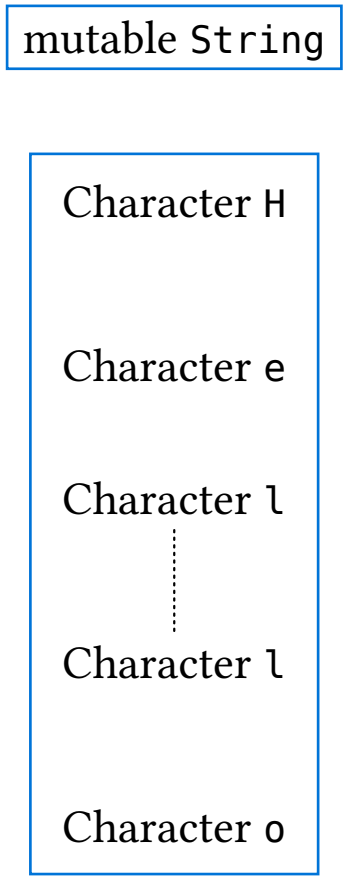
- A `&str` is an immutable slice of a `String`.
- A `String` is mutable vector of UTF-8 characters.

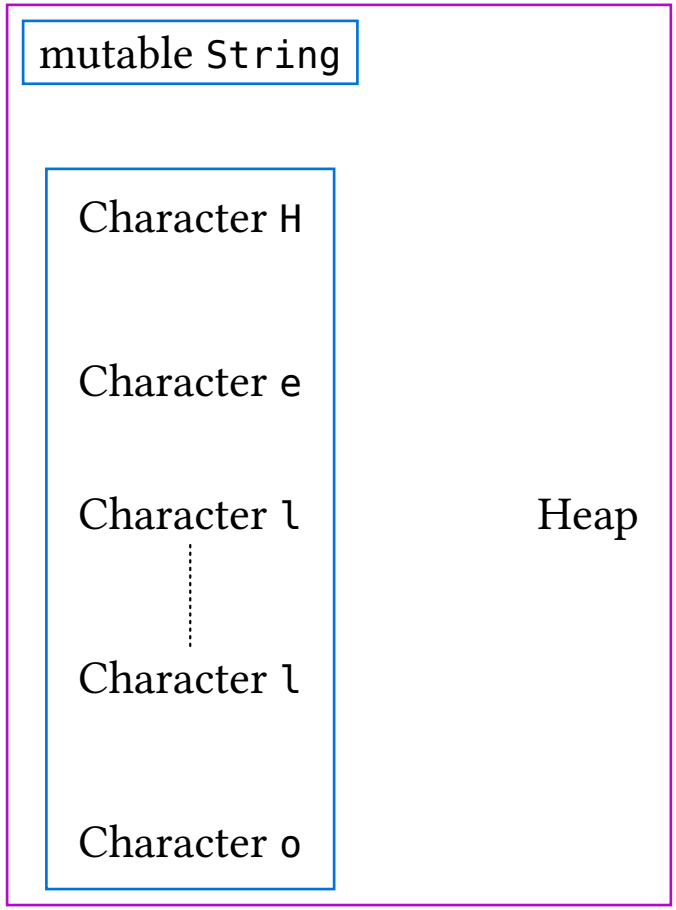
For the C++ programmers:

- `&str` is like `std::string_view`
- `String` is like `std::string`

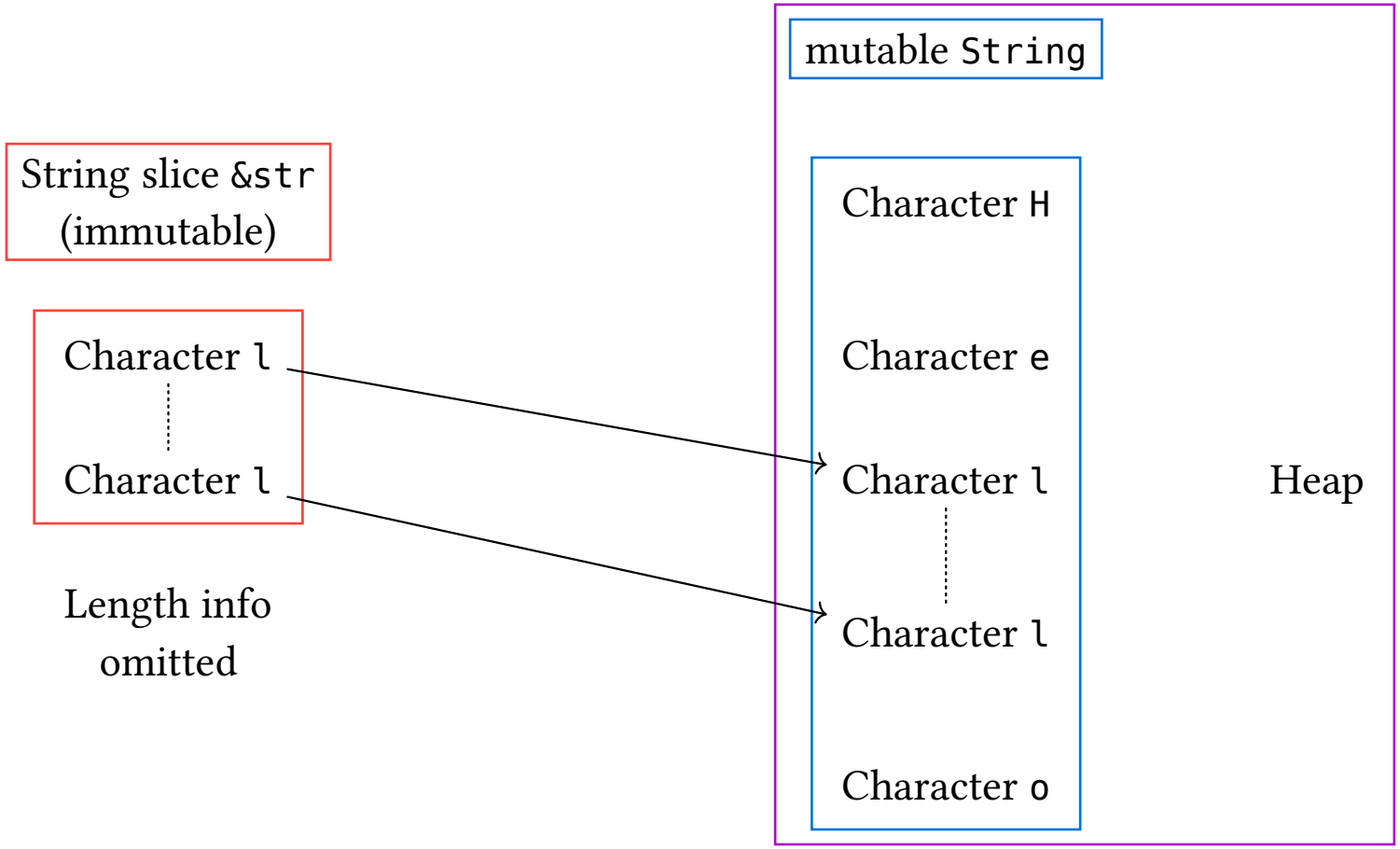
Usage:

- `String::from(&str)` creates a `String` from a `&str`.
- `&String` coerces to `&str` automatically.
- String interpolation with formatting macros: `println!`, `format!`, `dbg!`, etc.


mutable String
Character H
Character e
Character l
Character l
Character o


mutable String
Character H
Character e
Character l
Character l
Character o
Heap

5.9. Strings


mutable String
String slice &str
(immutable)
Character H
Character e
Character l
Character l
Character o
Character l
Character l
Heap
Length info
omitted

5.9. Strings

Is it possible to have a mutable string slice (`&mut str`)?

5.9. Strings

Is it possible to have a mutable string slice (`&mut str`)?

No.

5.9. Strings

Is it possible to have a mutable string slice (`&mut str`)?

No.

Why can't we have `&mut str`?

5.9. Strings

Is it possible to have a mutable string slice (`&mut str`)?

No.

Why can't we have `&mut str`?

Because strings are UTF-8 encoded and modifying individual bytes could create invalid UTF-8. Use `String` for mutation or `&mut [u8]` for byte manipulation.

5.9. Strings

Is it possible to have a mutable string slice (`&mut str`)?

No.

Why can't we have `&mut str`?

Because strings are UTF-8 encoded and modifying individual bytes could create invalid UTF-8. Use `String` for mutation or `&mut [u8]` for byte manipulation.

When should you use `String` vs `&str` in function parameters?

5.9. Strings

Is it possible to have a mutable string slice (`&mut str`)?

No.

Why can't we have `&mut str`?

Because strings are UTF-8 encoded and modifying individual bytes could create invalid UTF-8. Use `String` for mutation or `&mut [u8]` for byte manipulation.

When should you use `String` vs `&str` in function parameters?

Prefer `&str` for parameters (more flexible - accepts both `String` and `&str`). Use `String` when you need ownership or will modify it.

5.9. Strings

What if you **don't want UTF-8** encoding?,

5.9. Strings

What if you **don't want UTF-8** encoding?,

Youuse byte slices of type `&[u8]`:

```
println!("{:?}", b"abc");
println!("{:?}", &[97, 98, 99]);
```
(playground link)

Are you going "inception", use '\' escapes (or a special Rust feature '#'):

```
println!(r#"<a href="link.html">link</a>"#);
println!("<a href=\"link.html\">link</a>");
```
(playground link)

5.10. Reference validity

```
1 fn main() {
2     let x_ref = {
3         let x = 10;
4         &x
5     };
6     dbg!(x_ref);
7 }
```

(playground link)

5.10. Reference validity

```
1 fn main() {
2     let x_ref = {
3         let x = 10;
4         &x
5     };
6     dbg!(x_ref);
7 }
```

(playground link)

References in Rust are always safe to use, why?

5.10. Reference validity

```
1 fn main() {
2     let x_ref = {
3         let x = 10;
4         &x
5     };
6     dbg!(x_ref);
7 }
```

(playground link)

References in Rust are always safe to use, why?

- references can never be null
- references can't outlive the data they point to

(See next lectures)

5.11. Exercise: vectors

Write some utility functions for calculating the magnitude and normalizing 3D vectors.

Find the statement of the exercise here: `session-1/examples/s1e4-vectors.rs`.

6.1. Named structs

Exactly the same as in C/C++:

```
1 struct Person {
2     name: String,
3     age: u8,
4 }
5
6 fn describe(person: &Person) {
7     println!("{} is {} years old", person.name, person.age);
8 }
```

(playground link)

6.1. Named structs

6.1.1. Usage

```
1   struct Person {
2       name: String,
3       age: u8,
4   }
5
6   fn main() {
7       let name = String::from("Peter");
8       let mut peter = Person {
9           name,
10          age: 27,
11      };
12      describe(&peter);
13      peter.age = 28;
14  }
```

(playground link)

Warning

Struct fields do not support default values.

Use the `Default` trait to implement default values (see later).

6.1. Named `structs`

6.1.2. Inheritance

How does Rust handle `struct` inheritance (like class inheritance in other languages?

6.1. Named `structs`

6.1.2. Inheritance

How does Rust handle `struct` inheritance (like class inheritance in other languages?

Rust explicitly forbids `struct` inheritance.

Warning

Inheritance in Rust is replaced by trait composition.

Info

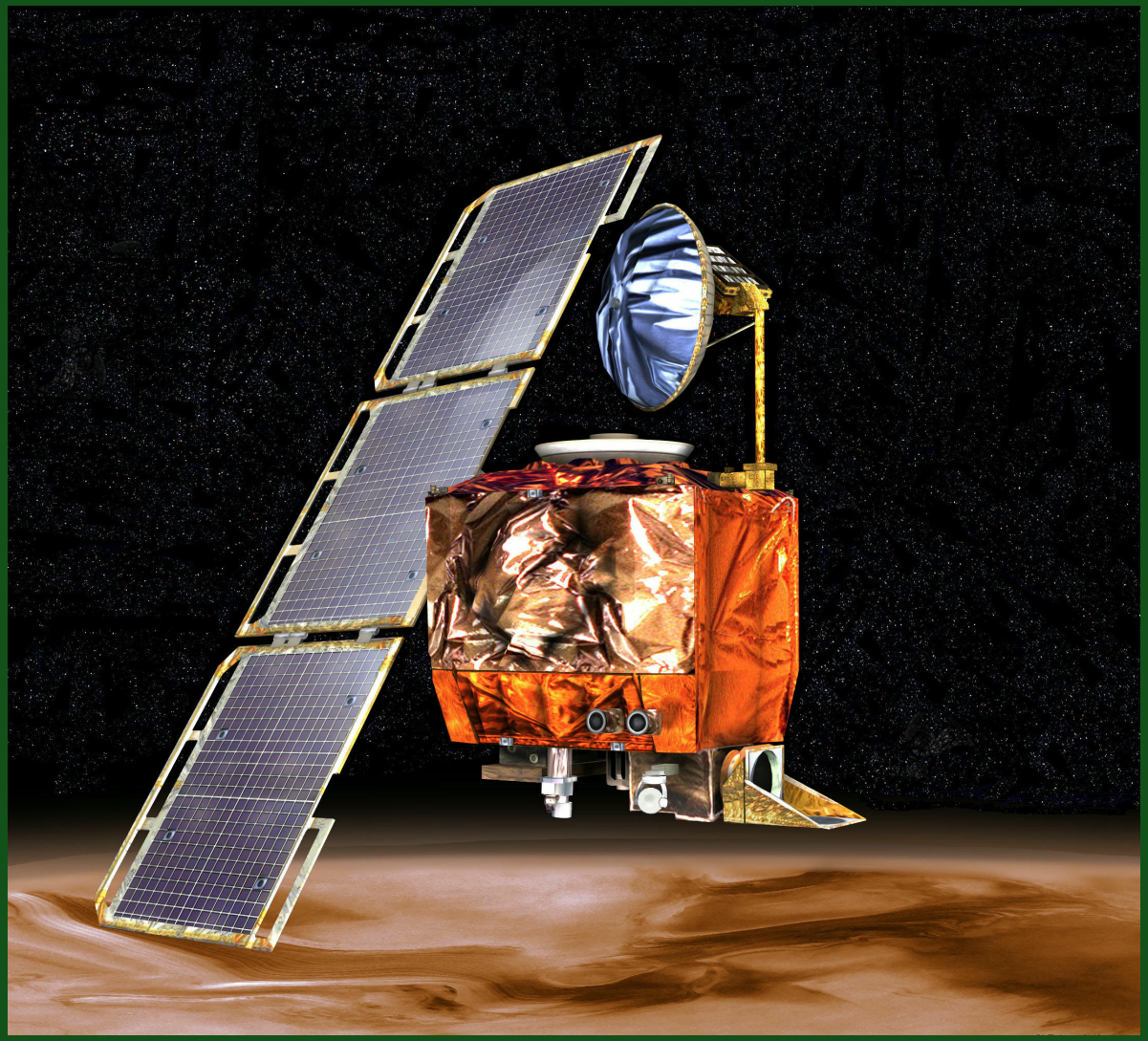
Only abstract interfaces can be inherited, concrete types such as `structs` cannot.

6.2. Tuple structs

If the field names are unimportant, you can use a tuple struct:

```
1   struct Point(i32, i32);
2
3 fn main() {
4     let p = Point(17, 23);
5     println!("({}, {})", p.0, p.1);
6 }
```

(playground link)



6.2. Tuple `structs`

What is the name of this satellite?

6.2. Tuple `structs`

What is the name of this satellite?

Mars Climate Orbiter

Why did it fail?

6.2. Tuple structs

What is the name of this satellite?

Mars Climate Orbiter

Why did it fail?

A unit conversion error: one team used imperial units (pounds of force), the other metric (Newtons).

6.2. Tuple `structs`

6.2.1. Usage

This is often used for single-field wrappers (called newtypes):

6.2. Tuple structs

```rust
struct PoundsOfForce(f64);
struct Newtons(f64);

fn compute_thruster_force() -> PoundsOfForce {
    todo!("Ask a rocket scientist at NASA")
}

fn set_thruster_force(force: Newtons) {
    // ...
}

fn main() {
    let force = compute_thruster_force();
    set_thruster_force(force);
}
```

(playground link)

6.2. Tuple `structs`

6.2.2. Remarks

- Great way to encode additional information about the value in a primitive type
- Value passed some validation when it was created, so you no longer have to validate it again at every use

How do you prevent invalidation of the value after validation during creation?

6.2. Tuple structs

6.2.2. Remarks

- Great way to encode additional information about the value in a primitive type
- Value passed some validation when it was created, so you no longer have to validate it again at every use

How do you prevent invalidation of the value after validation during creation?

Keep the inner field private by adding `pub` only to the struct, not to its fields.

6.3. Enums

The enum keyword allows the creation of a type which has a few different variants:

```
#[derive(Debug)]
enum Direction {
    Left,
    Right,
}
```
(playground link)

```
#[derive(Debug)]
enum PlayerMove {
    Pass,
    Run(Direction),
    Teleport { x: u32, y: u32 },
}
```
(playground link)

```
fn main() {
     let dir = Direction::Left;
     let player_move: PlayerMove = PlayerMove::Run(dir);
     println!("On this turn: {player_move:?}");
 }
```
(playground link)

6.3. Enums

6.3.1. Storage of `enum`

Info

An `enum` occupies enough space to store **its largest variant** plus some extra space to store which variant it currently is.

```
1    #[repr(u32)]
2  enum Bar {
3      A, // 0
4      B = 10000,
5      C, // 10001
6  }
7
8  fn main() {
9      println!("A: {}", Bar::A as u32);
10     println!("B: {}", Bar::B as u32);
11     println!("C: {}", Bar::C as u32);
12 }
```

(playground link)

6.3. Enums

Without repr, the discriminant type takes 2 bytes, because 10001 fits 2 bytes.

6.3. Enums

Who is Tony Hoare and what was his billion dollar mistake?

6.3. Enums

Who is Tony Hoare and what was his billion dollar mistake?

Tony Hoare is a British computer scientist who invented the null reference in 1965 while designing the ALGOL W programming language.

6.3. Enums

6.3.2. Option enum

The simplest example of an enum is the `Option<T>` type, which represents either a value of type `T` or no value at all:

```
1 enum Option<T> {
2     None,
3     Some(T),
4 }                                                    (playground link)
```

How much space does `Option<&u8>` occupy?

6.3. Enums

6.3.2. Option enum

The simplest example of an enum is the `Option<T>` type, which represents either a value of type `T` or no value at all:

```
1 enum Option<T> {
2     None,
3     Some(T),
4 }                                                    (playground link)
```

How much space does `Option<&u8>` occupy?

The same as a single reference (8 bytes on 64-bit), because `None` is represented as a null pointer, so no extra discriminant space is needed.

6.3.2. Option enum

The simplest example of an enum is the `Option<T>` type, which represents either a value of type `T` or no value at all:

```
1 enum Option<T> {
2     None,
3     Some(T),
4 }
```
(playground link)

How much space does `Option<&u8>` occupy?

The same as a single reference (8 bytes on 64-bit), because `None` is represented as a null pointer, so no extra discriminant space is needed.

Info

For the functional programmers: `Option` is also a **monad**, but in Rust monadic `do` notation is replaced by the short-circuiting `?` operator.

6.3.3. The actual machine representation

We'll use `transmute` to peek at the internal bit representation:

```
1 use std::mem::transmute;
2 macro_rules! dbg_bits {
3     ($e:expr, $bit_type:ty) => {
4         println!("- {}: {:#x}", stringify!($e), transmute::<_, $bit_type>($e));
5     };
6 }
```

(playground link)

For every expression, what is its bit representation (as a hexadecimal number)?

6.3.3. The actual machine representation

We'll use `transmute` to peek at the internal bit representation:

```
1 use std::mem::transmute;
2 macro_rules! dbg_bits {
3     ($e:expr, $bit_type:ty) => {
4         println!("- {}: {:#x}", stringify!($e), transmute::<_, $bit_type>($e));
5     };
6 }
```

(playground link)

For every expression, what is its bit representation (as a hexadecimal number)?

Warning

`transmute` is unsafe and reinterprets memory. The compiler provides no guarantees about enum representations.

6.3. Enums

6.3.4. Representation of `bool` and `Option<bool>`

```rust
unsafe {
    println!("bool:");
    dbg_bits!(false, u8);
    dbg_bits!(true, u8);

    println!("Option<bool>:");
    dbg_bits!(None::<bool>, u8);
    dbg_bits!(Some(false), u8);
    dbg_bits!(Some(true), u8);
}


                                (playground link)
```

Expected output:

```
bool:
- false: 0x0
- true: 0x1
Option<bool>:
- None::<bool>: 0x2
- Some(false): 0x0
- Some(true): 0x1        (playground link)
```

Info

`Option<bool>` uses `0x2` for `None`, allowing all three states in 1 byte!

6.3.5. Representation of `Option<Option<bool>>`

```
1  unsafe {
2      println!("Option<Option<bool>>:");
3      dbg_bits!(Some(Some(false)), u8);
4      dbg_bits!(Some(Some(true)), u8);
5      dbg_bits!(Some(None::<bool>), u8);
6      dbg_bits!(None::<Option<bool>>,
       u8);
7  }
8
9
10                                (playground link)
```

Expected output:

```
1  Option<Option<bool>>:
2  - Some(Some(false)): 0x0
3  - Some(Some(true)): 0x1
4  - Some(None::<bool>): 0x2
5  - None::<Option<bool>>: 0x3  (playground link)
```

Info

Four distinct states, still fitting in just 1 byte!

6.3. Enums

6.3.6. `Option<&i32>` and null pointer optimization

```
unsafe {
    println!("Option<&i32>:");
    dbg_bits!(None::<&i32>, usize);
    dbg_bits!(Some(&0i32), usize);
}


                                (playground link)
```

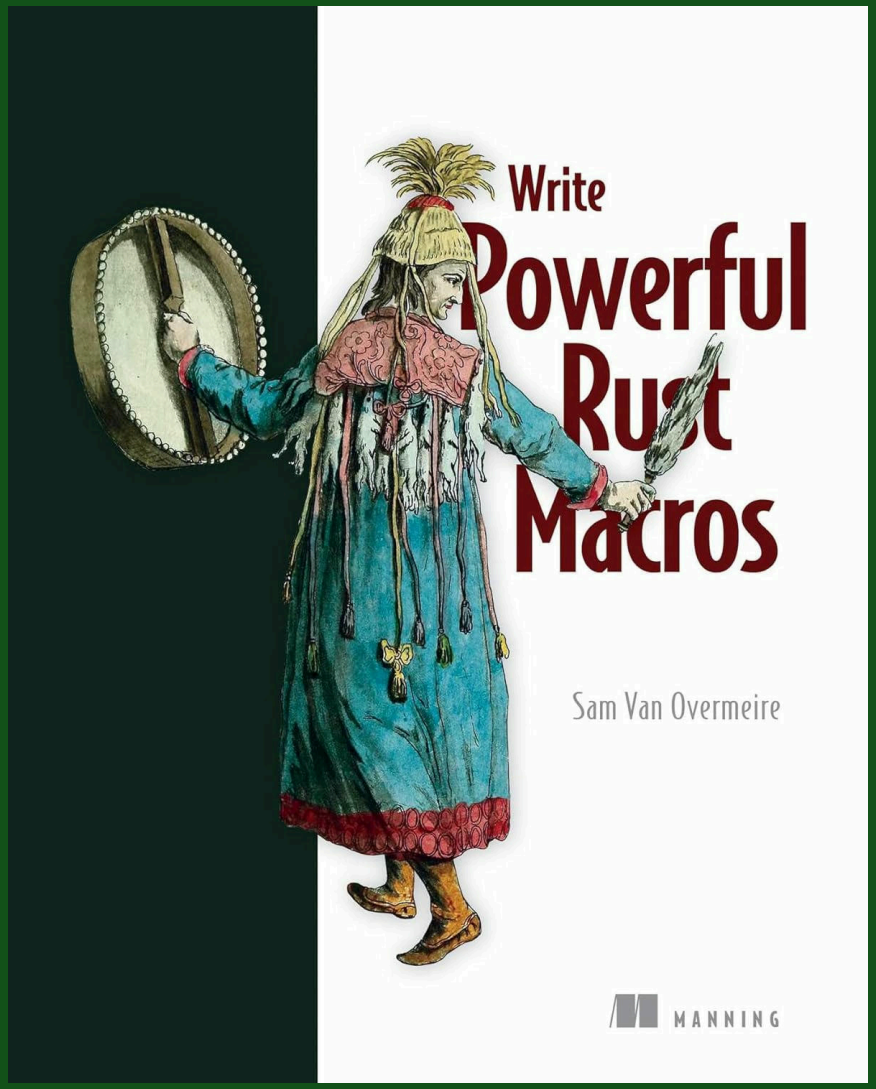
Expected output (on 64-bit):

```
Option<&i32>:
- None::<&i32>: 0x0
- Some(&0i32): 0x7ffc8e4a1b2c
```

(playground link)

Info

`None` is represented as null pointer (`0x0`), while `Some` contains the actual memory address. No extra space needed for the discriminant!


Write
Powerful
Rust
Macros
Sam Van Overmeire
MANNING

6.4. Compile-time const

```rust
const fn is_prime(n: u32) -> bool {
    if n < 2 { return false; }
    let mut i = 2;
    while i * i <= n {
        if n % i == 0 { return false; }
        i += 1;
    }
    true
}

fn main() {
    println!("First 25 primes: {:?}",
             PRIMES_BELOW_100);
}
```

(playground link)

```rust
const PRIMES_BELOW_100: [u32; 25] = {
    let mut primes = [0; 25];
    let mut count = 0;
    let mut n = 2;
    while n < 100 {
        if is_prime(n) {
            primes[count] = n;
            count += 1;
        }
        n += 1;
    }
    primes
};
```

(playground link)

Info

The prime sieve runs at **compile time**, resulting in zero runtime overhead!

6.4. Compile-time `const`

Info

Similar to C++'s `constexpr`, Zig's `comptime`

6.4. Compile-time `const`

Info

Similar to C++'s `constexpr`, Zig's `comptime`

What are the limitations of `const fn`?

6.4. Compile-time `const`

Info

Similar to C++'s `constexpr`, Zig's `comptime`

What are the limitations of `const fn`?

- Cannot allocate heap memory
- Cannot call non-`const` functions
- Cannot use floating-point operations (being relaxed in newer Rust versions)
- Limited to a subset of Rust operations

6.4. Compile-time `const`

Info

Similar to C++'s `constexpr`, Zig's `comptime`

What are the limitations of `const fn`?

- Cannot allocate heap memory
- Cannot call non-`const` functions
- Cannot use floating-point operations (being relaxed in newer Rust versions)
- Limited to a subset of Rust operations

Why these restrictions?

6.4. Compile-time `const`

Info

Similar to C++'s `constexpr`, Zig's `comptime`

What are the limitations of `const fn`?

- Cannot allocate heap memory
- Cannot call non-`const` functions
- Cannot use floating-point operations (being relaxed in newer Rust versions)
- Limited to a subset of Rust operations

Why these restrictions?

To guarantee deterministic, side-effect-free compile-time evaluation with predictable performance.

6.5. Static

Static variables will live during the whole execution of the program, and therefore will not move:

```
1 static BANNER: &str = "Welcome to RustOS 3.14";
2
3 fn main() {
4     println!("{BANNER}");
5 }
```

(playground link)

Info

Accessible from any thread.

Warning

Not inlined upon use and have an actual associated memory location.

Why is this often used in embedded Rust?

6.5. Static

Static variables will live during the whole execution of the program, and therefore will not move:

```
1 static BANNER: &str = "Welcome to RustOS 3.14";
2
3 fn main() {
4     println!("{BANNER}");
5 }
```

(playground link)

Info

Accessible from any thread.

Warning

Not inlined upon use and have an actual associated memory location.

Why is this often used in embedded Rust?

Stable addresses are often required. Memory shared between cores. No heap allocation is available.

6.5. Static

6.5.1. `static OnceLock` pattern

When a `static` needs to be initialised at runtime (just once) and is read-only, use `OnceLock`:

```rust
use std::sync::OnceLock;

static CONFIG: OnceLock<Config> = OnceLock::new();

fn main() {
    CONFIG.get_or_init(|| {
        Config::load("config.toml")
    });
}
```

(playground link)

6.5.2. Plant pot workshop

A few months ago, at this location, we built an embedded Rust plant pot monitor together

Instructions available at https://github.com/sysghent/plant-pot.

```
1   static HUMIDITY_PUBSUB_CHANNEL:
2     PubSubChannel<CriticalSectionRawMutex, f32, 1, 3, 1> =
3       PubSubChannel::new();
4
5   #[main]
6   async fn main(spawner: Spawner) -> ! {
7       let p = embassy_rp::init(config::Config::default());
8       let adc_component = Adc::new(p.ADC, Irqs, Config::default());
9       // ...
10    }                                                    (playground link)
```

6.5.2. Plant pot workshop

A few months ago, at this location, we built an embedded Rust plant pot monitor together

Instructions available at https://github.com/sysghent/plant-pot.

```
1   static HUMIDITY_PUBSUB_CHANNEL:
2     PubSubChannel<CriticalSectionRawMutex, f32, 1, 3, 1> =
3       PubSubChannel::new();
4
5   #[main]
6   async fn main(spawner: Spawner) -> ! {
7       let p = embassy_rp::init(config::Config::default());
8       let adc_component = Adc::new(p.ADC, Irqs, Config::default());
9       // ...
10    }                                                    (playground link)
```

In embedded Rust, you either use the `OnceLock` or `const` functions to initialise `static` variables.

Where is the `const fn` function in this fragment and how to find out?

6.5.2. Plant pot workshop

A few months ago, at this location, we built an embedded Rust plant pot monitor together

Instructions available at https://github.com/sysghent/plant-pot.

```
1   static HUMIDITY_PUBSUB_CHANNEL:
2     PubSubChannel<CriticalSectionRawMutex, f32, 1, 3, 1> =
3       PubSubChannel::new();
4
5   #[main]
6   async fn main(spawner: Spawner) -> ! {
7       let p = embassy_rp::init(config::Config::default());
8       let adc_component = Adc::new(p.ADC, Irqs, Config::default());
9       // ...
10    }
```

(playground link)

In embedded Rust, you either use the `OnceLock` or `const` functions to initialise `static` variables.

Where is the `const fn` function in this fragment and how to find out?

(In this example `PubSubChannel::new()` is a `const fn`.)

6.6. Summary: World map

Compile-time world

6.6. Summary: World map

Compile-time world

`const`

variables

6.6. Summary: World map

Compile-time world

`const` variables

`const fn` functions

6.6. Summary: World map

Compile-time world

`const` variables

`const fn` functions

`const` generics

6.6. Summary: World map

Compile-time world	
`const` variables	
`const fn` functions	
`const` generics	

6.6. Summary: World map

Compile-time world	Runtime world
`const` variables	
`const fn` functions	
`const` generics	

6.6. Summary: World map

Compile-time world	Wild West: embedded	Runtime world
`const` variables		
`const fn` functions		
`const` generics		

6.6. Summary: World map

Compile-time world	Wild West: embedded	Runtime world
`const` variables	`OnceLock` `LazyLock` “init on use”	
`const fn` functions		
`const` generics		

6.6. Summary: World map

Compile-time world	Wild West: embedded	Runtime world
`const` variables	`OnceLock` `LazyLock` “init on use”	
`const fn` functions	`unsafe` code	
`const` generics		

6.6. Summary: World map

Compile-time world	Runtime world: Wild West: embedded	Runtime world: Safe Rust

`const` variables	`OnceLock`
	`LazyLock`
	“init on use”	

`const fn` functions	`unsafe` code	
`const` generics		

6.6. Summary: World map

Compile-time world	Wild West: embedded	**Runtime world**	Safe Rust
`const` variables	`OnceLock` `LazyLock` “init on use”		Heap allocation
`const fn` functions	`unsafe` code		
`const` generics			

6.6. Summary: World map

Compile-time world	Wild West: embedded	**Runtime world**	Safe Rust
`const` variables	`OnceLock` `LazyLock` “init on use”		Heap allocation
`const fn` functions	`unsafe` code		Normal functions
`const` generics			

6.6. Summary: World map

Compile-time world	Wild West: embedded	**Runtime world**	Safe Rust	
`const` variables	`OnceLock` `LazyLock` “init on use”		Heap allocation	Mutable variables
`const fn` functions	`unsafe` code		Normal functions	
`const` generics				

6.7. Exercise: Elevator Events

6.7.1. Goal

Create a data structure to represent an event in an elevator control system.

Complete the code in `session-1/examples/s1e3-elevator.rs`.

7.1. Irrefutable patterns

Info

An **irrefutable pattern** is a pattern that always matches.

```
1   fn takes_tuple(tuple: (char, i32, bool)) {
2       let a = tuple.0;
3       let b = tuple.1;
4       let c = tuple.2;
5
6       // This does the same thing as above.
7       let (a, b, c) = tuple;
8
9       // Ignore the first element, only bind the
        second and third.
10      let (_, b, c) = tuple;
11
12      // Ignore everything but the last element.
13      let (.., c) = tuple;
14  }                                   (playground link)
```

7.1. Irrefutable patterns

Info

An **irrefutable pattern** is a pattern that always matches.

```
1   fn takes_tuple(tuple: (char, i32, bool)) {
2       let a = tuple.0;
3       let b = tuple.1;
4       let c = tuple.2;
5
6       // This does the same thing as above.
7       let (a, b, c) = tuple;
8
9       // Ignore the first element, only bind the
        second and third.
10      let (_, b, c) = tuple;
11
12      // Ignore everything but the last element.
13      let (.., c) = tuple;
14  }                                   (playground link)
```

What happens when adding or removing an element to the tuple and look at the resulting compiler errors

Info

An **irrefutable pattern** is a pattern that always matches.

```
1   fn takes_tuple(tuple: (char, i32, bool)) {
2       let a = tuple.0;
3       let b = tuple.1;
4       let c = tuple.2;
5
6       // This does the same thing as above.
7       let (a, b, c) = tuple;
8
9       // Ignore the first element, only bind the
        second and third.
10      let (_, b, c) = tuple;
11
12      // Ignore everything but the last element.
13      let (.., c) = tuple;
14  }                                   (playground link)
```

What happens when adding or removing an element to the tuple and look at the resulting compiler errors

The destructuring pattern must match the structure of the value exactly.

Info

An **irrefutable pattern** is a pattern that always matches.

```
1   fn takes_tuple(tuple: (char, i32, bool)) {
2       let a = tuple.0;
3       let b = tuple.1;
4       let c = tuple.2;
5
6       // This does the same thing as above.
7       let (a, b, c) = tuple;
8
9       // Ignore the first element, only bind the
        second and third.
10      let (_, b, c) = tuple;
11
12      // Ignore everything but the last element.
13      let (.., c) = tuple;
14  }                                   (playground link)
```

What happens when adding or removing an element to the tuple and look at the resulting compiler errors

The destructuring pattern must match the structure of the value exactly.

Info

A variable binding is actually a pattern itself!

The _ pattern matches anything.

7.1. Irrefutable patterns

7.1.1. Advanced usage of ..

Ignoring middle element:

```
1 fn takes_tuple(tuple: (char, i32, bool, u8)) {
2     let (first, .., last) = tuple;
3 }
```

(playground link)

7.1. Irrefutable patterns

7.1.1. Advanced usage of ..

Ignoring middle element:

```
1 fn takes_tuple(tuple: (char, i32, bool, u8)) {
2     let (first, .., last) = tuple;
3 }
```
(playground link)

Works with arrays as well:

```
1 fn takes_array(array: [u8; 5]) {
2   let [first, .., last] = array;
3 }
```
(playground link)

7.1. Irrefutable patterns

7.1.1. Advanced usage of ..

Ignoring middle element:

```
1 fn takes_tuple(tuple: (char, i32, bool, u8)) {
2     let (first, .., last) = tuple;
3 }                                                    (playground link)
```

Works with arrays as well:

```
1 fn takes_array(array: [u8; 5]) {
2   let [first, .., last] = array;
3 }                                                    (playground link)
```

Warning

Pattern matching works on all data types in Rust!

7.2. Matching values

Patterns can also be simple values like characters:

```rust
#[rustfmt::skip]
fn main() {
    let input = 'x';
    match input {
        'q'                          => println!("Quitting"),
        'a' | 's' | 'w' | 'd'        => println!("Moving around"),
        '0'..='9'                    => println!("Number input"),
        key if key.is_lowercase() => println!("Lowercase: {key}"),
        _                            => println!("Something else"),
    }
}
```

(playground link)

A variable in the pattern (key in this example) will create a binding that can be used within the match arm.

7.2. Matching values

A **match guard**:

- causes the arm to match only if the condition is true.
- If the condition is false the match will continue checking later cases.

```
1 #[rustfmt::skip]
2 fn main() {
3     let input = 'a';
4     match input {
5         key if key.is_uppercase() => println!("Uppercase"),
6         key => if input == 'q' { println!("Quitting") },
7         _   => println!("Lowercase"),
8     }
9 }
```

(playground link)

What is the bug in this program?

7.2. Matching values

A **match guard**:

- causes the arm to match only if the condition is true.
- If the condition is false the match will continue checking later cases.

```rust
#[rustfmt::skip]
fn main() {
    let input = 'a';
    match input {
        key if key.is_uppercase() => println!("Uppercase"),
        key => if input == 'q' { println!("Quitting") },
        _   => println!("Lowercase"),
    }
}
```

(playground link)

What is the bug in this program?

The second arm will always match, so the third arm is unreachable. Write match guard in front of the second arm: `key if key == 'q' => ...`

Warning

Can't use an existing variable as the condition in a match arm, as it will instead be interpreted as a variable name pattern, which creates a new variable that will shadow the existing one.

```
1 let expected = 5;
2 match 123 {
3     expected => println!("Expected value is 5, actual is {expected}"),
4     _ => println!("Value was something else"),
5 }
```
(playground link)

What is the output of this program?

Warning

Can't use an existing variable as the condition in a match arm, as it will instead be interpreted as a variable name pattern, which creates a new variable that will shadow the existing one.

```
1 let expected = 5;
2 match 123 {
3     expected => println!("Expected value is 5, actual is {expected}"),
4     _ => println!("Value was something else"),
5 }
```
(playground link)

What is the output of this program?

Will always print "Expected value is 5, actual is 123".

7.2. Matching values

Use `@` to bind to a value while also testing it:

```
let expected = 5;
match 123 {
    val @ 1..=10 => println!("Value {val} is between 1 and 10"),
    val @ 11..=20 => println!("Value {val} is between 11 and 20"),
    val => println!("Value {val} is something else"),
}
```

(playground link)

7.3. Structs

Like tuples, structs can also be destructured by matching:

```
1  struct Foo {
2      x: (u32, u32),
3      y: u32,
4  }
5
6  #[rustfmt::skip]
7  fn main() {
8      let foo = Foo { x: (1, 2), y: 3 };
9      match foo {
10         Foo { y: 2, x: i }   => println!("y = 2, x = {i:?}"),
11         Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
12         Foo { y, .. }        => println!("y = {y}, other fields were ignored"),
13     }
14 }
```

(playground link)

Add or remove fields to `Foo` and see what happens!

7.3.1. Reference patterns

When matching on a reference, Rust **automatically dereferences** it for you:

```
1 fn main() {
2     let foo = Foo { x: (1, 2), y: 3 };
3     match &foo {  // `foo` is turned into a reference
4         Foo { y: 2, x: i }   => println!("y = 2, x = {i:?}"),
5         Foo { x: (1, b), y } => println!("x.0 = 1, b = {b}, y = {y}"),
6         Foo { y, .. }        => println!("y = {y}, other fields were ignored"),
7     }
8 }
```

(playground link)

Info

The pattern `Foo { ... }` works on `&foo` because Rust automatically dereferences. You could also write `&Foo { ... }` explicitly, but it's not required.

7.3. Structs

Two equivalent ways to match on a reference:

```
1 match &foo {
2     Foo { y, .. } => println!("y = {y}"),  // Implicit dereference
3 }
4
5 match &foo {
6     &Foo { y, .. } => println!("y = {y}"), // Explicit dereference pattern
7 }                                                          (playground link)
```

Pick the one you like best!

Warning

In the implicit version, bound variables like `y` will be references. In the explicit version with `&Foo`, `y` will be a value (copied from the struct).

7.3.2. Mutable reference patterns

When matching on `&mut`, bound variables are mutable references:

```
1   struct Foo {
2       x: (u32, u32),
3       y: u32,
4   }
5   fn main() {
6       let mut foo = Foo { x: (1, 2), y: 3 };
7       match &mut foo {
8           Foo { x: (1, 2), y } => *y = 4,  // y is &mut u32, need * to assign
9           Foo { y, .. }        => println!("y = {y}"),
10      }
11  }
```

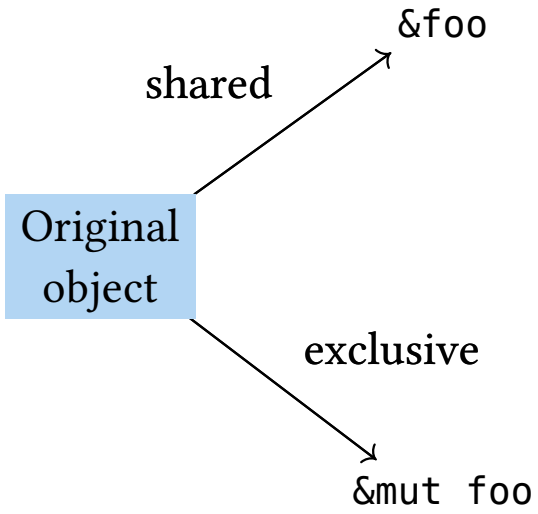
(playground link)

Warning

Bound variables like `y` are `&mut u32`, not `u32`. You must dereference with `*` to assign values.

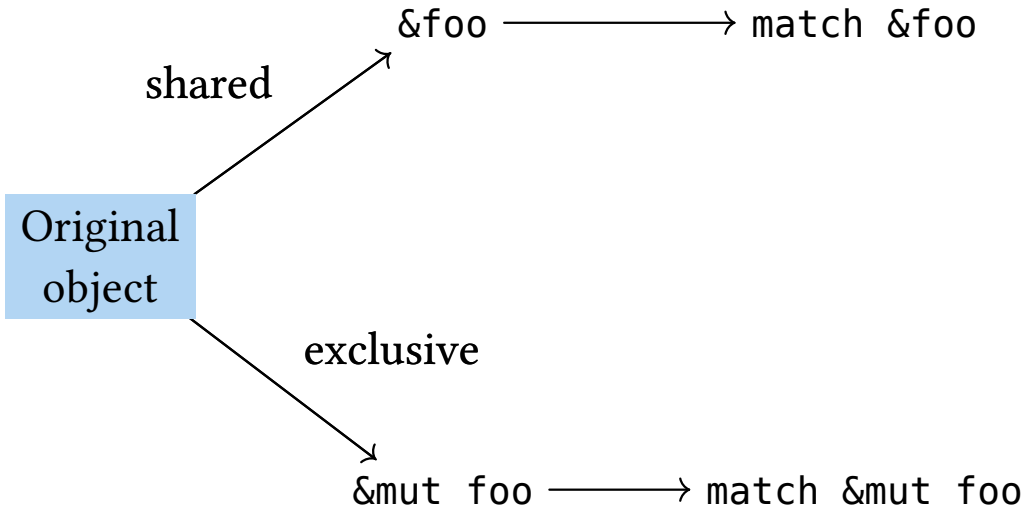
7.3. Structs

7.3.3. Intuition



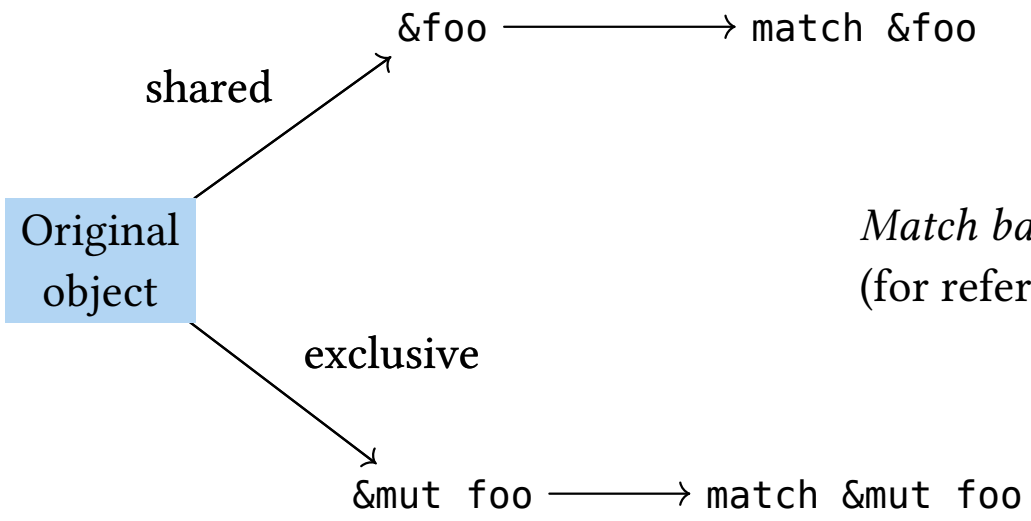
7.3. Structs

7.3.3. Intuition



7.3. Structs

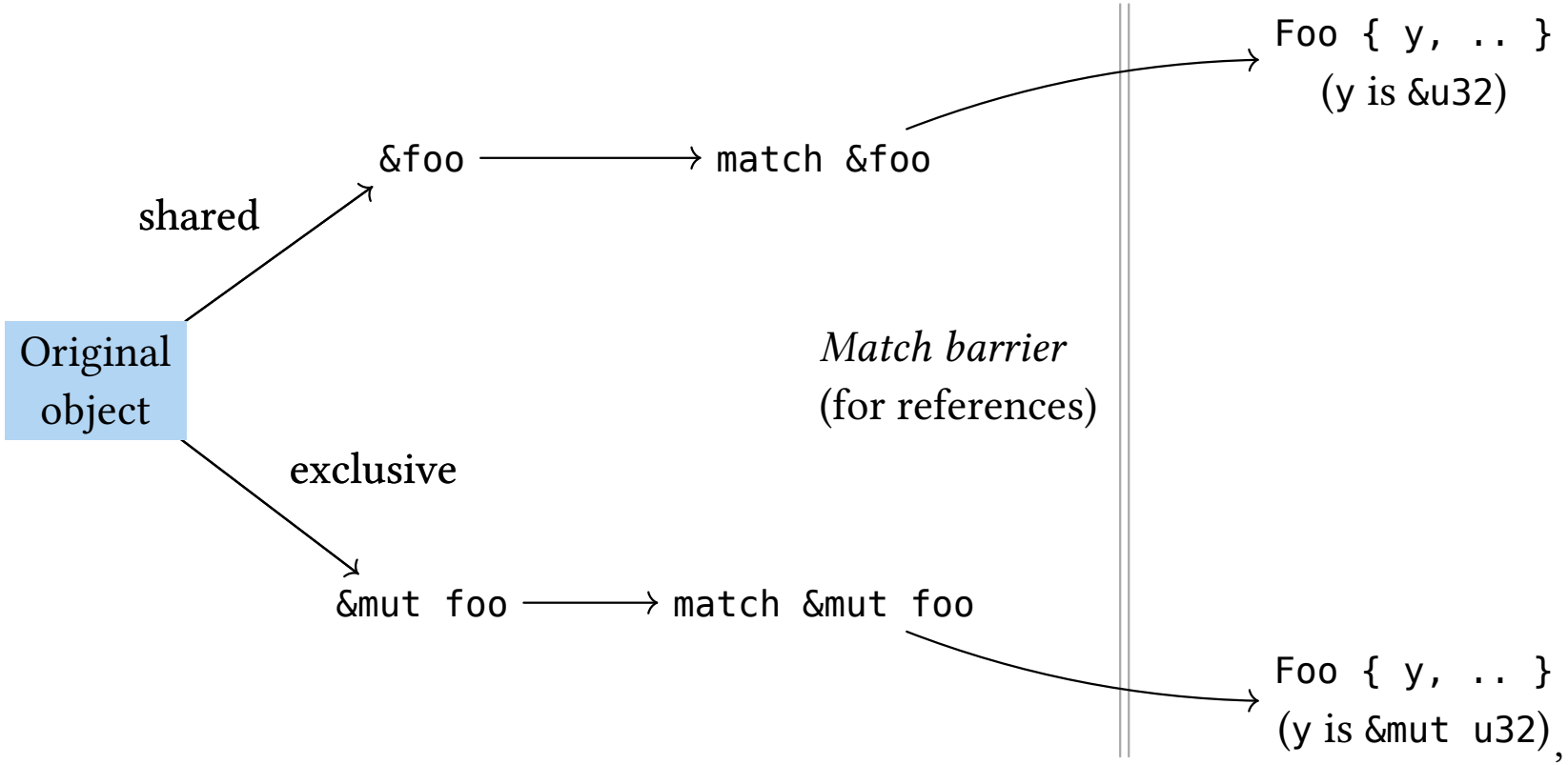
7.3.3. Intuition



Match barrier
(for references)

7.3. Structs

7.3.3. Intuition



7.4. Enums

Like tuples, enums can also be destructured by matching:

```
enum Result {
    Ok(i32),
    Err(String),
}

fn divide_in_two(n: i32) -> Result {
    if n % 2 == 0 {
        Result::Ok(n / 2)
    } else {
        Result::Err(format!("cannot divide {n} into two equal parts"))
    }
}
```

(playground link)

7.4. Enums

Like tuples, enums can also be destructured by matching:

```
    enum Result {
        Ok(i32),
        Err(String),
    }

    fn divide_in_two(n: i32) -> Result {
        if n % 2 == 0 {
            Result::Ok(n / 2)
        } else {
            Result::Err(format!("cannot divide {n} into two equal parts"))
        }
    }
```

(playground link)

```
fn main() {
     let n = 100;
     match divide_in_two(n) {
         Result::Ok(half) => println!("{n} divided in two is {half}"),
         Result::Err(msg) => println!("sorry, an error happened: {msg}"),
     }
}
```

(playground link)

Warning

Rust does not allow non-exhaustive matches

7.5. Let control flow

```rust
use std::time::Duration;

fn sleep_for(secs: f32) {
    let result = Duration::try_from_secs_f32(secs);

    if let Ok(duration) = result {
        std::thread::sleep(duration);
        println!("slept for {duration:?}");
    }
}

fn main() {
    sleep_for(-10.0);
    sleep_for(0.8);
}
```

(playground link)

7.5. Let control flow

When to use `if let` over `match`?

7.5. Let control flow

```rust
use std::time::Duration;

fn sleep_for(secs: f32) {
    let result = Duration::try_from_secs_f32(secs);

    if let Ok(duration) = result {
        std::thread::sleep(duration);
        println!("slept for {duration:?}");
    }
}

fn main() {
    sleep_for(-10.0);
    sleep_for(0.8);
}
```

(playground link)

7.5. Let control flow

When to use `if let` over `match`?

Use `if let` when you only care about one specific pattern and want to ignore all others.

7.6. Exercise: Expression evaluation

Let's write a simple recursive evaluator for arithmetic expressions.

Complete the `eval` function in the file `session-1/examples/s1e6-evaluation.rs`.

8.1. Methods

In Rust, methods are separated from fields:

```rust
#[derive(Debug)]
struct CarRace {
    name: String,
    laps: Vec<i32>,
}

impl CarRace {
    // No receiver, a static method
    fn new(name: &str) -> Self {
        Self { name: String::from(name), laps: Vec::new() }
    }
    // Exclusive borrowed read-write access to self
    fn add_lap(&mut self, lap: i32) {
        self.laps.push(lap);
    }
}

fn main() {
    let mut race = CarRace::new("Monaco Grand Prix");
    race.add_lap(70);
}
```

(playground link)

8.1. Methods

Like in other languages, methods group functionality around *acting* data types, types that *do* related things.

How are methods in Rust different from methods in other languages?

8.1. Methods

Like in other languages, methods group functionality around *acting* data types, types that *do* related things.

How are methods in Rust different from methods in other languages?

Methods cannot be overridden.

```
1 impl CarRace {
2     fn add_lap(&mut self, lap: i32) {
3         self.laps.push(lap);
4     }
5 }
```
(playground link)

What is the `&mut self` parameter syntax sugar for?

8.1. Methods

Like in other languages, methods group functionality around *acting* data types, types that *do* related things.

How are methods in Rust different from methods in other languages?

Methods cannot be overridden.

```
1 impl CarRace {
2     fn add_lap(&mut self, lap: i32) {
3         self.laps.push(lap);
4     }
5 }
```
(playground link)

What is the `&mut self` parameter syntax sugar for?

For `self: &mut Self`.

```
1 impl CarRace {
2     fn add_lap(self: &mut Self, lap: i32) {
3         self.laps.push(lap);
4     }
5 }
```
(playground link)

8.1. Methods

Methods can **consume** `self`:

```
1  #[derive(Debug)]
2  struct CarRace {
3      name: String,
4      laps: Vec<i32>,
5  }
6
7  impl CarRace {
8      // Exclusive ownership of self (covered later)
9      fn finish(self) {
10         let total: i32 = self.laps.iter().sum();
11         println!("Race {} is finished, total lap time:
           {}", self.name, total);
12     }
13 }
14
15 fn main() {
16     let mut race = CarRace::new("Monaco Grand Prix");
17     race.add_lap(70);
18     race.finish();
19     // race.add_lap(42);
20 }                                          (playground link)
```

8.1. Methods

Methods can **consume** `self`:

```
1  #[derive(Debug)]
2  struct CarRace {
3      name: String,
4      laps: Vec<i32>,
5  }
6
7  impl CarRace {
8      // Exclusive ownership of self (covered later)
9      fn finish(self) {
10         let total: i32 = self.laps.iter().sum();
11         println!("Race {} is finished, total lap time:
           {}", self.name, total);
12     }
13 }
14
15 fn main() {
16     let mut race = CarRace::new("Monaco Grand Prix");
17     race.add_lap(70);
18     race.finish();
19     // race.add_lap(42);
20 }                                          (playground link)
```

What is the name given to the `self` parameter in methods?

8.1. Methods

Methods can **consume** `self`:

```
1  #[derive(Debug)]
2  struct CarRace {
3      name: String,
4      laps: Vec<i32>,
5  }
6
7  impl CarRace {
8      // Exclusive ownership of self (covered later)
9      fn finish(self) {
10         let total: i32 = self.laps.iter().sum();
11         println!("Race {} is finished, total lap time:
           {}", self.name, total);
12     }
13 }
14
15 fn main() {
16     let mut race = CarRace::new("Monaco Grand Prix");
17     race.add_lap(70);
18     race.finish();
19     // race.add_lap(42);
20 }                                          (playground link)
```

What is the name given to the `self` parameter in methods?

`self` is called the **receiver**.

After calling a method with `object.finish(self)`, you can no longer use `object`. It has been *consumed*.

Warning

Use the `self` receiver to define destructors or functionality that should happen only once at the end.

8.2. Receivers

Rust allows receivers to be one of the following:

- `self`, consuming self
- any kind of reference to self: `&self`, `&mut self`
- exceptions (covered later)

Name a few reference types in Rust.

8.2. Receivers

Rust allows receivers to be one of the following:

- `self`, consuming self
- any kind of reference to self: `&self`, `&mut self`
- exceptions (covered later)

Name a few reference types in Rust.

`&T`, `&mut T`, `Box<T>`, `Rc<T>`, `Arc<T>`, ...

Info

Reference types are also called **wrappers** and some are **smart pointers**.

8.3. Traits

Rust lets you abstract over types with traits. Similar to interfaces:

```
1 trait Pet {
2     /// Return a sentence from this pet.
3     fn talk(&self) -> String;
4
5     /// Print a string to the terminal greeting this pet.
6     fn greet(&self);
7 }
```

(playground link)

Properties of traits:

8.3. Traits

Rust lets you abstract over types with traits. Similar to interfaces:

```
1 trait Pet {
2     /// Return a sentence from this pet.
3     fn talk(&self) -> String;
4
5     /// Print a string to the terminal greeting this pet.
6     fn greet(&self);
7 }
```

(playground link)

Properties of traits:

- A trait is a list of methods that a type **must implement**

8.3. Traits

Rust lets you abstract over types with traits. Similar to interfaces:

```
1 trait Pet {
2     /// Return a sentence from this pet.
3     fn talk(&self) -> String;
4
5     /// Print a string to the terminal greeting this pet.
6     fn greet(&self);
7 }
```

(playground link)

Properties of traits:

- A trait is a list of methods that a type **must implement**
- The **signatures** of the type's methods **must be identical** to the trait's method signatures

8.3. Traits

Rust lets you abstract over types with traits. Similar to interfaces:

```
1 trait Pet {
2     /// Return a sentence from this pet.
3     fn talk(&self) -> String;
4
5     /// Print a string to the terminal greeting this pet.
6     fn greet(&self);
7 }
```

(playground link)

Properties of traits:

- A trait is a list of methods that a type **must implement**
- The **signatures** of the type's methods **must be identical** to the trait's method signatures
- **Default implementations** can be provided

8.4. Implementing traits

```
1   trait Pet {
2     fn talk(&self) -> String;
3
4     fn greet(&self) {
5         println!("Oh you're a cutie!
          What's your name? {}",
          self.talk());
6     }
7   }
8
9   struct Dog {
10    name: String,
11    age: i8,
12  }                          (playground link)
```

8.4. Implementing traits

```rust
trait Pet {
    fn talk(&self) -> String;

    fn greet(&self) {
        println!("Oh you're a cutie! What's your name? {}", self.talk());
    }
}

struct Dog {
    name: String,
    age: i8,
}
```

(playground link)

```rust
impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Woof, my name is {}!", self.name)
    }
}

fn main() {
    let fido = Dog { name: String::from("Fido"), age: 5 };
    dbg!(fido.talk());
    fido.greet();
}
```

(playground link)

8.4. Implementing traits

```rust
trait Pet {
    fn talk(&self) -> String;

    fn greet(&self) {
        println!("Oh you're a cutie! What's your name? {}", self.talk());
    }
}

struct Dog {
    name: String,
    age: i8,
}
```

(playground link)

```rust
impl Pet for Dog {
    fn talk(&self) -> String {
        format!("Woof, my name is {}!", self.name)
    }
}

fn main() {
    let fido = Dog { name: String::from("Fido"), age: 5 };
    dbg!(fido.talk());
    fido.greet();
}
```

(playground link)

Warning

a Cat type with a `talk()` method would not automatically satisfy Pet unless it is in an impl Pet block

8.4. Implementing traits

You can:

- split trait implementation blocks
- override default method implementations

```
1     impl Pet for Dog {
2         fn talk(&self) -> String {
3             format!("Woof, my name is {}!", self.name)
4         }
5     }
6
7     impl Pet for Dog {
8         fn greet(&self) -> String {
9             format!("Woof, my name is {}!", self.name)
10        }
11    }                                                    (playground link)
```

Keywords

level of pointer indirection each time a new object is added to the mix (and, practically, a heap allocation).

Although there were other reasons as well, this issue of expensive composition is the key thing that drove Rust towards adopting a different model. It is particularly a problem when one considers, for example, the implications of composing together the `Future`s which will eventually make up an asynchronous task (including address-sensitive `async fn` state machines). It is plausible that there could be many layers of `Future`s composed together, including multiple layers of `async fn`s handling different parts of a task. It was deemed unacceptable to force indirection and allocation for each layer of composition in this case.

`Pin<Ptr>` is an implementation of the third option. It allows us to solve the issues discussed with the second option by building a *shared contractual language* around the guarantees of "pinning" data.

Using `Pin<Ptr>` to pin values

In order to pin a value, we wrap a *pointer to that value* (of some type `Ptr`) in a `Pin<Ptr>`. `Pin<Ptr>` can wrap any pointer type, forming a promise that the **pointee** will not be *moved* or otherwise invalidated.

We call such a `Pin`-wrapped pointer a **pinning pointer,** (or pinning reference, or pinning `Box`, etc.) because its existence is the thing that is conceptually pinning the underlying pointee in place: it is the metaphorical "pin" securing the data in place on the pinboard (in memory).

Notice that the thing wrapped by `Pin` is not the value which we want to pin itself, but rather a pointer to that value! A `Pin<Ptr>` does not pin the `Ptr`; instead, it pins the pointer's ***pointee** value*.

Pinning as a library contract

Pinning does not require nor make use of any compiler "magic"[2], only a specific contract between the `unsafe` parts of a library API and its users.

It is important to stress this point as a user of the `unsafe` parts of the `Pin` API. Practically, this means that performing the

8.5. Contracts

Rust, being a systems programming language, focuses on safety.

8.5. Contracts

Rust, being a systems programming language, focuses on safety.

In Rust documentation, you will often encounter the word **contract**.

Info

A contract is an agreement between two parties. There are mainly two kinds of contracts in Rust:

- Contracts between **two pieces of code**
- Contracts between **the programmer and the compiler**

8.5. Contracts

Rust, being a systems programming language, focuses on safety.

In Rust documentation, you will often encounter the word **contract**.

Info

A contract is an agreement between two parties. There are mainly two kinds of contracts in Rust:

- Contracts between **two pieces of code**
- Contracts between **the programmer and the compiler**

Contracts between two pieces of code are generally safe (can be enforced automatically).

8.5. Contracts

Rust, being a systems programming language, focuses on safety.

In Rust documentation, you will often encounter the word **contract**.

Info

A contract is an agreement between two parties. There are mainly two kinds of contracts in Rust:

- Contracts between **two pieces of code**
- Contracts between **the programmer and the compiler**

Contracts between two pieces of code are generally safe (can be enforced automatically).

Warning

... but contracts between the programmer and the compiler can be unsafe or **may have to be checked by the programmer**.

Examples of such contracts: `Pin`, `Send`, `Sync` traits (advanced topic).

8.6. Super-traits

```
1  trait Animal {
2      fn leg_count(&self) -> u32;
3  }
4
5  trait Pet: Animal {
6      fn name(&self) -> String;
7  }
8
9  struct Dog(String);
10
11 impl Animal for Dog {
12     fn leg_count(&self) -> u32 {
13         4
14     }
15 }
16
17 impl Pet for Dog {
18     fn name(&self) -> String {
19         self.0.clone()
20     }
21 }                                (playground link)
```

Super-traits are an extra constraint on traits that say: “to implement this trait, you must also implement that other trait”.

8.6. Super-traits

```
1  trait Animal {
2      fn leg_count(&self) -> u32;
3  }
4
5  trait Pet: Animal {
6      fn name(&self) -> String;
7  }
8
9  struct Dog(String);
10
11 impl Animal for Dog {
12     fn leg_count(&self) -> u32 {
13         4
14     }
15 }
16
17 impl Pet for Dog {
18     fn name(&self) -> String {
19         self.0.clone()
20     }
21 }                                  (playground link)
```

Super-traits are an extra constraint on traits that say: "to implement this trait, you must also implement that other trait".

Warning

Super traits are the kind of language feature you should **avoid as long as you are stuck with the OOP mindset**.

Once you are willing to forget OOP, you can see super-traits are actually easy.

8.6. Super-traits

```
1  trait Animal {
2      fn leg_count(&self) -> u32;
3  }
4
5  trait Pet: Animal {
6      fn name(&self) -> String;
7  }
8
9  struct Dog(String);
10
11 impl Animal for Dog {
12     fn leg_count(&self) -> u32 {
13         4
14     }
15 }
16
17 impl Pet for Dog {
18     fn name(&self) -> String {
19         self.0.clone()
20     }
21 }                                  (playground link)
```

Super-traits are an extra constraint on traits that say: "to implement this trait, you must also implement that other trait".

Warning

Super traits are the kind of language feature you should **avoid as long as you are stuck with the OOP mindset**.

Once you are willing to forget OOP, you can see super-traits are actually easy.

Advanced

... at least as long as you don't constrain **associated types** of super traits in subtraits

8.7. Associated Types

```
1  #[derive(Debug)]
2  struct Meters(i32);
3  #[derive(Debug)]
4  struct MetersSquared(i32);
5
6  trait Multiply {
7      type Output;
8      fn multiply(&self, other: &Self) -> Self::Output;
9  }
10
11 impl Multiply for Meters {
12     type Output = MetersSquared;
13     fn multiply(&self, other: &Self) -> Self::Output {
14         MetersSquared(self.0 * other.0)
15     }
16 }
17
18 fn main() {
19     println!("{:?}", Meters(10).multiply(&Meters(20)));
20 }                                          (playground link)
```

Associated types are **placeholder types that** are supplied by the trait implementation.

8.7. Associated Types

```
1  #[derive(Debug)]
2  struct Meters(i32);
3  #[derive(Debug)]
4  struct MetersSquared(i32);
5
6  trait Multiply {
7      type Output;
8      fn multiply(&self, other: &Self) -> Self::Output;
9  }
10
11 impl Multiply for Meters {
12     type Output = MetersSquared;
13     fn multiply(&self, other: &Self) -> Self::Output {
14         MetersSquared(self.0 * other.0)
15     }
16 }
17
18 fn main() {
19     println!("{:?}", Meters(10).multiply(&Meters(20)));
20 }                                          (playground link)
```

Associated types are **placeholder types that** are supplied by the trait implementation.

Why are associated types are sometimes also called “output types”

8.7. Associated Types

```
1  #[derive(Debug)]
2  struct Meters(i32);
3  #[derive(Debug)]
4  struct MetersSquared(i32);
5
6  trait Multiply {
7      type Output;
8      fn multiply(&self, other: &Self) -> Self::Output;
9  }
10
11 impl Multiply for Meters {
12     type Output = MetersSquared;
13     fn multiply(&self, other: &Self) -> Self::Output {
14         MetersSquared(self.0 * other.0)
15     }
16 }
17
18 fn main() {
19     println!("{:?}", Meters(10).multiply(&Meters(20)));
20 }
```

(playground link)

Associated types are **placeholder types that** are supplied by the trait implementation.

Why are associated types are sometimes also called “output types”

The implementer, not the caller, chooses the concrete associated type. (Compare with generics.)

8.7. Associated Types

```
1  #[derive(Debug)]
2  struct Meters(i32);
3  #[derive(Debug)]
4  struct MetersSquared(i32);
5
6  trait Multiply {
7      type Output;
8      fn multiply(&self, other: &Self) -> Self::Output;
9  }
10
11 impl Multiply for Meters {
12     type Output = MetersSquared;
13     fn multiply(&self, other: &Self) -> Self::Output {
14         MetersSquared(self.0 * other.0)
15     }
16 }
17
18 fn main() {
19     println!("{:?}", Meters(10).multiply(&Meters(20)));
20 }                                          (playground link)
```

Associated types are **placeholder types that** are supplied by the trait implementation.

Why are associated types are sometimes also called “output types”

The implementer, not the caller, chooses the concrete associated type. (Compare with generics.)

Iterators from the standard library have an associated type `Item`:

```
1 pub trait Iterator {
2     type Item;
3     fn next(&mut self) ->
      Option<Self::Item>;
4 }                                  (playground link)
```

8.8. Deriving

Supported traits can be automatically implemented for your custom types, as follows:

```
1  #[derive(Debug, Clone, Default)]
2  struct Player {
3      name: String,
4      strength: u8,
5      hit_points: u8,
6  }
7
8  fn main() {
9      let p1 = Player::default(); // Default trait adds `default` constructor.
10     let mut p2 = p1.clone(); // Clone trait adds `clone` method.
11     p2.name = String::from("ElderScrollz");
12     // Debug trait adds support for printing with `{:?}`.
13     println!("{p1:?} vs. {p2:?}");
14 }
```

(playground link)

How is the derive functionality implemented in Rust?

8.8. Deriving

Supported traits can be automatically implemented for your custom types, as follows:

```
1  #[derive(Debug, Clone, Default)]
2  struct Player {
3      name: String,
4      strength: u8,
5      hit_points: u8,
6  }
7
8  fn main() {
9      let p1 = Player::default(); // Default trait adds `default` constructor.
10     let mut p2 = p1.clone(); // Clone trait adds `clone` method.
11     p2.name = String::from("ElderScrollz");
12     // Debug trait adds support for printing with `{:?}`.
13     println!("{p1:?} vs. {p2:?}");
14 }
```

(playground link)

How is the derive functionality implemented in Rust?

With **procedural macros**, and many crates provide useful derive macros to add useful functionality.

8.8. Deriving

Supported traits can be automatically implemented for your custom types, as follows:

```
#[derive(Debug, Clone, Default)]
struct Player {
    name: String,
    strength: u8,
    hit_points: u8,
}

fn main() {
    let p1 = Player::default(); // Default trait adds `default` constructor.
    let mut p2 = p1.clone(); // Clone trait adds `clone` method.
    p2.name = String::from("EldurScrollz");
    // Debug trait adds support for printing with `{:?}`.
    println!("{p1:?} vs. {p2:?}");
}
```

(playground link)

How is the derive functionality implemented in Rust?

With **procedural macros**, and many crates provide useful derive macros to add useful functionality.

For example, `serde` can derive serialization support for a struct using `#[derive(Serialize)]`.

8.8. Deriving

8.8.1. Why is deriving useful?

A manual implementation of the `Clone` trait for the `Player` struct would look like this:

```
1 impl Clone for Player {
2     fn clone(&self) -> Self {
3         Player {
4             name: self.name.clone(),
5             strength: self.strength.clone(),
6             hit_points: self.hit_points.clone(),
7         }
8     }
9 }
```

(playground link)

It is easier to just write `#[derive(Clone)]`

Info

The derive attribute is similar to `deriving` in Haskell.

8.9. Exercise: Logger trait

Complete the test code in `session-1/tests/s1e8-min.rs`.

Homework (**deadline = next session**):

- **Read about generics** in chapter 11 of "Programming Rust" (2nd edition): "Traits and Generics" p. 384-431 (PDF available)
- Do exercise on generics in `session-1/examples/s1e8-generics.rs`

Homework (**deadline = next session**):

- **Read about generics** in chapter 11 of "Programming Rust" (2nd edition): "Traits and Generics" p. 384-431 (PDF available)
- Do exercise on generics in `session-1/examples/s1e8-generics.rs`
- Think about **who you would like to work with** on a project and what you want to do.

Homework (**deadline = next session**):

- **Read about generics** in chapter 11 of "Programming Rust" (2nd edition): "Traits and Generics" p. 384-431 (PDF available)
- Do exercise on generics in `session-1/examples/s1e8-generics.rs`
- Think about **who you would like to work with** on a project and what you want to do.

Next week:

- Test about today's topics.
- New topics: closures, standard library

Homework (**deadline = next session**):

- **Read about generics** in chapter 11 of "Programming Rust" (2nd edition): "Traits and Generics" p. 384-431 (PDF available)
- Do exercise on generics in `session-1/examples/s1e8-generics.rs`
- Think about **who you would like to work with** on a project and what you want to do.

Next week:

- Test about today's topics.
- New topics: closures, standard library

Questions?

Feel free to contact me:

- Video meeting / email: willemvanhulle@protonmail.com
- Phone: +32 479 080 252