RUST

2024. 3

HOME › BABY BOY NAMES

Ferris

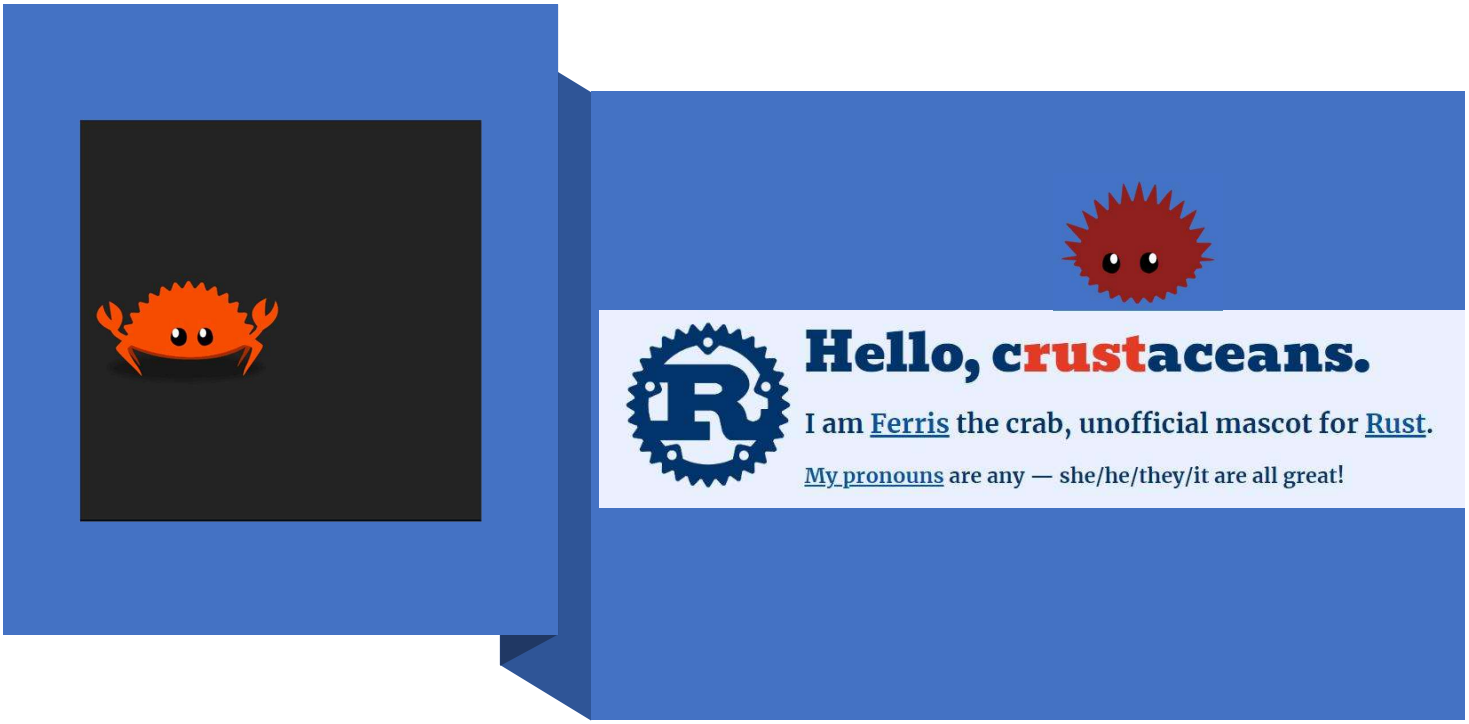
Irish, English | "strong man or ironworker"

↑ 4% this week

SHARE COPY LINK

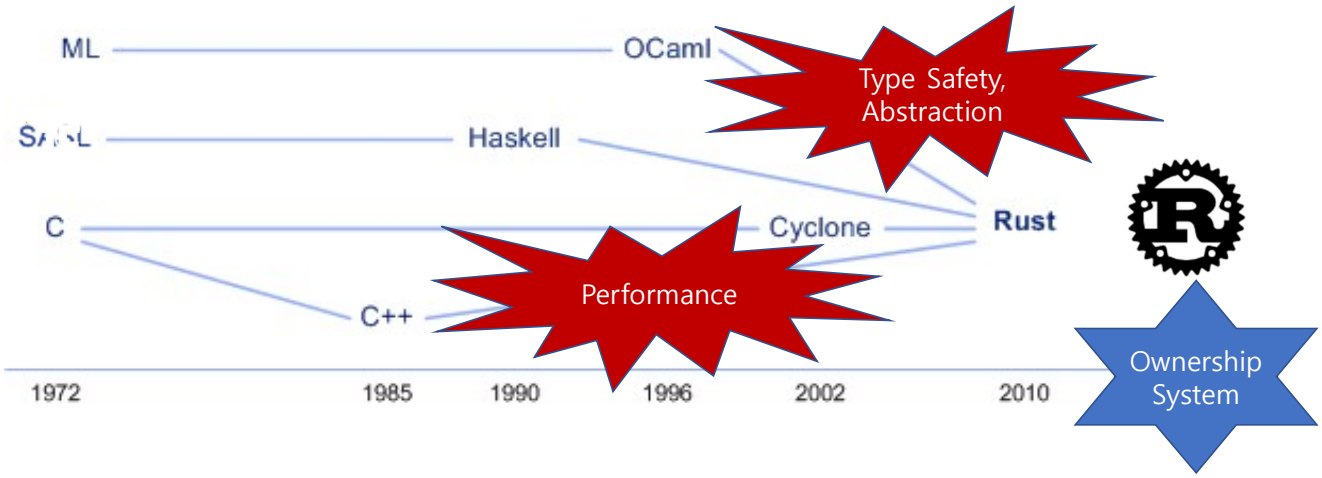
Ferris Origin and Meaning

The name Ferris is boy's name of Irish origin meaning "strong man or ironworker".


Hello, crustaceans.
I am Ferris the crab, unofficial mascot for Rust.
My pronouns are any — she/he/they/it are all great!


RUST
VS
C++

Genesis of Rust



Rust

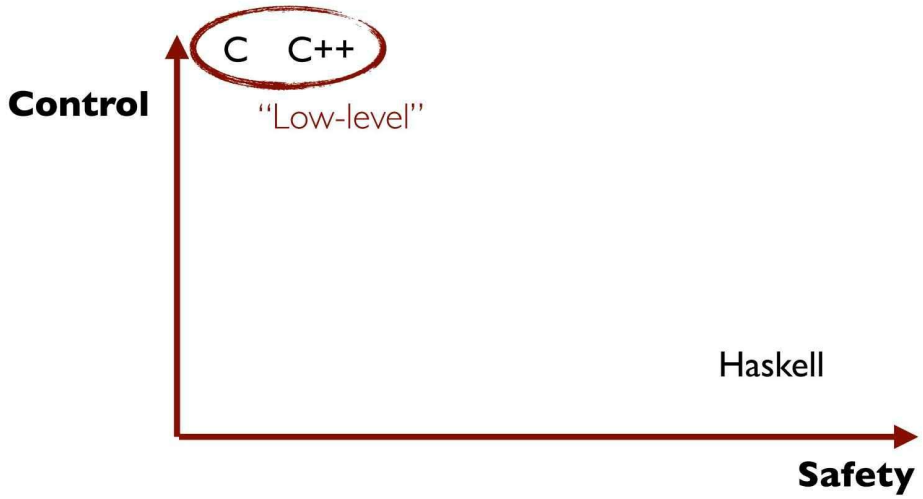
A language empowering everyone
to build reliable and efficient software.

GET STARTED

Version 1.77.0

Why Rust?

Performance	Reliability	Productivity
- No runtime - No garbage collection - Suited for embedded systems	- Rich type system - Compile-time checks - Ownership model - Guarantees memory safety and thread safety	- Good documentation - Good tooling - Friendly compiler and useful error messages



Rust is energy, time, and memory efficient.

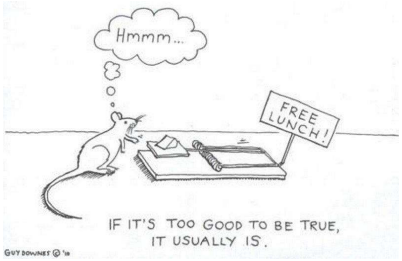
Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Zero Cost Abstractions

- In Rust, you can add abstractions without affecting runtime performance.
- Improve code quality and readability without any run-time performance cost.

What you don't use, you don't pay for. And further:
What you do use, you couldn't hand code any better.

No Free Lunch




```
error[E0382]: borrow of moved value
```

```
error[E0597]: `x` does not live long enough
```

To Fight or To Live Together with Compiler?

Course Contents

1. Basic language Features
2. Ownership Systems and Borrow Checker
3. Structs, Enums and Pattern Matching
4. Error handling Mechanism
5. Generics and Traits
6. Closures
7. Type Conversions
8. Iterators
9. Smart Pointers
10. Lifetimes
11. Unsafe Rust
12. Concurrency
13. Async Rust, FFI (extern and Interoperation with C) and Macros

Let's meet Rust!

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

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

macro

Rust doesn't care where you define your functions, only that they're defined somewhere in a scope that can be seen by the caller.

Rust function does not support variable length of arguments (such as **var_args**).

- Rust code uses *snake case* as the conventional style for function and variable names, in which all letters are lowercase and underscores separate words.

Variables

```
let hello: String = String::from("hello");

let x = 5; // type annotation can be omitted
           // if it can be inferred

let x; // uninitialized variable must be set
       // before its use
```

In Rust, *by default*, variables are *immutable*

- By default, variables are immutable for safefy and easy concurrency.
- For immutable variables, once a value is bound, you can't change that value.

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

```
2 |   let x = 5;
  |       -
  |       |
  |       first assignment to `x`
  |       help: consider making this binding mutable: `mut x`
4 |     x = 6;  // cannot assign twice to an immutable variable
  |     ^^^^^ cannot assign twice to immutable variable
```


Mutability

- Introduce mutability with the `mut` keyword:

```
let x = 5;
x = 6; // error!
```

```
let mut x = 5;
x = 6; // no problem!
```

mutable variable binding

- When a binding is mutable, it means you're allowed to change the bounded value through the variable.
  - *The binding of x changed from one i32 to another (i.e., from 5 to 6).*
  - *But, you may think that the content of the variable can be changed through the variable if the variable is not a reference. Will talk later more …*

Constants

SCREAMING_SNAKE_CASE
Pros: Can demonstrate your anger with text.
Cons: Makes your eyes deaf.
LOOK_AT_THIS, LOOK_AT_THAT, LOOK_HERE_YOU_MORON, …

all capital letters

type annotation is mandotory

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

- Constants are values that are bound to a name and are not allowed to change.
- You are **not** allowed to use `mut` with constants.
  - Constants are always immutable.
- Constants can be declared in any scope, including the global scope.
- Constants may be set only to a ***constant expression***, not the result of a value that could only be computed at runtime.

Shadowing

- If you declare a new variable with the same name, the first variable is *shadowed* by the second.

```
fn main() {
    let x = 5;
    let x = x + 1;  // shadowing also occurs here at the same scope
    {
        let x = x * 2; // shadowing occurs here in inner scope
        println!("The value of x in the inner scope is: {x}");
    }
    println!("The value of x is: {x}");
}
```

- When shadowing, **mutability and types can be changed**.

```
let spaces: &str = "   ";
let mut spaces: usize = spaces.len();
```

Data Types

Rust is a *statically typed language*, which means that compiler must know the types of all variables at ***compile time***.

- Scalar types
  - A scalar type represents a single value.
  - Four primary types: integers, floating-point numbers, booleans, and characters
- Compound types
  - Compound types can group multiple values into one type.
  - Two primitive compound types: **array** and **tuple**.

Primitive Types

- `bool`: {`true`, `false`}, 1 byte
- `i8`, `i16`, `i32`, `i64`, `i128`: signed integers { `i32::MIN` .. `i32::MAX` }
- `u8`, `u16`, `u32`, `u64`, `u128`: unsigned integers, { `u32::MIN` .. `u32::MAX` }
- `f32`, `f64`: floating-point numbers, { `f32::MIN` .. `f32::MAX` }
- `isize`: pointer-sized signed integer type (either 4 or 8 bytes)
- `usize`: pointer-sized unsigned integer type (either 4 or 8 bytes)
- `char`: *Unicode Scalar Value*, 4 bytes
- `str`: string slices, stored as a sequence of bytes encoded with UTF-8

Integer Literals

Number literals	Example
Decimal	`98_222`
Hex	`0xff`
Octal	`0o77`
Binary	`0b1111_0000`
Byte (`u8` only)	`b'A'`

- Integer types default to i32.
- The primary situation in which you'd use isize or usize is when *indexing* some sort of collection.

`char` type and `str` type

char Type

- Represents a single character like letter, number, emoji, etc.
- Unicode scalar value
- 4 bytes in size

Any Unicode scalar.

str Type

- A sequence of u8-array of unknown length guaranteed to hold UTF-8 encoded code points.

Rarely seen alone, but as `&str` instead.

The Tuple Type

- A tuple is a general way of grouping together a number of values with a variety of types into one compound type.
- Tuples have a *fixed length*: once declared, they cannot grow or shrink in size.

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
    let tup = (500, 6.4, 1);
    let (x, y, z) = tup;  // destructuring

    let first = tup.0;
    let second = tup.1;
    let third = tup.2;
}
```


The Array Type

- In Rust, arrays must be *homogeneous* and have a *fixed length*.

```
let a = [1, 2, 3, 4, 5];  // a.len() == 5
let a: [i32; 5] = [1, 2, 3, 4, 5];
let a = [3; 5]; // [3, 3, 3, 3, 3]

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

- Rust will check that the index is less than the array length at runtime, and panics otherwise.

```
let val = a[100];  // panicked at 'index out of bounds'
```

Functions with Parameters

- In function signatures, you must declare the type of each parameter.

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

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


Statements and Expressions

- **Statements** are instructions that perform some action and do not return a value.
  - Function definition itself is a statement.
  - Creating a variable and assigning a value to it with the `let` keyword is a statement. `let y = 6;`
  - Everything that ends with a semicolon (`;`) is a statement.
- **Expressions** evaluate to a resultant value.
  - Calling a function
  - Calling a macro.
  - A new scope block created with curly brackets (`{}`)
- *Rust is an expression-based language.*

```
let y = {
    let x = 3;
    x + 1
};
```

Unlike C, the following code is illegal

- In C:

```
int x, y;
x = y = 6;
```

expression

- In Rust:

```
let x = (let y = 6);
```

Error: Statement, not expression

Functions with Return Values

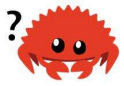
- In Rust, return type must be specified unless it is unit type (`()`).
- The return value of the function is *the value of the final expression* in the body of a function.
- You can return early by using the `return` keyword and specifying a value, but most functions return the last expression implicitly.

```
fn five() -> i32 {
    5  // return 5;
}

fn main() { // fn main() -> () {
    let x = five();
    println!("The value of x is: {x}");
}
```

If return type is unit (()), it can be omitted.

What's Wrong?

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

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



Conditional Branch

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

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

if is an expression

- The conditions must be a `bool`. If the condition isn't a `bool`, we'll get an error.

```
let number = 3;

if number { // error: expected `bool`, found integer
    println!("number was three");
}
```

- Because `if` is an *expression*, we can use it on the right side of a `let` statement.

```
let condition = true;
let number = if condition { 5 } else { 6 };

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


Unconditional loops with `loop`

- Repetition with loop

```
fn main() {
    loop {
        println!("again!");
    }
}
```

- Returning values from Loops

```
let mut counter = 0;

let result = loop {
    counter += 1;

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

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

Loop Labels to Disambiguate Between Nested Loops

```
    let mut count = 0;
'counting_up: loop {
    println!("count = {count}");
    let mut remaining = 10;

    loop {
        println!("remaining = {remaining}");
        if remaining == 9 { break; }
        if count == 2 {
            break 'counting_up;
        }
        remaining -= 1;
    }

    count += 1;
}
println!("End count = {count}");
```


Conditional loops with `while`

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

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

        number -= 1;
    }

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

Looping Through a Range/Collection with `for`

```
let numbers = [10, 20, 30, 40, 50];
let mut index = 0;

while index < 5 {
    println!("the value is: {}", numbers[index]);
    index += 1;
}
```

```
let numbers = [10, 20, 30, 40, 50];

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


More on for-loop

```
// Loop over array
let numbers = [1, 2, 3, 5];
for i: i32 in numbers {
    println!("{}", i);
}

// for-loop with step
for number in (1..10).step_by(2) {
    println!("{number}");
}

// Countdown for-loop
for number in (1..4).rev() {
    println!("{number}!");
}
```

```
// Loop over vector
let vs: Vec<i32> = vec![1, 2, 3, 4, 5];
for e: i32 in vs {
    println!("{e}");
}

let v = vec!['a', 'b', 'c', 'd'];

for (i: usize, ch: &i32) in
  v.iter().enumerate() {
    println!("{i}: {ch}");
}
// 0: a
// 1: b
// 2: c
// 3: d
```

More on for-loop and iterators later!