Introduction to Rust 2

Includes material from Ferrous Systems' rust training course.

CC-BY-SA 4.0.

Agenda

- Compound Types
- Error Handling
- Methods and Traits
- Lifetimes
- Modules

```
git clone https://github.com/Awlaursen/rust-examples
```

```
git pull
```

Compound Types

Structs

A `struct` groups and names data of different types.

Definition

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

RUN

Construction

- there is no partial initialization

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

fn main() {
    let p = Point { x: 1, y: 2 };
}
```

RUN

Construction

- there is no partial initialization

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

fn main() {
    let p = Point { x: 1, y: 2 };
}
```

RUN

Tuples

- Holds values of different types together.
- Like an anonymous `struct`, with fields numbered 0, 1, etc.

```rust
fn main() {
    let p = (1, 2);
    println!("{}", p.0);
    println!("{}", p.1);
}
```

RUN

Tuples

- Holds values of different types together.
- Like an anonymous `struct`, with fields numbered 0, 1, etc.

```rust
fn main() {
    let p = (1, 2);
    println!("{}", p.0);
    println!("{}", p.1);
}
```

RUN

()

- the *empty tuple*
- represents the absence of data
- we often use this similarly to how you'd use `void` in C

```
fn prints_but_returns_nothing(data: &str) -> () {
    println!("passed string: {}", data);
}
```

RUN

Tuple Structs

- Like a `struct`, with fields numbered 0, 1, etc.

```
struct Point(i32,i32);

fn main() {
    let p = Point(1, 2);
    println!("{}", p.0);
    println!("{}", p.1);
}
```

RUN

Tuple Structs

- Like a `struct`, with fields numbered 0, 1, etc.

```rust
struct Point(i32,i32);

fn main() {
    let p = Point(1, 2);
    println!("{}", p.0);
    println!("{}", p.1);
}
```

RUN

Tuple Structs

- Like a `struct`, with fields numbered 0, 1, etc.

```rust
struct Point(i32,i32);

fn main() {
    let p = Point(1, 2);
    println!("{}", p.0);
    println!("{}", p.1);
}
```

RUN

Enums

- An `enum` represents different variations of the same subject.
- The different choices in an enum are called *variants*

enum: Definition and Construction

```rust
enum Shape {
    Square,
    Circle,
    Rectangle,
    Triangle,
}

fn main() {
    let shape = Shape::Rectangle;
}
```

RUN

enum: Definition and Construction

```rust
enum Shape {
    Square,
    Circle,
    Rectangle,
    Triangle,
}

fn main() {
    let shape = Shape::Rectangle;
}
```

RUN

Enums with Values

```rust
enum Movement {
    Right(i32),
    Left(i32),
    Up(i32),
    Down { speed: i32, excitement: u8 },
}

fn main() {
    let movement = Movement::Left(12);
    let movement = Movement::Down { speed: 12, excitement: 5 };
}
```

RUN

Enums with Values

```rust
enum Movement {
    Right(i32),
    Left(i32),
    Up(i32),
    Down { speed: i32, excitement: u8 },
}

fn main() {
    let movement = Movement::Left(12);
    let movement = Movement::Down { speed: 12, excitement: 5 };
}
```

RUN

Enums with Values

```rust
enum Movement {
    Right(i32),
    Left(i32),
    Up(i32),
    Down { speed: i32, excitement: u8 },
}

fn main() {
    let movement = Movement::Left(12);
    let movement = Movement::Down { speed: 12, excitement: 5 };
}
```

RUN

Enums with Values

```rust
enum Movement {
    Right(i32),
    Left(i32),
    Up(i32),
    Down { speed: i32, excitement: u8 },
}

fn main() {
    let movement = Movement::Left(12);
    let movement = Movement::Down { speed: 12, excitement: 5 };
}
```

RUN

Enums with Values

```rust
enum Movement {
    Right(i32),
    Left(i32),
    Up(i32),
    Down { speed: i32, excitement: u8 },
}

fn main() {
    let movement = Movement::Left(12);
    let movement = Movement::Down { speed: 12, excitement: 5 };
}
```

RUN

Enums with Values

- An enum value is the same size, no matter which variant is picked
- It will be the size of the largest variant (plus a tag)

Doing a `match` on an `enum`

- When an `enum` has variants, you use `match` to extract the data
- New variables are created from the *pattern* (e.g. `radius`)

```rust
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
}

fn check_shape(shape: Shape) {
    match shape {
        Shape::Circle(radius) => {
            println!("It's a circle, with radius {}", radius);
        }
        _ => {
            println!("Try a circle instead");
        }
    }
}
```

RUN

Doing a `match` on an `enum`

- When an `enum` has variants, you use `match` to extract the data
- New variables are created from the *pattern* (e.g. `radius`)

```
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
}

fn check_shape(shape: Shape) {
    match shape {
        Shape::Circle(radius) => {
            println!("It's a circle, with radius {}", radius);
        }
        _ => {
            println!("Try a circle instead");
        }
    }
}
```

RUN

Doing a `match` on an `enum`

- When an `enum` has variants, you use `match` to extract the data
- New variables are created from the *pattern* (e.g. `radius`)

```
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
}

fn check_shape(shape: Shape) {
    match shape {
        Shape::Circle(radius) => {
            println!("It's a circle, with radius {}", radius);
        }
        _ => {
            println!("Try a circle instead");
        }
    }
}
```

RUN

Doing a `match` on an `enum`

- When an `enum` has variants, you use `match` to extract the data
- New variables are created from the *pattern* (e.g. `radius`)

```rust
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
}

fn check_shape(shape: Shape) {
    match shape {
        Shape::Circle(radius) => {
            println!("It's a circle, with radius {}", radius);
        }
        _ => {
            println!("Try a circle instead");
        }
    }
}
```

RUN

Doing a `match` on an `enum`

- There are two variables called `radius`
- The binding of `radius` in the pattern on line 9 hides the `radius` variable on line 7

```
1  enum Shape {
2      Circle(i32),
3      Rectangle(i32, i32),
4  }
5
6  fn check_shape(shape: Shape) {
7      let radius = 10;
8      match shape {
9          Shape::Circle(radius) => {
10             println!("It's a circle, with radius {}", radius);
11         }
12         _ => {
13             println!("Try a circle instead");
14         }
15     }
16 }
```

RUN

Doing a `match` on an `enum`

- There are two variables called `radius`
- The binding of `radius` in the pattern on line 9 hides the `radius` variable on line 7

```
1  enum Shape {
2      Circle(i32),
3      Rectangle(i32, i32),
4  }
5
6  fn check_shape(shape: Shape) {
7      let radius = 10;
8      match shape {
9          Shape::Circle(radius) => {
10             println!("It's a circle, with radius {}", radius);
11         }
12         _ => {
13             println!("Try a circle instead");
14         }
15     }
16 }
```

RUN

Match guards

Match guards allow further refining of a `match`

```rust
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
}

fn check_shape(shape: Shape) {
    match shape {
        Shape::Circle(radius) if radius > 10 => {
            println!("It's a BIG circle, with radius {}", radius);
        }
        _ => {
            println!("Try a big circle instead");
        }
    }
}
```

RUN

Combining patterns

- You can use the | operator to join patterns together

```rust
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
    Square(i32),
}

fn test_shape(shape: Shape) {
    match shape {
        Shape::Circle(size) | Shape::Square(size) => {
            println!("Shape has single size field {}", size);
        }
        _ => {
            println!("Not a circle, nor a square");
        }
    }
}
```

RUN

Combining patterns

- You can use the | operator to join patterns together

```rust
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
    Square(i32),
}

fn test_shape(shape: Shape) {
    match shape {
        Shape::Circle(size) | Shape::Square(size) => {
            println!("Shape has single size field {}", size);
        }
        _ => {
            println!("Not a circle, nor a square");
        }
    }
}
```

RUN

Shorthand: `if let` conditionals

- You can use `if let` if only one case is of interest.
- Still *pattern matching*

```
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
}

fn test_shape(shape: Shape) {
    if let Shape::Circle(radius) = shape {
        println!("Shape is a Circle with radius {}", radius);
    }
}
```

RUN

Shorthand: `let else` conditionals

- If you expect it to match, but want to handle the error...
- The `else` block must *diverge*

```
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
}

fn test_shape(shape: Shape) {
    let Shape::Circle(radius) = shape else {
        println!("I only like circles");
        return;
    };
    println!("Shape is a Circle with radius {}", radius);
}
```

RUN

Shorthand: `while let` conditionals

- Keep looping whilst the pattern still matches

```rust
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
}

fn main() {
    while let Shape::Circle(radius) = make_shape() {
        println!("got circle, radius {}", radius);
    }
}

fn make_shape() -> Shape {
    todo!()
}
```

RUN

Foreshadowing!

Two very important enums

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

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

RUN

We'll come back to them after we learn about error handling.

Exercise

`describe_shape` should print `{Shape} with {radius/base/height} {x/y}`.

It should also print `Large shape detected` if the shape has a dimension larger than 10.

```
enum Shape {
    Circle(i32),
    Rectangle(i32, i32),
    Triangle(i32, i32),
}

fn main() {
    let shapes = [
        Shape::Circle(5),
        Shape::Rectangle(8, 15),
        Shape::Triangle(12, 11),
    ];

    for shape in shapes {
        describe_shape(shape);
```

Solution

Possible solution:

```rust
fn describe_shape(shape: Shape) {
    match shape {
        // Handle Circle case
        Shape::Circle(radius) => {
            println!("Circle with radius {}", radius);
            if radius > 10 {
                println!("Large shape detected");
            }
        }

        // Handle Rectangle case
        Shape::Rectangle(width, height) => {
            println!("Rectangle with width {} and height {}", width, height);
            if width > 10 && height > 10 {
                println!("Large shape detected");
            }
```

RUN

Error Handling

There are no exceptions

Rust has two ways of indicating errors:

- Returning a value
- Panicking

Returning a value

```
fn parse_header(data: &str) -> bool {
    if !data.starts_with("HEADER: ") {
        return false;
    }

    true
}
```

RUN

It would be nice if we could return *data* as well as *ok*, *or error*...

Foretold enums strike back! 🤯

Remember these? They are *very important* in Rust.

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

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

RUN

I can’t find it

If you have an function where one outcome is "can't find it", we use `Option`:

```rust
fn parse_header(data: &str) -> Option<&str> {
    if !data.starts_with("HEADER: ") {
        return None;
    }
    Some(&data[8..])
}
```

RUN

That’s gone a bit wrong

When the result of a function is *either* **Ok**, or some **Error** value, we use `Result`:

```rust
enum MyError {
    BadHeader
}

// Need to describe both the Ok type and the Err type here:
fn parse_header(data: &str) -> Result<&str, MyError> {
    if !data.starts_with("HEADER: ") {
        return Err(MyError::BadHeader);
    }
    Ok(&data[8..])
}
```

RUN

Handling Results by hand

You can handle `Result` like any other enum:

```
use std::io::prelude::*;

fn read_file(filename: &str) -> Result<String, std::io::Error> {
    let mut file = match std::fs::File::open("data.txt") {
        Ok(f) => f,
        Err(e) => {
            return Err(e);
        }
    };
    let mut contents = String::new();
    if let Err(e) = file.read_to_string(&mut contents) {
        return Err(e);
    }
    Ok(contents)
}
```

RUN

Handling Results with ?

It is idiomatic Rust to use ? to handle errors.

```rust
use std::io::prelude::*;

fn read_file(filename: &str) -> Result<String, std::io::Error> {
    let mut file = std::fs::File::open("data.txt")?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents)
}
```

RUN

What kind of Error?

You can put anything in for the `E` in `Result<T, E>`:

```
fn literals() -> Result<(), &'static str> {
    Err("oh no")
}

fn strings() -> Result<(), String> {
    Err(String::from("oh no"))
}

fn enums() -> Result<(), Error> {
    Err(Error::BadThing)
}

enum Error { BadThing, OtherThing }
```

RUN

Using String Literals as the Err Type

Setting `E` to be `&'static str` lets you use `"String literals"`

- It's cheap
- It's expressive
- But you can't change the text to include some specific value
- And your program can't tell what *kind* of error it was

Using Strings as the Err Type

Setting `E` to be `String` lets you make up text at run-time:

- It's expressive
- You can render some values into the `String`
- But it costs you a heap allocation to store the bytes for the `String`
- And your program still can't tell what *kind* of error it was

Using enums as the Err Type

An `enum` is ideal to express *one* of a number of different *kinds* of thing:

```
/// Represents the ways this module can fail
enum Error {
    /// An error came from the underlying transport
    Io,
    /// During an arithmetic operation a result was produced that could not be stored
    NumericOverflow,
    /// etc
    DiskFull,
    /// etc
    NetworkTimeout,
}
```

RUN

Enum errors with extra context

An `enum` can also hold data for each variant:

```rust
/// Represents the ways this module can fail
enum Error {
    /// An error came from the underlying transport
    Io(std::io::Error),
    /// During an arithmetic operation a result was produced that could not
    /// be stored
    NumericOverflow,
    /// Ran out of disk space
    DiskFull,
    /// Remote system did not respond in time
    NetworkTimeout(std::time::Duration),
}
```

RUN

The std::error::Error trait

- The Standard Library has a `trait` that your `enum Error` should implement
- However, it's not easy to use
- Many people didn't bother
- See https://doc.rust-lang.org/std/error/trait.Error.html

Helper Crates

So, people created helper crates like `thiserror`

```rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum DataStoreError {
    #[error("data store disconnected")]
    Disconnect(#[from] io::Error),
    #[error("the data for key `{0}` is not available")]
    Redaction(String),
    #[error("invalid header (expected {expected:?}, found {found:?})")]
    InvalidHeader { expected: String, found: String },
    #[error("unknown data store error")]
    Unknown,
}
```

RUN

Something universal

Exhaustively listing all the ways your dependencies can fail is hard.

One solution:

```rust
fn main() -> Result<(), Box<dyn std::error::Error>> {
    let _f = std::fs::File::open("hello.txt")?; // IO Error
    let _s = std::str::from_utf8(&[0xFF, 0x65])?; // Unicode conversion error
    Ok(())
}
```

RUN

Anyhow

The anyhow crate gives you a nicer type:

```
fn main() -> Result<(), anyhow::Error> {
    let _f = std::fs::File::open("hello.txt")?; // IO Error
    let _s = std::str::from_utf8(&[0xFF, 0x65])?; // Unicode conversion error
    Ok(())
}
```

RUN

Panicking

The other way to handle errors is to generate a controlled, program-ending, failure.

- You can `panic!("x too large ({})", x);`
- You can call an API that panics on error (like indexing, e.g. `s[99]`)
- You can convert a `Result::Err` into a panic with `.unwrap()` or `.expect("Oh no")`

Excercise

Write a function that attempts to read and parse a configuration file. If the file is missing or its contents are invalid, the function should return an appropriate error.

```rust
enum ConfigError {
    FileNotFound(io::Error),
    InvalidFormat,
}

fn read_config(filename: &str) -> Result<i32, ConfigError> {
    // Try to read the file
    // useful functions: fs::read_to_string, Result::map_err
    let contents = !todo!("Read the contents of the file");

    // Try to parse the contents as an integer
    // useful functions: str::parse, Result::map_err
    let number = !todo!("Parse the contents as integer");

    Ok(number)
}
```

RUN

Solution

```rust
fn read_config(filename: &str) -> Result<i32, ConfigError> {
    // Try to read the file
    let contents = fs::read_to_string(filename)
        .map_err(|e| ConfigError::FileNotFound(e))?;

    // Try to parse the contents as an integer
    let number = contents.trim().parse::<i32>()
        .map_err(|_| ConfigError::InvalidFormat)?;

    Ok(number)
}
```

RUN

Methods and Traits

Methods

Methods

- Methods in Rust, are functions in an `impl` block
- They take `self` (or similar) as the first argument (the *method receiver*)
- They can be called with the *method call operator*

Example

```rust
struct Square(f64);

impl Square {
    fn area(&self) -> f64 { self.0 * self.0 }
    fn double(&mut self) { self.0 *= 2.0; }
    fn destroy(self) -> f64 { self.0 }
}

fn main() {
    let mut sq = Square(5.0);

    sq.double();  // Square::double(&mut sq)
    println!("area is {}", sq.area()); // Square::area(&sq)
    sq.destroy(); // Square::destroy(sq)
}
```

RUN

Method Receivers

- `&self` means `self: &Self`
- `&mut self` means `self: &mut Self`
- `self` means `self: Self`
- `Self` means whatever type this `impl` block is for

Method Receivers

- Other, fancier, *method receivers* are available!

```rust
struct Square(f64);

impl Square {
    fn by_value(self: Self) {}
    fn by_ref(self: &Self) {}
    fn by_ref_mut(self: &mut Self) {}
    fn by_box(self: Box<Self>) {}
    fn by_rc(self: Rc<Self>) {}
    fn by_arc(self: Arc<Self>) {}
    fn by_pin(self: Pin<&Self>) {}
    fn explicit_type(self: Arc<Example>) {}
    fn with_lifetime<'a>(self: &'a Self) {}
    fn nested<'a>(self: &mut &'a Arc<Rc<Box<Alias>>>) {}
    fn via_projection(self: <Example as Trait>::Output) {}
}
```

RUN

Associated Functions

- You can also just declare functions with no *method receiver*.
- You call these with normal *function call* syntax.
- Typically we provide a function called `new`

```rust
pub struct Square(f64);

impl Square {
    pub fn new(width: f64) -> Square {
        Square(width)
    }
}

fn main() {
    // Just an associated function - nothing special about `new`
    let sq = Square::new(5.0);
}
```

RUN

Associated Constants

`impl` blocks can also have `const` values:

```rust
pub struct Square(f64);

impl Square {
    const NUMBER_OF_SIDES: u8 = 4;

    pub fn perimeter(&self) -> f64 {
        self.0 * f64::from(Self::NUMBER_OF_SIDES)
    }
}
```

RUN

Traits

Traits

- A trait is a list of methods and functions that a type must have.
- A trait can provide *default* implementations if desired.

```rust
trait HasArea {
    /// Get the area, in m².
    fn area_m2(&self) -> f64;

    /// Get the area, in acres.
    fn area_acres(&self) -> f64 {
        self.area_m2() / 4046.86
    }
}
```

RUN

An example

```rust
trait HasArea {
    fn area_m2(&self) -> f64;
}

struct Square(f64);

impl HasArea for Square {
    fn area_m2(&self) -> f64 {
        self.0 * self.0
    }
}

fn main() {
    let sq = Square(5.0);
    println!("{}", sq.area_m2());
}
```

RUN

Associated Types

A trait can also have some *associated types*, which are type aliases chosen when the trait is *implemented*.

```
trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}

struct MyRange { start: u32, len: u32 }

impl Iterator for MyRange {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        todo!();
    }
}
```

RUN

Rules for Implementing

You can only *implement* a *Trait* for a *Type* if:

- The *Type* was declared in this module, or
- The *Trait* was declared in this module

You can't implement someone else's trait on someone else's type!

Rules for Using

You can only *use* the trait methods provided by a *Trait* on a *Type* if:

- The trait is in scope
- (e.g. you add `use Trait;` in that module)

Traits

- The standard library provides lots of traits, such as:
  - std::cmp::PartialEq and std::cmp::Eq
  - std::fmt::Debug and std::fmt::Display
  - std::iter::IntoIterator and std::iter::Iterator
  - std::convert::From and std::convert::Into

Sneaky Workarounds

If a trait method uses `&mut self` and you really want it to work on some `&SomeType` reference, you can:

```rust
impl SomeTrait for &SomeType {
    // ...
}
```

RUN

The I/O traits do this.

Using Traits Statically

- One way to use traits is by using `impl Trait` as a type.
- This is static-typing, and a new function is generated for every actual type passed.
  - Known as *monomorphisation*
- You can also `impl Trait` in the return position.

Using Traits Statically: Example

```
trait HasArea {
    fn area_m2(&self) -> f64;
}

struct AreaCalculator {
    area_m2: f64
}

impl AreaCalculator {
    // Multiple symbols may be generated by this function
    fn add(&mut self, shape: impl HasArea) {
        self.area_m2 += shape.area_m2();
    }

    fn total(&self) -> impl std::fmt::Display {
        self.area_m2
```

RUN

Using Traits Dynamically

- Rust also supports *trait references*
- The types are given at run-time through a *vtable*
- The reference is now a *wide pointer*

Using Traits Dynamically: Example

```rust
trait HasArea {
    fn area_m2(&self) -> f64;
}

struct AreaCalculator {
    area_m2: f64
}

impl AreaCalculator {
    // Only one symbol is generated by this function. The reference contains
    // a pointer to the table, *and* a pointer to a function table.
    fn add(&mut self, shape: &dyn HasArea) {
        self.area_m2 += shape.area_m2();
    }

    fn total(&self) -> &dyn std::fmt::Display {
```

RUN

Which is better?

Monomorphisation? Or *Polymorphism*?

Requiring other Traits

- Traits can also *require* other traits to also be implemented

```rust
trait Printable: std::fmt::Debug {
    fn print(&self) {
        println!("I am {:?}", self);
    }
}
```

RUN

Special Traits

- Some traits have no functions (`Copy`, `Send`, `Sync`, etc)
  - But code can require that the trait is implemented
  - More in this in generics!
- Traits can be marked `unsafe`
  - Must use the `unsafe` keyword to implement
  - They're telling you to read the instructions!

Exercise

Implement the following methods for `Circle`:

- `new(radius: f64) -> Self`: Creates a new `Circle` instance.
- `area(&self) -> f64`: Returns the area of the circle.
- `circumference(&self) -> f64`: Returns the circumference of the circle.

Impl `HasPerimeter` for `Circle`.

```
struct Circle(f64);

trait HasPerimeter {
    fn perimeter(&self) -> f64;
}
```

RUN

Solution

```
impl Circle {
    fn new(radius: f64) -> Self {
        Circle(radius)
    }

    fn area(&self) -> f64 {
        std::f64::consts::PI * self.0 * self.0
    }

    fn circumference(&self) -> f64 {
        2.0 * std::f64::consts::PI * self.0
    }
}

impl HasPerimeter for Circle {
    fn perimeter(&self) -> f64 {
```

RUN

Lifetimes

Rust Ownership

- Every piece of memory in Rust program has exactly one owner at the time
- Ownership changes ("moves")
  - `fn takes_ownership(data: Data)`
  - `fn producer() -> Data`
  - `let people = [paul, john, emma];`

Producing owned data

```
fn producer() -> String {
    String::new()
}
```

RUN

Producing references?

```
fn producer() -> &str {
    // ???
}
```

RUN

- `&str` "looks" at some string data. Where can this data come from?

Local Data

Does this work?

```
fn producer() -> &str {
    let s = String::new();
    &s
}
```

RUN

Local Data

No, we can't return a reference to local data...

```
error[E0515]: cannot return reference to local variable `s`
 --> src/lib.rs:3:5
  |
3 |     &s
  |     ^^ returns a reference to data owned by the current function
```

Local Data

You will also see:

```
error[E0106]: missing lifetime specifier
 --> src/lib.rs:1:18
  |
1 | fn producer() -> &str {
  |                  ^ expected named lifetime parameter
```

Static Data

```
fn producer() -> &'static str {
    "hello"
}
```

RUN

- bytes `h e l l o` are "baked" into your program
- part of *static* memory (not heap or stack)
- a slice pointing to these bytes will always be valid
- **safe** to return from `producer` function

Static Data

It doesn't have to be a string literal - any reference to a static is OK.

```
static HELLO: [u8; 5] = [0x68, 0x65, 0x6c, 0x6c, 0x6f];

fn producer() -> &'static str {
    std::str::from_utf8(&HELLO).unwrap()
}
```

RUN

`'static` annotation

- Rust never assumes `'static` for function returns or fields in types
- `&'static T` means this reference to `T` will never become invalid
- `T: 'static` means that "if type `T` has any references inside they should be `'static`"
  - `T` may have no references inside at all!
- string literals are always `&'static str`

```rust
fn takes_and_returns(s: &str) -> &str {

}
```

RUN

Where can the returned `&str` come from?

```rust
fn takes_and_returns(s: &str) -> &str {

}
```

RUN

Where can the returned `&str` come from?

- can't be local data

```rust
fn takes_and_returns(s: &str) -> &str {

}
```

RUN

Where can the returned `&str` come from?

- can't be local data
- is not marked as `'static`

```rust
fn takes_and_returns(s: &str) -> &str {

}
```

RUN

Where can the returned `&str` come from?

- can't be local data
- is not marked as `'static`
- **Conclusion: must come from `s`!**

Multiple sources

```
fn takes_many_and_returns(s1: &str, s2: &str) -> &str {

}
```

RUN

Where can the returned `&str` come from?

Multiple sources

```rust
fn takes_many_and_returns(s1: &str, s2: &str) -> &str {

}
```

RUN

Where can the returned `&str` come from?

- is not marked as `'static`

Multiple sources

```rust
fn takes_many_and_returns(s1: &str, s2: &str) -> &str {

}
```

RUN

Where can the returned `&str` come from?

- is not marked as `'static`
- should it be `s1` or `s2`?

Multiple sources

```rust
fn takes_many_and_returns(s1: &str, s2: &str) -> &str {

}
```

RUN

Where can the returned `&str` come from?

- is not marked as `'static`
- should it be `s1` or `s2`?
- **Ambiguous. Should ask programmer for help!**

Tag system

```
fn takes_many_and_returns<'a>(s1: &str, s2: &'a str) -> &'a str {

}
```

RUN

"Returned `&str` comes from `s2`"

`'a`

- "Lifetime annotation"
- often called "lifetime" for short, but that's a very bad term
  - every reference has a lifetime
  - annotation doesn't name a lifetime of a reference, but used to tie lifetimes of several references together
  - builds *"can't outlive"* and *"should stay valid for as long as"* relations
- arbitrary names: `'a`, `'b`, `'c`, `'whatever`

Lifetime annotations in action

```
fn first_three_of_each(s1: &str, s2: &str) -> (&str, &str) {
    (&s1[0..3], &s1[0..3])
}

fn main() {
    let amsterdam = format!("AMS Amsterdam");

    let (amsterdam_code, denver_code) = {
        let denver = format!("DEN Denver");
        first_three_of_each(&amsterdam, &denver)
    };

    println!("{} -> {}", amsterdam_code, denver_code);
}
```

RUN

Annotate!

```
fn first_three_of_each<'a, 'b>(s1: &'a str, s2: &'b str) -> (&'a str, &'b str) {
    (&s1[0..3], &s1[0..3])
}
```

RUN

Annotations are used to validate function body

"The source you used in code doesn't match the tags"

```
error: lifetime may not live long enough
 --> src/lib.rs:2:5
  |
1 | fn first_three_of_each<'a, 'b>(s1: &'a str, s2: &'b str) -> (&'a str, &'b str) {
  |                        --  -- lifetime `'b` defined here
  |                        |
  |                        lifetime `'a` defined here
2 |     (&s1[0..3], &s1[0..3])
  |     ^^^^^^^^^^^^^^^^^^^^^^ function was supposed to return data with lifetime `'b`
  |
  = help: consider adding the following bound: `'a: 'b`
```

Annotations are used to validate reference lifetimes at a call site

"Produced reference *can't outlive* the source"

```
error[E0597]: `denver` does not live long enough
  --> src/main.rs:10:41
   |
8  |     let (amsterdam_code, denver_code) = {
   |          -------------- borrow later used here
9  |         let denver = format!("DEN Denver");
   |             ------ binding `denver` declared here
10 |         first_three_of_each(&amsterdam, &denver)
   |                                         ^^^^^^^ borrowed value does not live long
11 |     };
   |     - `denver` dropped here while still borrowed

For more information about this error, try `rustc --explain E0597`.
```

Lifetime annotations help the compiler help you!

- You give Rust hints
- Rust checks memory access for correctness

```rust
fn first_three_of_each<'a, 'b>(s1: &'a str, s2: &'b str) -> (&'a str, &'b str) {
    (&s1[0..3], &s2[0..3])
}

fn main() {
    let amsterdam = format!("AMS Amsterdam");
    let denver = format!("DEN Denver");

    let (amsterdam_code, denver_code) = {
        first_three_of_each(&amsterdam, &denver)
    };

    println!("{} -> {}", amsterdam_code, denver_code);
}
```

What if multiple parameters can be sources?

```
fn pick_one(s1: &'? str, s2: &'? str) -> &'? str {
    if coin_flip() {
        s1
    } else {
        s2
    }
}
```

RUN

What if multiple parameters can be sources?

```
fn pick_one<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if coin_flip() {
        s1
    } else {
        s2
    }
}
```

RUN

- returned reference *can't outlive* either `s1` or `s2`
- potentially more restrictive

Example

```rust
fn coin_flip() -> bool { false }

fn pick_one<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if coin_flip() {
        s1
    } else {
        s2
    }
}

fn main() {
    let a = String::from("a");
    let b = "b";
    let result = pick_one(&a, b);
    // drop(a);
    println!("{}", result);
```

RUN

Lifetime annotations for types

```
struct Configuration {
    database_url: &str,
}
```

RUN

Where does the string data come from?

Generic lifetime parameter

```rust
struct Configuration<'a> {
    database_url: &'a str,
}
```

RUN

- An instance of `Configuration` *can't outlive* a string that it refers to via `database_url`.
- The string *can't be dropped* *while* an instance of `Configuration` *still* refers to it.

Exercise

```rust
struct LongestStr<'a> {
    value: &'a str,
    length: usize,
}

fn choose_longest(a: &str, b: &str) -> LongestStr {
    // TODO: Implement function logic to find the longest string
}
```

RUN

Solution

```rust
fn choose_longest<'a>(a: &'a str, b: &'a str) -> LongestStr<'a> {
    let longest = if a.len() > b.len() { a } else { b };
    LongestStr { value: longest, length: longest.len() }
}
```

RUN

Imports and Modules

Namespaces

- A namespace is simply a way to distinguish two things that have the same name.
- It provides a *scope* to the identifiers within it.

Rust supports namespacing in two ways:

1. Crates for re-usable software libraries
2. Modules for breaking up your crates

Crates

- A crate is the unit of Rust software suitable for shipping.
- Yes, it's a deliberate pun.
- The Rust Standard Library is a crate.
- Binary Crates and Library Crates

There’s no build file

- Have you noticed that `Cargo.toml` says nothing about which files to compile?
- Cargo starts with `lib.rs` for a library or the relevant `main.rs` for a binary
- It then finds all the *modules*

Modules

- A module is block of source code within a crate
- It qualifies the names of everything in it
- It has a parent module (or it is the crate root)
- It can have child modules
- The crate is therefore a *tree*

Standard Library

We've been using modules from the Rust Standard Library...

```rust
use std::fs;
use std::io::prelude::*;

fn main() -> std::io::Result<()> {
    let mut f = fs::File::create("hello.txt")?;
    f.write(b"hello")?;
    Ok(())
}
```

RUN

In-line modules

You can declare a module in-line:

```
mod animals {
    pub struct Cat { name: String }

    impl Cat {
        pub fn new(name: &str) -> Cat {
            Cat { name: name.to_owned() }
        }
    }
}

fn main() {
    let c = animals::Cat::new("Mittens");
    // let c = animals::Cat { name: "Mittens".to_string() };
}
```

RUN

Modules in a file

You can also put modules in their own file on disk.

This will load from either `./animals/mod.rs` or `./animals.rs`:

```
mod animals;

fn main() {
    let c = animals::Cat::new("Mittens");
    // let c = animals::Cat { name: "Mittens".to_string() };
}
```

RUN

Modules can be nested…

```
~/probe-run $ tree src
src
├── backtrace
│   ├── mod.rs
│   ├── pp.rs
│   ├── symbolicate.rs
│   └── unwind.rs
├── canary.rs
├── cli.rs
├── cortexm.rs
├── dep
│   ├── cratesio.rs
│   ├── mod.rs
│   ├── rust_repo.rs
│   ├── rust_std
│   │   └── toolchain.rs
│   │
```

What kind of import?

Choosing whether to import the parent module, or each of the types contained within, is something of an art form.

```
use std::fs;
use std::collections::VecDeque;
use std::io::prelude::*;
```

RUN

Standard Library

There's also a more compact syntax for imports.

```rust
use std::{fs, io::prelude::*};

fn main() -> std::io::Result<()> {
    let mut f = fs::File::create("hello.txt")?;
    f.write(b"hello")?;
    Ok(())
}
```

RUN

Exercise

Refactor the code by moving each struct and its associated methods into separate modules. Organize these modules under a root module and decide if any submodules are necessary. Update main.rs to import and use the refactored modules.

Solution

Consider the following structure:

```
src
├── main.rs
└── shapes
    ├── mod.rs
    ├── circle.rs
    ├── rectangle.rs
    └── triangle.rs
```

`src/shapes/mod.rs`:

```rust
pub mod circle;
pub mod rectangle;
pub mod triangle;

pub use circle::Circle;
pub use rectangle::Rectangle;
pub use triangle::Triangle;
```

RUN

That’s it!