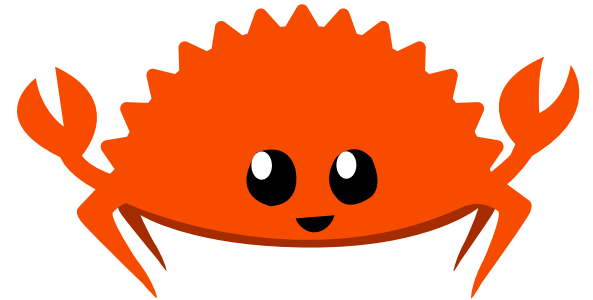


Safe Programming in Rust

Lecture 1 - Why Rust & Intro

Frans Skarman



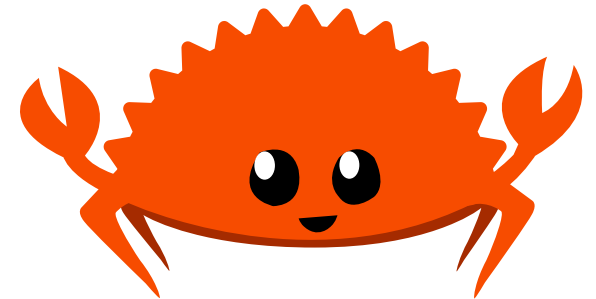
Who are we

Frans Skarman

- Rust user since 1.0
- HDL developer

Who are you?

- Name
- Why you're taking the course
- What is your go to programming language?
- If you were a sea creature, which sea creature would you be?



☒ New course alert ☒

☒ New course alert ☒

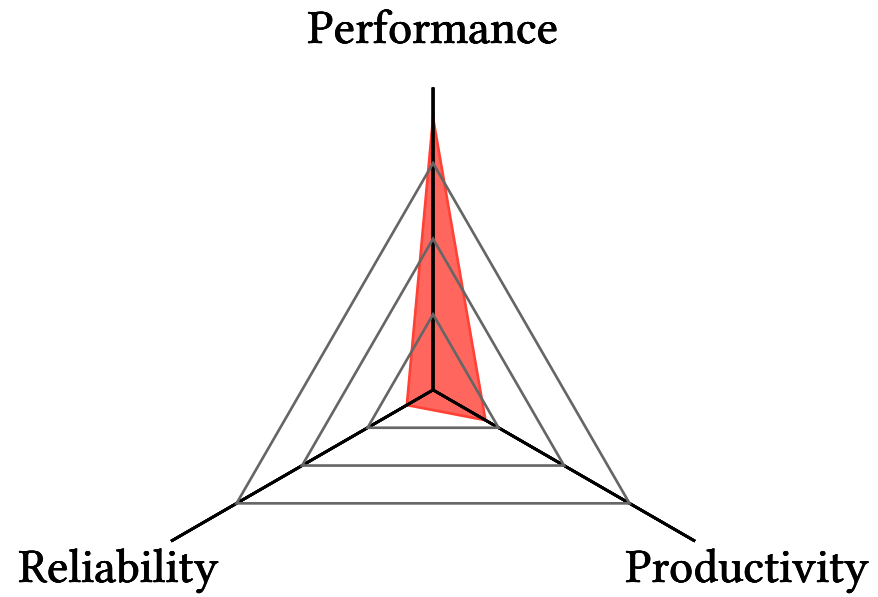
☒ New teacher alert ☒

☒ New course alert ☒

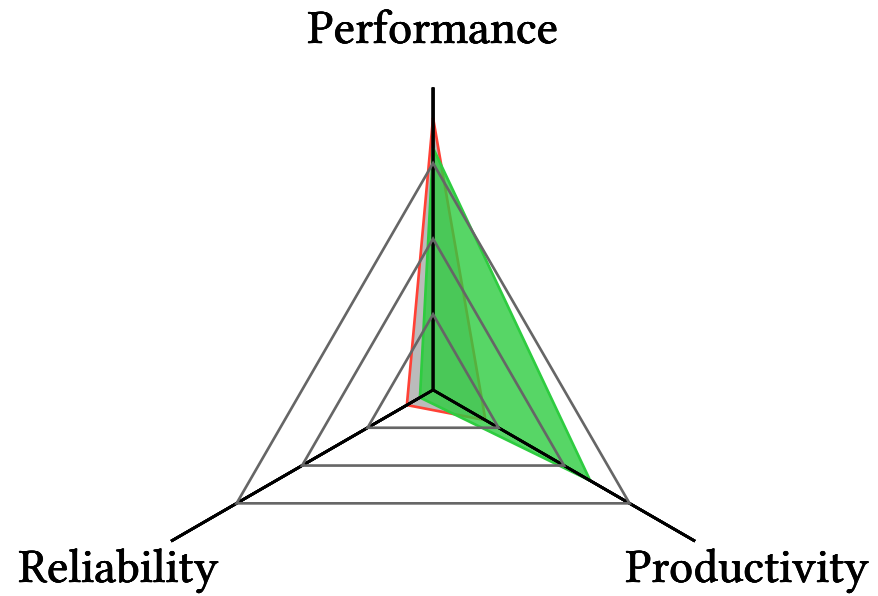
☒ New teacher alert ☒

Feedback is highly appreciated! :)

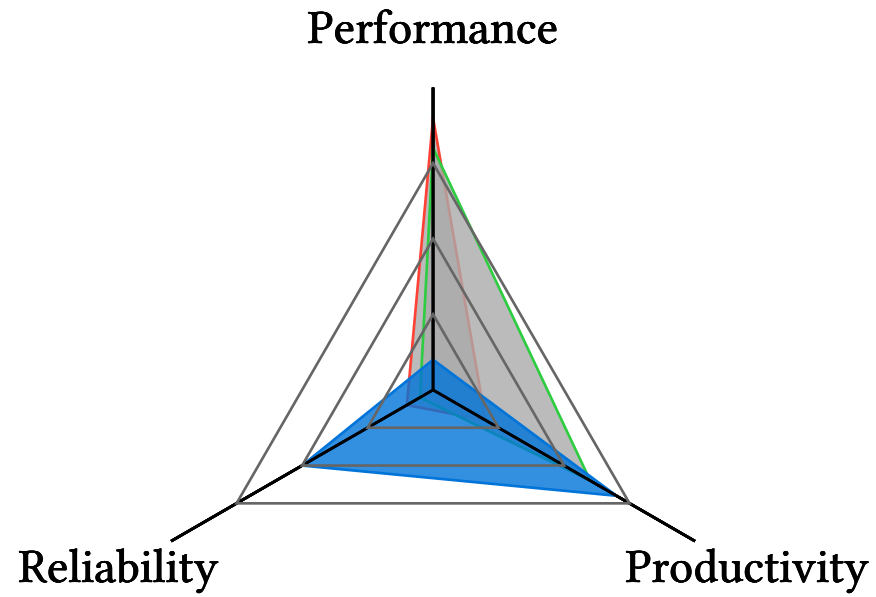
Why Rust



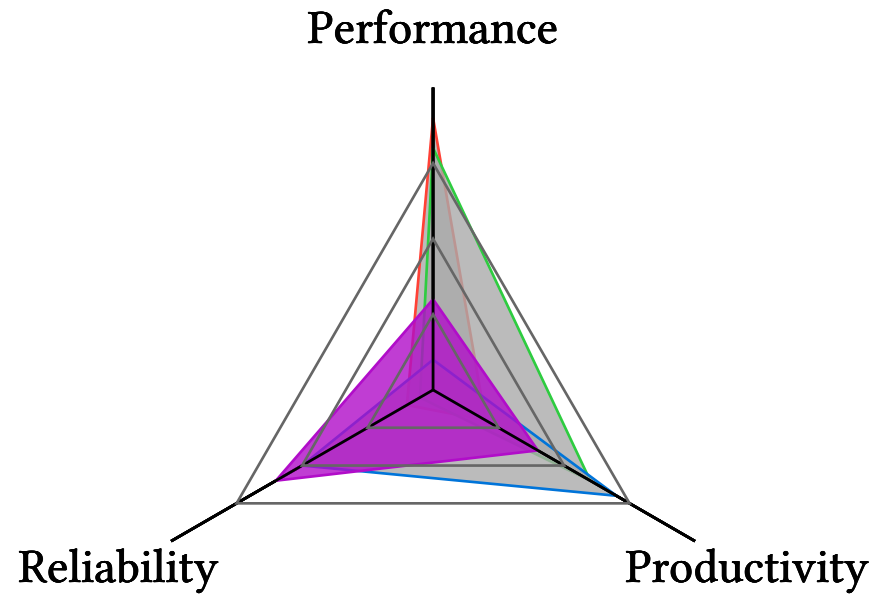
Why Rust



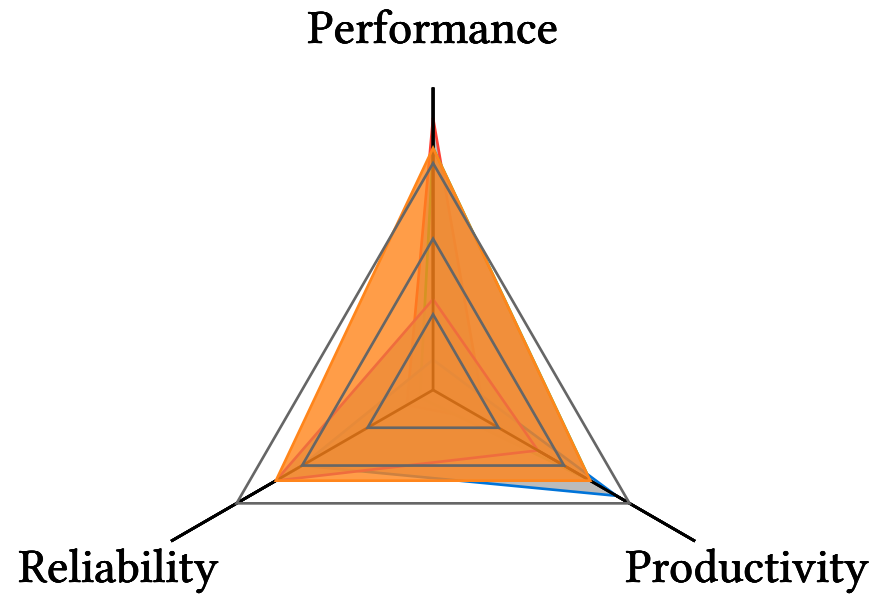
Why Rust



Why Rust



Why Rust



Fast, reliable, productive: pick three

Memory Safety

C/C++ have great performance and offer low level control

Memory Safety

C/C++ have great performance and offer low level control

With great power comes great responsibility

Memory Safety

C/C++ have great performance and offer low level control

With great power comes great responsibility

- Read/write past buffers
- Use after free
- Race conditions

Memory Safety

C/C++ have great performance and offer low level control

Garbage collection fixes this!

With great power comes great responsibility

- Read/write past buffers
- Use after free
- Race conditions

Memory Safety

C/C++ have great performance and offer low level control

With great power comes great responsibility

- Read/write past buffers
- Use after free
- Race conditions

Garbage collection fixes this!

But at the cost of performance and control.

Memory Safety

C/C++ have great performance and offer low level control

With great power comes great responsibility

- Read/write past buffers
- Use after free
- Race conditions

Garbage collection fixes this!

But at the cost of performance and control.

Rust does compile time analysis of memory use

Lecture Plan

1. (Today): Intro & Rust Basics
2. Ownership & Borrowing
3. Structs, Enums, Errors
4. Generics and Traits
5. Modules, Collections, Iterators
6. Lifetimes, Closures
7. Smart Pointers, Trait Objects
8. Parallelism, Shared Memory
9. Message Passing
10. Unsafe
11. Async

Labs

One session every week after the lectures

Not graded, **there for you to learn**

Final Project

(Some details still need working out)

Build something with Rust.

A new project or contribute to something existing, up to you!

Final Project

(Some details still need working out)

Build something with Rust.

A new project or contribute to something existing, up to you!

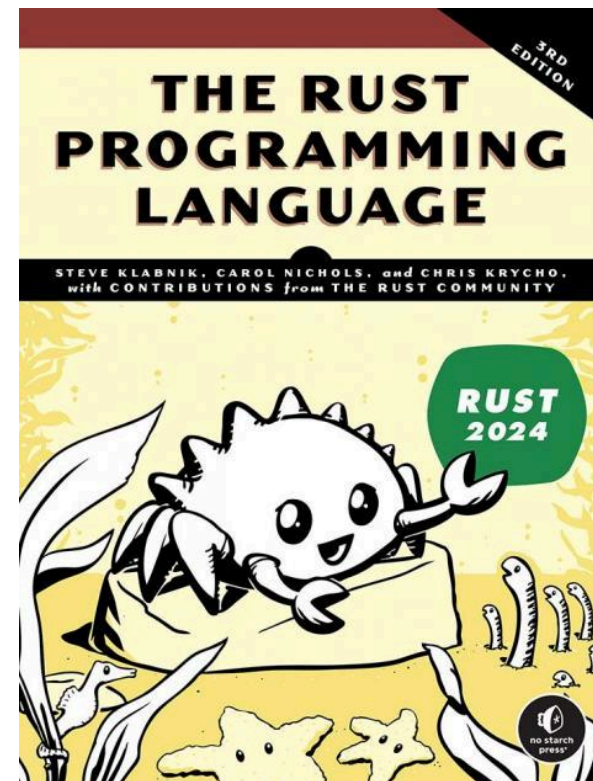
No LLMs for generating code. Asking questions is fine

Course literature

The Rust Programming Language

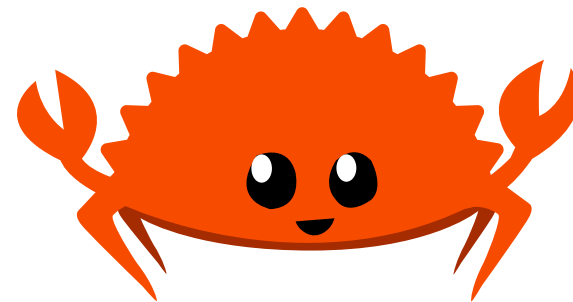
Steve Klabnik, Carol Nichols, and Chris Krycho, with contributions from the Rust Community

Available online¹ and physical form



¹<https://doc.rust-lang.org/stable/book/index.html>

Let's learn some Rust (finally)



Anatomy of a Rust Program

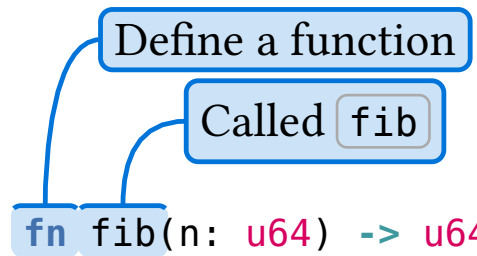
```
fn fib(n: u64) -> u64 {  
    if n <= 1 {  
        return 1;  
    } else {  
        let mut result = fib(n - 1);  
        result += fib(n - 2);  
        return result;  
    }  
}
```

Anatomy of a Rust Program

Define a function

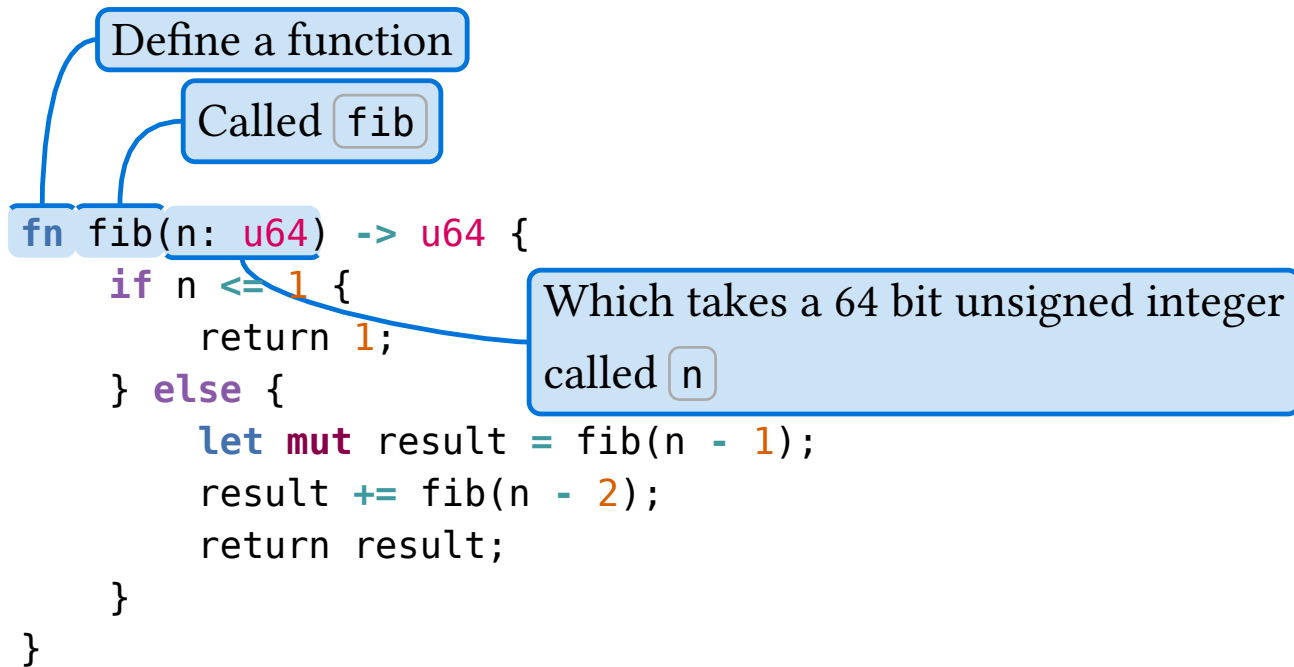
```
fn fib(n: u64) -> u64 {  
    if n <= 1 {  
        return 1;  
    } else {  
        let mut result = fib(n - 1);  
        result += fib(n - 2);  
        return result;  
    }  
}
```

Anatomy of a Rust Program

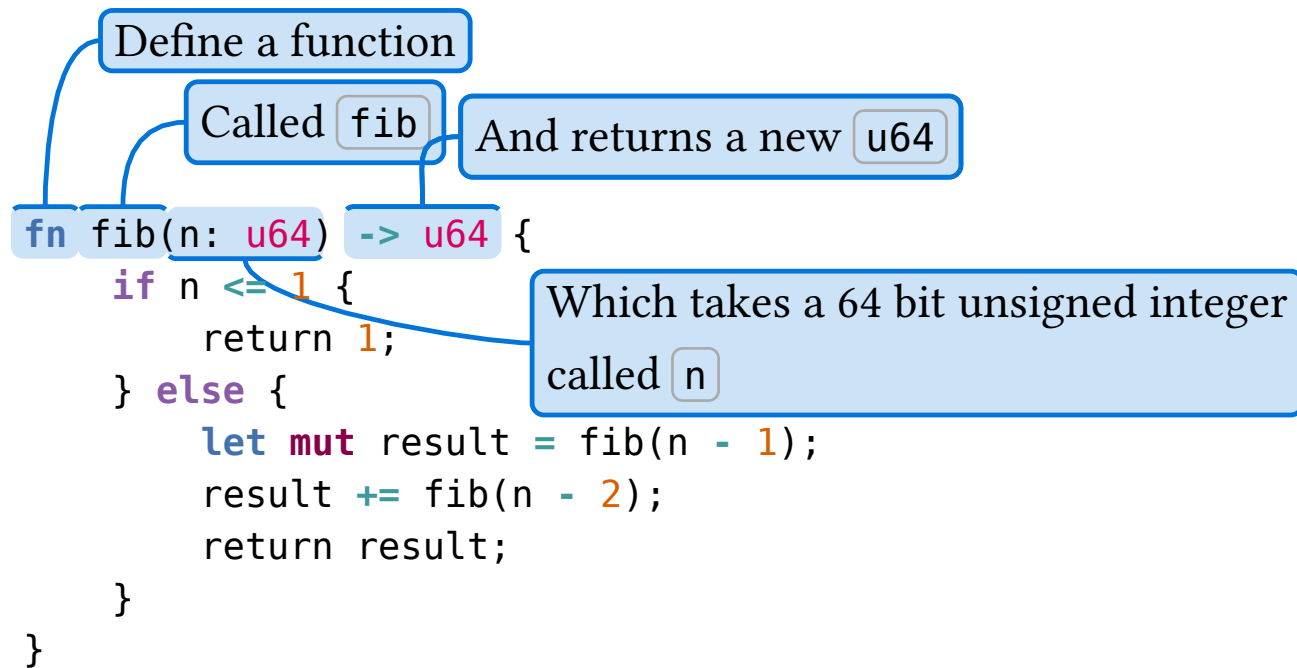


```
fn fib(n: u64) -> u64 {  
    if n <= 1 {  
        return 1;  
    } else {  
        let mut result = fib(n - 1);  
        result += fib(n - 2);  
        return result;  
    }  
}
```

Anatomy of a Rust Program



Anatomy of a Rust Program



Anatomy of a Rust Program

```
fn fib(n: u64) -> u64 {  
    if n <= 1 {  
        return 1;  
    } else {  
        let mut result = fib(n - 1);  
        result += fib(n - 2);  
        return result;  
    }  
}
```

Just a normal if statement

Anatomy of a Rust Program

```
fn fib(n: u64) -> u64 {  
    if n <= 1 {  
        return 1;  
    } else {  
        let mut result = fib(n - 1);  
        result += fib(n - 2);  
        return result;  
    }  
}
```

let defines a variable

Anatomy of a Rust Program

```
fn fib(n: u64) -> u64 {  
    if n <= 1 {  
        return 1;  
    } else {  
        let mut result = fib(n - 1);  
        result += fib(n - 2);  
        return result;  
    }  
}
```

Variables are immutable by default

`mut` marks them as mutable

~ The opposite of `const` from C/C++

Anatomy of a Rust Program

```
fn fib(n: u64) -> u64 {  
    if n <= 1 {  
        return 1;  
    } else {  
        let mut result = fib(n - 1);  
        result += fib(n - 2);  
        return result;  
    }  
}
```

Variables are immutable by default

`mut` marks them as mutable

~ The opposite of `const` from C/C++

This code works, but it isn't *Rusty* ☒

Try avoiding mutable variables

```
fn fib(n: u64) -> u64 {  
    if n <= 1 {  
        return n;  
    } else {  
        let result = fib(n-1) + fib(n-2);  
        return result;  
    }  
}
```

Try avoiding mutable variables

```
fn fib(n: u64) -> u64 {  
    if n <= 1 {  
        return n;  
    } else {  
        let result = fib(n-1) + fib(n-2);  
        return result;  
    }  
}
```

Easy change in this case



Try avoiding mutable variables

```
fn fib(n: u64) -> u64 {  
    if n <= 1 {  
        return n;  
    } else {  
        let result = fib(n-1) + fib(n-2);  
        return result;  
    }  
}
```

Easy change in this case

Still not Rusty enough ☒

Rust is Expression Based

```
let mut result = y;  
if cond {  
    result = x  
};
```

Rust is Expression Based

```
let mut result = y;  
if cond {  
    result = x  
};
```

```
let result = if cond {  
    x  
} else {  
    y  
};
```

Rust is Expression Based

```
let mut result = y;  
if cond {  
    result = x  
};
```

```
let result = if cond {  
    x  
} else {  
    y  
};
```

Can no longer forget to set `result`

More Rusty ☒!

Rust is Expression Based

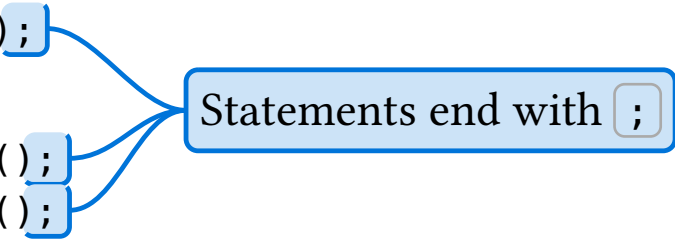
- Statements *do* something
- Expressions have a value
- Statements end with `;`

Rust is Expression Based

- Statements *do* something
- Expressions have a value
- Statements end with `;`

```
fn some_function() -> u32 {  
    let cond = foo();  
    if cond {  
        let a = bar();  
        let b = baz();  
        a + b  
    } else {  
        idk()  
    }  
}
```

Statements end with `;`

A light blue rounded rectangular callout box with a blue border contains the text "Statements end with ;". Three blue lines originate from the box and point to the semicolons at the end of the lines "let cond = foo();", "let a = bar();", and "let b = baz();".

Rust is Expression Based

- Statements *do* something
- Expressions have a value
- Statements end with `;`

```
fn some_function() -> u32 {
```

```
    let cond = foo();
```

```
    if cond {
```

```
        let a = bar();
```

```
        let b = baz();
```

```
        a + b
```

```
    } else {
```

```
        idk()
```

```
    }
```

```
}
```

Statements end with `;`

Expressions end without `;`

Rust is Expression Based

- Statements *do* something
- Expressions have a value
- Statements end with `;`

```
fn some_function() -> u32 {  
    let cond = foo();  
  
    if cond {  
        let a = bar();  
        let b = baz();  
        a + b  
    } else {  
        idk()  
    }  
}
```

Since they are the last expression
in the block, they are the results

Rust is Expression Based

- Statements *do* something
- Expressions have a value
- Statements end with `;`

```
fn some_function() -> u32 {  
    let cond = foo();
```

```
    if cond {  
        let a = bar();  
        let b = baz();  
        a + b  
    } else {  
        idk()  
    }  
}
```

The whole if expression is also an expression

```
}
```

Rust is Expression Based

- Statements *do* something
- Expressions have a value
- Statements end with `;`

```
fn some_function() -> u32 {  
    let cond = foo();  
    if cond {  
        let a = bar();  
        let b = baz();  
        a + b  
    } else {  
        idk()  
    }  
}
```

The whole if expression is also an expression

And it is the last expression in the function body.
So it is returned

Use an expression in arithmetic

Even Statements are Expressions

```
fn one() -> u64 {  
    1;  
}
```

Even Statements are Expressions

```
fn one() -> u64 {  
    1;  
}
```

error[E0308]: mismatched types

--> src/lib.rs:1:13

```
|  
1 | fn one() -> u64 {  
|   ---      ^^^ expected `u64`, found `()`  
|   |  
|   implicitly returns `()` as its body has no tail or `return` expression  
|  
2 |     1;  
|     - help: remove this semicolon to return this value
```

For more information about this error, try `rustc --explain E0308`.

error: could not compile `playground` (lib) due to 1 previous error

Even Statements are Expressions

```
fn one() -> u64 {  
    1;  
}
```

```
error[E0308]: mismatched  
--> src/lib.rs:1:13
```

```
1 | fn one() -> u64 {
```

```
    --- ^^^ expected u64 , found ()
```

```
    |  
    | implicitly returns `()` as its body has no tail or `return` expression
```

```
2 |     1;
```

```
    - help: remove this semicolon to return this value
```

`()` is the “unit type”. Kind of like `void`

For more information about this error, try ``rustc --explain E0308``.
error: could not compile `playground` (lib) due to 1 previous error

The Compiler is Your Friend

Compiler errors are generally very helpful

Don't think of it as **fighting** the compiler. Work with it

Rust is Expression Based

Even loops can “return”

```
let result = loop {  
    if cond {  
        break 5;  
    }  
    // ...  
};
```

Types

- Unsigned integers: `u8`, `u16`, `u32`, ...
- Signed integers: `i8`, `i16`, `i32`, ...
- Floats: `f32`, `f64`
- Bool: `bool`

Types

usize

The size of this primitive is how many bytes it takes to reference any location in memory. For example, on a 32 bit target, this is 4 bytes and on a 64 bit target, this is 8 bytes.

Types

usize

The size of this primitive is how many bytes it takes to reference any location in memory. For example, on a 32 bit target, this is 4 bytes and on a 64 bit target, this is 8 bytes.

Used for things like array indexing

Tuples

```
let x = (1, 2);  
let y: (u8, u32) = x;  
let e0 = x.0;  
let (e0, _) = x;
```

Tuples

Tuples are written like this

```
let x = (1, 2);  
let y: (u8, u32) = x;  
let e0 = x.0;  
let (e0, _) = x;
```

Tuples

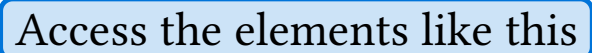
```
let x = (1, 2);  
let y: (u8, u32) = x;  
let e0 = x.0;  
let (e0, _) = x;
```

The type like this

Tuples

```
let x = (1, 2);  
let y: (u8, u32) = x;  
let e0 = x.0;  
let (e0, _) = x;
```

Access the elements like this



Tuples

```
let x = (1, 2);  
let y: (u8, u32) = x;  
let e0 = x.0;  
let (e0, _) = x;
```

Or the more rusty way:
destructuring with a pattern

Strong Typing

Implicit casts are almost never allowed

```
let x: u32 = 0;  
let y: u64 = x;
```

Strong Typing

Implicit casts are almost never allowed

```
let x: u32 = 0;  
let y: u64 = x;
```

```
error[E0308]: mismatched types  
--> src/lib.rs:3:14  
|  
3 | let y: u64 = x;  
|           ^ expected `u64`, found `u32`  
|           |  
|           expected due to this  
|  
help: you can convert a `u32` to a `u64`  
|  
3 | let y: u64 = x.into();  
|               +++++++
```

Strong Typing

Implicit casts are almost never allowed

```
let x: u32 = 0;  
let y: u64 = x;
```

use `.into()` or `as u64` to convert

No implicit casts?

```
let x = (1, 2);  
let y: (u8, u32) = x;
```

No implicit casts?

```
let x = (1, 2);  
let y: (u8, u32) = x;
```

Then what type do these integers have?

No implicit casts?

```
let x = (1, 2);  
let y: (u8, u32) = x;
```

It has to be *some* integer type
since they are integers. So

(integer, integer)

No implicit casts?

```
let x = (1, 2);  
let y: (u8, u32) = x;
```

It has to be *some* integer type
since they are integers. So

(integer, integer)

Here x is used where it has to be (u8, u32)

So it is inferred that x has to be (u8, u32)

Type Inference

- Most of the time, types can be inferred.
- If inference leads to ambiguity, you get an error
- Not dynamic types! It just saves some typing

Function inputs and outputs cannot be inferred

Rapid Fire Time

```
fn rapid_fire() {  
    let s = "foobar"; // string literals  
    let x = 1.0 + 2.0 / 3.0 * 4.0; // arithmetic  
    let b = true || false; // bools  
    let s: bool = 1 < 2; // explicit type annotations  
    let c = '🐱'; // unicode  
    let tup = (' ', "Ferris"); // tuples  
    while false { // while loops  
        println!("Uh-oh");  
    }  
    for i in 0..10 { // for-loops  
        println!("Loop iteration {i}") // printing  
    }  
    loop { // Explicitly infinite loops  
    }  
}
```