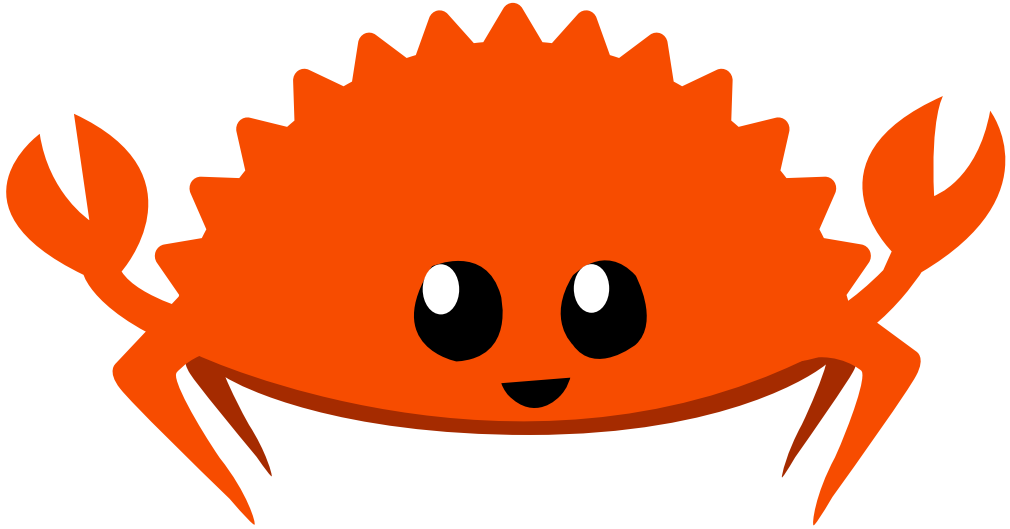
Intro to Rust Lang

Introduction

Welcome!

Meet Ferris!

- Ferris is Rust's mascot, and ours too!



Why Rust?

Why Rust?

- What is Rust?
- How does Rust compare to other languages?
- What are the biggest advantages of Rust?
- What are some issues that Rust has?
- Who is Rust for?

What is Rust?

- Rust started as a personal project of Graydon Hoare, a Mozilla Research employee, in 2006
- Mozilla sponsored the project in 2009, and released the source code in 2010
- The first stable release, Rust 1.0, was announced in May 2015
- From the official rust website, Rust is:
  - Fast
  - Reliable
  - Productive

What is Rust?

- Compiled language
- No runtime (no garbage collector)
- Imperative, but with functional features
- Strong static typing

Rust vs Python

- Significantly faster
- Much lower memory use
- Comprehensive type system
- Explicit error handling

Rust vs Java

- No runtime overhead from the JVM or a garbage collector
- Much lower memory use
- Zero-cost abstractions
- First-class support for modern paradigms

Rust vs C/C++

- **No segfaults!**
- **No null pointers!**
- **No buffer overflows!**
- **No data races!**
- Memory safety as a guarantee through the type checker
- Robust type system with functional patterns
- Unified build system and dependency management

Rust Is Memory Safe

- *"C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off"*
  - Bjarne Stroustrup, creator of C++
- Safety by default makes it much harder to shoot yourself in the foot
- Memory accesses checked at compile-time
- Powerful type system supports thread safety

Rust Is Low-Level

- Compiles to machine code
- No runtime (no garbage collector)
- Memory can be allocated deliberately
- Support for raw pointers
- Support for inline assembly
- Zero-overhead FFI

Rust Is Modern

- Rust is only 10 years old
- Algebraic and generic data types
- Modern developer tooling
- Included build system and dependency management
- Asynchronous execution as a first-class language feature
- Macros / Metaprogramming support

Issue: Learning curve

- Writing Rust *feels* very different
- The borrow checker can get in your way
- No object-oriented programming
- **That is what we are here for!**

Issue: Ecosystem

- Rust is only 10 years old...
- Smaller and less mature ecosystem compared to some older languages
- However, adoption among a wide range of companies is rapid!

Other Issues

- Compile time is slow
- Using established C++ libraries requires complicated bindings
- Programming in a systems language still takes more time than in a higher-level language

Who is Rust for?

- Rust targets complex programs while providing stability and security
- Rust is intended to be fast, reliable, and productive
- Rust is a language for building **Foundational Software**

Rust for Foundational Software

> Rust is a language for building foundational software.

- Niko Matsakis
  - Co-lead of the Rust language design team
  - Senior Principal Engineer at AWS

Foundational Software

What is Foundational Software?

- Not exactly "systems" software
- It is software that systems *rely* on
  - *Software that other systems are built on top of*
- Software that is:
  - Reliable and secure; **absolutely cannot fail**
  - Performant (zero-cost abstractions, fearless concurrency)
  - Stable (without stagnation)

Why Rust?

- Rust is an exceptional tool with many great features
- However, Rust is **not** a magic silver bullet
- In this course, we will teach you both the strengths and weaknesses of the Rust programming language

Course Goals

By the end of the semester, we want you all to:

- Be able to read, write, and reason about Rust code
- Become an intermediate to advanced Rust developer
- Be confident that you can use Rust going forward!

Cargo Basics

Hello World!

To create an executable, we need a `main` function:

src/main.rs

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

To compile `main.rs`, use `rustc`.

```
$ rustc main.rs
```

Cargo

Rust has a built-in build system and package manager called **Cargo**.

- **Build system**: Build and run in one command
- **Package manager:** Manages dependencies, like `pip` for `python` or `npm` for `node.js`

Creating a new project

To create a new cargo project called `hello_cargo`, use `cargo new`.

```
$ cargo new hello_cargo
$ cd hello_cargo
```

- You will find a few important things
  - `.git` repository and `.gitignore`
  - `Cargo.toml`
  - `src/main.rs`
- We will come back `Cargo.toml` in future weeks

Building your project

To build your project, use `cargo build`.

```
$ cargo build
   Compiling hello_cargo v0.1.0 (<path>/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 1.00s
```

- This creates an executable file at `target/debug/hello_cargo`
- What if we want to run this executable?
  - We could run `./target/debug/hello_cargo`, but this is a lot to type...

Running your project

To run your project, use `cargo run`.

```
$ cargo run
   Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/hello_cargo`
Hello, world!
```

Check if your project compiles

To check your code for syntax and type errors, use `cargo check`

```
$ cargo check
   Checking hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 0.42s
```

- Much faster than `cargo build` since it doesn't build the executable
- Useful when programming to check if your code still compiles

Cargo Recap

- We can create a project using `cargo new`
- We can build a project using `cargo build`
- We can build and run a project in one step using `cargo run`
- We can check a project for errors using `cargo check`
- Cargo stores our executable in the `target/debug` directory

Variables and Mutability

Variables

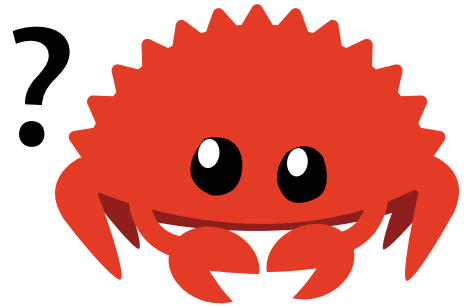
Variables are values bound to a name. We define variables with the `let` keyword.

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
}
```

Immutability

All variables in Rust are *immutable* by default.

```
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```



- What happens when we try to compile this?

Immutability

When we try to compile, we get this error message:

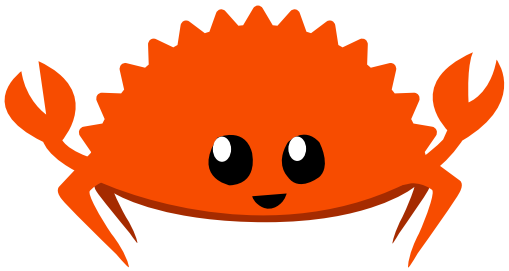
```
error[E0384]: cannot assign twice to immutable variable `x`
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: consider making this binding mutable: `mut x`
3 |     println!("The value of x is: {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
```

- Let's follow the compiler's advice!

Mutability

To declare a variable as mutable, we use the `mut` keyword.

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```



When we run the program now, we now get this:

```
$ cargo run
    <-- snip -->
The value of x is: 5
The value of x is: 6
```

Constants

Like immutable variables, constants are values bound to a name.

```
const THREE_HOURS_IN_SECONDS: u32 = 60 * 60 * 3;
```

- Constants cannot be `mut`
- Constants must have an explicit type
  - We will talk about types like `u32` in a few slides

Scopes and Shadowing

You can create nested scopes within functions with curly braces `{}`.

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {}", x);
    }

    println!("The value of x is: {}", x);
}
```

- Let's dissect this!

```
let x = 5;
let x = x + 1;
{
    let x = x * 2;
    println!("The value of x in the inner scope is: {}", x);
}
println!("The value of x is: {}", x);
```

- `x` is bound to `5` first
- A new variable `x` is created and bound to `x + 1`, i.e. `6`
- An inner scope is created with the opening curly brace `{`
- The third `let` statement shadows `x`
- The shadowed `x` is set to `x * 2 = 12`
- The inner scope ends with the closing curly brace `}`
- `x` returns to being 6 again

```
let x = 5;
let x = x + 1;
{
    let x = x * 2;
    println!("The value of x in the inner scope is: {}", x);
}
println!("The value of x is: {}", x);
```

Let's run this!

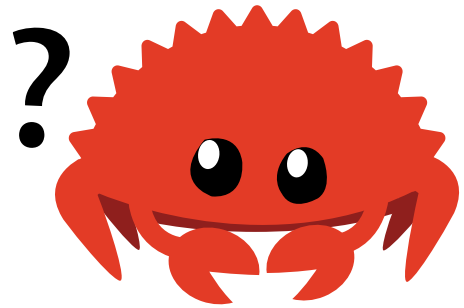
```
$ cargo run
    <-- snip -->
The value of x in the inner scope is: 12
The value of x is: 6
```

Aside: Shadowing vs Mutability

Mutability:

```
let mut spaces = "   ";
spaces = spaces.len();
```

```
  |
2 |     let mut spaces = "   ";
  |                      ----- expected due to this value
3 |     spaces = spaces.len();
  |              ^^^^^^^^^^^^ expected `&str`, found `usize`
  |
```

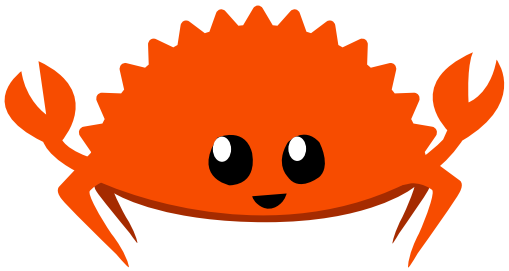


- Expected one *type*, got something else
  - We'll talk about types in a few slides!

Aside: Shadowing vs Mutability

Shadowing:

```
let spaces = "   ";
let spaces = spaces.len();
```



- Even though the types are different, the `let` keyword allows us to redefine the `spaces` variable

Shadowing vs Mutability

- Mutability lets us change the value of a variable
  - We get a compile time error if we try to modify a non-`mut` variable
- Shadowing allows us to change what a variable's name refers to
  - In addition to changing the value, it can also change types

Types

Types

Like most languages, there are two main categories of Data Types.

- Scalar Types
  - Integers
  - Floating-Points
  - Boolean
  - Character
- Compound Types
  - Tuples
  - Arrays

Integers

Rust has similar integer types you would expect to see in C.

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

Floating-Points

Rust has both a 32-bit and 64-bit floating-point type.

```
fn main() {
    let x = 2.0; // f64

    let y: f32 = 3.0; // f32
}
```

Numeric Operations

```
// addition
let sum = 5 + 10;

// subtraction
let difference = 95.5 - 4.3;

// multiplication
let product = 4 * 30;

// division
let quotient = 56.7 / 32.2;
let truncated = -5 / 3; // Results in -1

// remainder / modulo
let remainder = 43 % 5;
```

Integer Casting

Rust has no implicit type conversion (coercion). However, we can explicitly convert types using the `as` keyword.

```
fn main() {
    let decimal: f32 = 65.4321;

    let integer = decimal as u8;
    let character = integer as char;

    println!("{}, {}, {}", decimal, integer, character);
}
```

```
$ cargo run
    <-- snip -->
65.4321, 65, A
```

Booleans

A *boolean* in Rust has two values `true` and `false` (as in most other languages).

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}
```

- Booleans are always 1 byte in size

Characters

Rust has a UTF-32 character type `char`.

```
fn main() {
    let c = 'z';
    let z: char = 'ℤ'; // with explicit type annotation
    let heart_eyed_cat = '😍';
}
```

- Use `char` with single quotes (`'a'` vs. `"a"`)
- Due to `char` being UTF-32, a `char` is always **4 bytes in length**
- We will talk more about this and UTF-8 / UTF-32 in the future!

Tuples

A *tuple* is a way of grouping together a number of values with a variety of types.

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

Tuples

You can destructure tuples like so:

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
}
```

Tuples

You can also access specific elements in the tuples like so:

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

Arrays

To store a collection of multiple values, we use *arrays*.

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let months = ["January", "February", "March", "April", "May", "June", "July",
                  "August", "September", "October", "November", "December"];
}
```

- Unlike tuples, all elements must be the same type
- The number of elements is always fixed at compile time
    - If you want a collection that grows and shrinks, use a vector (lecture 4)
- Similar to stack-allocated arrays you would see in C

Arrays

We define an array's type be specifying the type of the elements and the length of the array.

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

We can also initialize the array such that every element has the same value.

```
let a = [3; 5];
// let a = [3, 3, 3, 3, 3];
```

Arrays

To access an array element, we use square brackets.

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

- Rust will ensure that the index is within bounds at runtime
  - This is *not* done in C/C++

Functions, Statements, and Expressions

Functions

Like most programming languages, Rust has functions!

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

fn another_function() {
    println!("Another function.");
}
```

```
$ cargo run
    <-- snip -->
Hello, world!
Another function.
```

Functions

All parameters / arguments to functions must be given an explicit type.

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {}{}", value, unit_label);
}
```

```
$ cargo run
    <-- snip -->
The measurement is: 5h
```

Returning from Functions

You can return values back to the caller of a function with the `return` keyword.

```
fn main() {
    let x = plus_one(5);
    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    return x + 1;
}
```

```
$ cargo run
    <-- snip -->
The value of x is: 6
```

Returning from Functions

You can also omit the `return` keyword.

```
fn plus_one(x: i32) -> i32 {
    x + 1
}
```

```
$ cargo run
    <-- snip -->
The value of x is: 6
```

- Why are we allowed do this?

Statements and Expressions

All functions are a series of statements optionally ending in an expression.

```
fn main() {
    let x = 6; // Statement
    let y = 2 + 2; // Statement resulting from the expression `2 + 2`
    2 + 2; // Expression ending in a semicolon, which turns the expression
           // into a statement with no effect
}
```

- **Statements** are instructions that do some action and don't return a value
- **Expressions** evaluate / return to a resultant value
- A more precise explanation can be found here

Statements and Expressions

- Statements
  - `let y = 6;` is a statement and does not return a value
  - You *cannot* write `x = y = 6`
- Expressions
  - `2 + 2` is an expression
  - Calling a function is an expression
  - A scope is also an expression
- If you add a semicolon to an expression, it turns into a statement
- If a scope is an expression, can scopes return values?

Statements and Expressions

Observe the following code, where a scope returns a value.

```
fn main() {
    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {}", y);
}
```

- Notice that there is no semicolon after `x + 1`
- Scopes return the value of their last expression
- Since functions are scopes, they can also return values in this way!

Function Return Types

Let's revisit this code snippet.

```
fn main() {
    let x = plus_one(5);
    println!("The value of x is: {}", x);
}

fn plus_one(x: i32) -> i32 {
    x + 1
}
```

- Functions must have a specific return value, or return nothing
  - No return type is equivalent to returning the unit type `()`
- Notice again that there is no semicolon after `x + 1`

Suppose we did add a semicolon:

```
fn plus_one(x: i32) -> i32 {
    x + 1;
}
```

We get this error:

```
error[E0308]: mismatched types
 --> src/main.rs:7:24
  |
1 | fn plus_one(x: i32) -> i32 {
  |    --------            ^^^ expected `i32`, found `()`
  |    |
  |    implicitly returns `()` as its body has no tail or `return` expression
2 |     x + 1;
  |          - help: remove this semicolon to return this value
```

Control Flow

`if` Expressions

We can define runtime control flow with `if`.

```
fn main() {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }
}
```

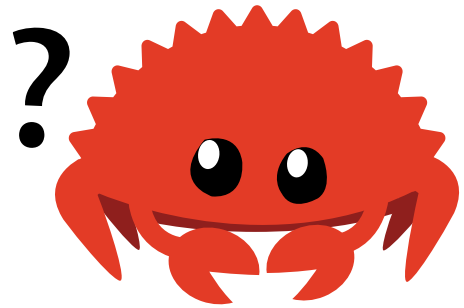
`if` Expressions

`if` expressions must condition on a boolean expression.

```
fn main() {
    let number = 3;

    if number {
        println!("number was three");
    }
}
```

```
error[E0308]: mismatched types
 --> src/main.rs:4:8
  |
4 |     if number {
  |        ^^^^^^ expected `bool`, found integer
```



`else if` Branching

You can handle multiple conditions with `else if`

```
fn main() {
    let number = 6;

    if number % 4 == 0 {
        println!("divisible by 4");
    } else if number % 3 == 0 {
        println!("divisible by 3");
    } else if number % 2 == 0 {
        println!("divisible by 2");
    } else {
        println!("not divisible by 4, 3, or 2");
    }
}
```

`if`s are Expressions!

Since `if` expressions are expressions, we can bind the result of an `if` expression to a variable.

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {}", number);
}
```

- `if` expressions must always return the same type in all branches

Loops

There are 3 kinds of loops in Rust.

- `loop`
- `while`
- `for`

`loop` loops

`loop` will loop forever until you tell it to stop with `break`.

```
fn main() {
    let mut counter = 0;

    loop {
        counter += 1;
        if counter == 10 {
            break;
        }
    }

    println!("The counter is {}", counter);
}
```

- `break` and `continue` apply to the innnermost loop where they are called

`loop`s are Expressions

Like everything else, you can return a value from a `loop`.

```
fn main() {
    let mut counter = 0;

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {}", result);
}
```

Loop Labels

You can label loops to use with `break` and `continue` to specify which loop it applies to.

```
'outer: loop {
    println!("Entered the outer loop");

    'inner: loop {
        println!("Entered the inner loop");

        // break; // <-- This would break only the inner loop
        break 'outer; // <-- This breaks the outer loop
    }

    println!("This point will never be reached");
}
println!("Exited the outer loop");
```

Loop Labels

```
'outer: loop {
    println!("Entered the outer loop");
    'inner: loop {
        println!("Entered the inner loop");
        break 'outer;
    }
    println!("This point will never be reached");
}
println!("Exited the outer loop");
```

```
Entered the outer loop
Entered the inner loop
Exited the outer loop
```

- Applies to `while` and `for` loops too

`while` loops

Just like other languages, we have `while` loops that stop after some condition.

```
fn main() {
    let mut number = 3;

    while number != 0 {
        println!("{}!", number);

        number -= 1;
    }

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

`for` loops

We can also loop through collections with a `for` loop.

```
fn main() {
    let a = [10, 20, 30, 40, 50];

    for element in a {
        println!("the value is: {}", element);
    }
}
```

`for` loops and ranges

To loop over a range, use the `..` syntax to create a range.

```
fn main() {
    for number in 1..4 {
        println!("{}...", number);
    }
    println!("SURPRISE!!!");
}
```

Recap

- Variables and Mutability
- Scalar and Compound Data Types
- Functions, Statements, and Expressions
- Control Flow

Course Logistics

Syllabus

You can find our course syllabus here or on our website.

- There is a quiz on Gradescope worth 50 points (half a homework) that checks that you have read the entire syllabus
- Please make sure you understand the **bolded** parts!

Course Logistics: Grading

- **Attendance is mandatory**
  - We have to take attendance every lecture
  - You get a maximum of 2 unexcused absences by StuCo guidelines
- Homeworks / programming assignments are worth 100 points each
- You need at least 1000 points to pass this course

Course Logistics: Communication

- Piazza
- Unofficial Discord
- Email
- Talk to us!

Course Logistics: Homework

- Homeworks are designed to take less than an hour per week
  - If you are spending more than that, please let us know!
- Autograded assignments through Gradescope
- 7 late days
  - *You can ask us for more late days if you ask in advance…*

Course Logistics: Homework Solutions

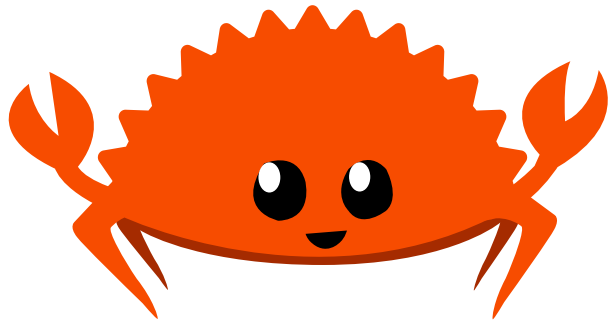
- We have made homework solutions **public**
- We **strongly encourage** students to avoid looking at these solutions
- Give a good-faith attempt at the homework before resorting to this
- You will not learn anything from copying and pasting our code
    - *We will catch you, and we are required to report you to the university*

Homework 1

- This first homework consists of 8 small puzzles and 4 simple function implementations
- The objective is to build confidence with Rust syntax and experience interpreting error messages
- Refer to `README.md` for further instructions
- Please let us know if you have any questions!

Next Lecture: Ownership (Part 1)

Thanks for coming!



Slides created by:

Connor Tsui, Benjamin Owad, David Rudo,
Jessica Ruan, Fiona Fisher, Terrance Chen