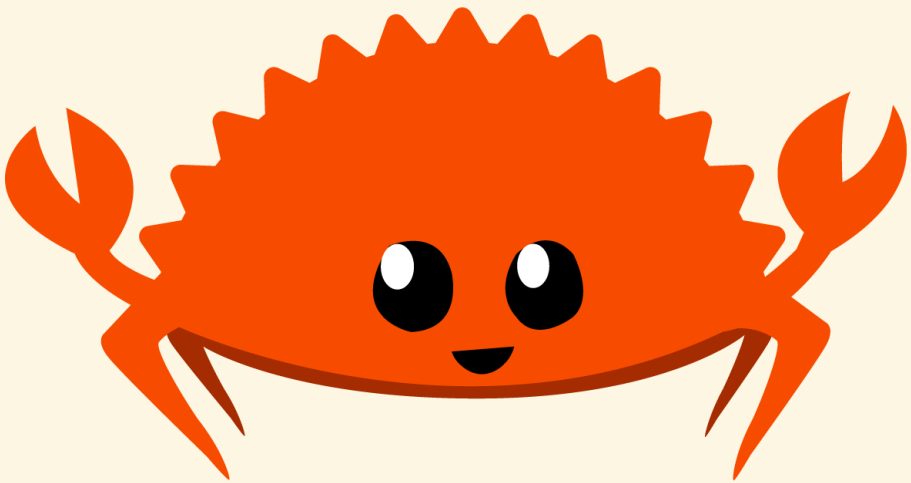
AN INTRODUCTORY TOUR OF RUST



Richard Gibson | @rickityg | Hexlabs.io

RUST: A LOVED LANGUAGE

Most Loved, Dreaded, and Wanted Languages

For five years running, Rust has taken the top spot as the most loved programming language. TypeScript is second surpassing Python compared to last year. We also see big gains in Go, moving up to 5th from 10th last year.

VBA, Objective C, and Perl hold the top spots for the most dreaded languages—languages that had a high percentage of developers who are currently using them, but have no interest in continuing to do so.

If we look at technologies that developers report that they do not use but want to learn, Python takes the top spot for the fourth year in a row. We also see some modest gains in the interest in learning Rust.

Loved Dreaded Wanted

% of developers who are developing with the language or technology and have expressed interest in continuing to develop with it

ABOUT RUST

- Primarily a systems programming language
- Started at Mozilla
- Used extensively at Amazon, Google & Microsoft
- Used in Firefox, Deno and Firecracker

ABOUT RUST

- Compiled language
- Runs natively without runtime
- No Garbage Collection
- Performance on par with C
- Strong static typing
- Imperative with FP & OOP Features

ABOUT RUST

Compiler/Type system ensures

- Incorrect Memory (De)allocation
- No surprises with types
  - (no null or thrown exceptions)
- No race conditions across threads

RUST COMMUNITY

- Rust foundation
  - community driven with industry support
- Open RFC process
- Documentation encouraged
- Great free learning resources
- Inclusive culture

RUST TOOLING

- cargo: build and dependencies
- clippy: linter
- rustfmt: code formatter
- rustdoc: documentation
- excellent editor support
- compiler with sane error messages!!!

SETUP

Linux or MacOS

```
$ curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

```
Rust is installed now. Great!
```

Other instructions available in Rust book

CARGO

Default build tool installed with Rust

```
cargo new dev_bash --bin # --bin for program, --lib for librar

cd dev_bash/
cargo build # fetches dependencies, uses profile `dev`
Compiling dev_bash v0.1.0 (.../dev_bash)
Finished dev [unoptimized + debuginfo] target(s) in 0.75s

./target/debug/dev_bash
Hello, world!

cargo run
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running `target/debug/dev_bash`
Hello, world!
```

CARGO

Default build tool installed with Rust

```
cargo new dev_bash --bin # --bin for program, --lib for librar

cd dev_bash/
cargo build # fetches dependencies, uses profile `dev`
Compiling dev_bash v0.1.0 (.../dev_bash)
Finished dev [unoptimized + debuginfo] target(s) in 0.75s

./target/debug/dev_bash
Hello, world!

cargo run
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running `target/debug/dev_bash`
Hello, world!
```

CARGO

Default build tool installed with Rust

```
cargo new dev_bash --bin # --bin for program, --lib for librar

cd dev_bash/
cargo build # fetches dependencies, uses profile `dev`
Compiling dev_bash v0.1.0 (.../dev_bash)
Finished dev [unoptimized + debuginfo] target(s) in 0.75s

./target/debug/dev_bash
Hello, world!

cargo run
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running `target/debug/dev_bash`
Hello, world!
```

CARGO

Default build tool installed with Rust

```
cargo new dev_bash --bin # --bin for program, --lib for librar

cd dev_bash/
cargo build # fetches dependencies, uses profile `dev`
Compiling dev_bash v0.1.0 (.../dev_bash)
Finished dev [unoptimized + debuginfo] target(s) in 0.75s

./target/debug/dev_bash
Hello, world!

cargo run
Finished dev [unoptimized + debuginfo] target(s) in 0.00s
Running `target/debug/dev_bash`
Hello, world!
```

OTHER USEFUL CARGO COMMANDS

`cargo test` - Compile and execute tests

`cargo fetch` - Fetch dependencies of a package

`cargo tree` - Display dependency graph

`cargo search` - Search packages in crates.io

`cargo fmt` - Format code files using rustfmt

`cargo help` - explain cargo command

CARGO PACKAGE MANAGEMENT

- Cargo dependency management
- Binarys available from a central registry
- Code can be built from git repo or local path
- RustDoc comments become documentation for crate on registry

RUST LANGUAGE BASICS

VARIABLES

Optional explicit types otherwise implicit

```
let i = 1000;
let j: i32 = 1000;
```

```
let box_i = Box::new(1);
let z = *b + 1;
```

DEFAULT IMMUTABILITY

```
    let mut i = 1000;
    let j = 1000;
    i +=1;
    j +=1;
```

```
error[E0384]: cannot assign twice to immutable variable `j`
  --> src/main.rs:11:5
   |
9  |     let j = 1000;
   |         -
   |         |
   |         first assignment to `j`
   |         help: make this binding mutable: `mut j`
10 |     i +=1;
11 |     j +=1;
   |     ^^^^^ cannot assign twice to immutable variable
```

DEFAULT IMMUTABLE STRUCTURES

```
    let mut mut_vec = Vec::new();
    mut_vec.push("some value");

    let immut_vec = Vec::new();
    immut_vec.push("another value");
```

```
error[E0596]: cannot borrow `immut_vec` as mutable, as it is not
  --> src/main.rs:15:5
   |
14 |     let immut_vec = Vec::new();
   |         --------- help: consider changing this to be mutable
15 |     immut_vec.push("another value");
   |     ^^^^^^^^^ cannot borrow as mutable
```

FUNCTION SYNTAX

```
fn square_then_add(i: i32, j: i32) -> i32 {
    let i_sq = i * i;
    let j_sq = j * j;
    i_sq + j_sq
}
```

```
fn square_then_add_then_print(i: i32, j: i32) {
    let i_sq = i * i;
    let j_sq = j * j;
    println!("square & sum of ({},{}) is {}", i, j, i_sq + j_sq);
}
```

OWNERSHIP: RUSTS MEMORY MANAGEMENT

Garbage (memory no longer used) must be cleaned up

- Explicitly through code
- Implicitly through a garbage collector

OWNERSHIP: RUSTS MEMORY MANAGEMENT

Rust enforces memory safety at Compile time through ownership

- Memory is cleaned up as a reference goes out of scope
- Strict rules on references to ensure scoping is correct

Same technique used for files and connections

OWNERSHIP: EXAMPLE

```rust
    // s owner of string
    let s = String::from("hello");
    // transfer ownership to y
    let y = s;
    // data no longer held at s
    println!("{}, world!", s);
```

```
error[E0382]: borrow of moved value: `s`
  --> src/main.rs:43:28
   |
41 |     let s = String::from("hello");
   |         - move occurs because `s` has type `String`, which d
42 |     let _y = s;
   |              - value moved here
43 |     println!("{}, world!", s);
   |                            ^ value borrowed here after move
```

OWNERSHIP: RULES

- One owner of each piece of data
- Data cleaned up when owner goes out of scope
- Owner can transfer ownership to a piece of data or lend it out

OWNERSHIP: EXAMPLE FIXED

```
let s = String::from("hello");
//give y a read only reference of s
let y = &s;
println!("{}, world!", s);

//hello, world!
```

- Unlimited number of immutable references / readers
- Only one mutable reference &mut T allowed
- Reference Counting structures also available s

STRUCTS

```rust
#[derive(Debug, PartialEq, Eq)]
 struct Person {
    name: String,
    age: u32,
}

println!("person: {:?}", Person {
        name: String::from("bob"),
        age: 23,
    });
```

```
person: Person { name: "bob", age: 23 }
```

ENUMERATIONS

```
enum PC {
    RED,
    GREEN,
    BLUE,
}

let j: u32 = match colour {
    PC::RED => 1,
    PC::GREEN | PC::BLUE => 0,
}
// must be exhaustive `_` used as wildcard
let i: u32 = match colour {
    PC::RED => 1,
    _ => 0
}
```

OPTION

Enumerations can hold data, A.K.A algebraic data types

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

RESULT

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

OPTION AND RESULT PATTERN MATCHING

```
let o1 = Some(1);
match o1 {
    // compiler recognises `i` as i32
    Some(i) => println!("value {} present", i),
    None => println!("empty")
};
```

```
 match r1 {
        Ok(i) => println!("success with value {}", i),
        Err(e) => println!("{}", e)
    }

//also
if let &Ok(i ) = &r1 {
    println!("success with value {}", i)
}
```

OPTION AND RESULT ? SYNTAX

```
fn combine(o1: Option<i32>, o2: Option<i32>) -> Option<i32> {
    let i: i32 = o1?;
    let j: i32 = o2?;
    Some(i + j)
}
```

```
fn combine(r1: Result<i32, Error>,
           r2: Result<i32, Error>) -> Result<i32, Error> {
    let i: i32 = r1?;
    let j: i32 = r2?;
    Ok(i + j)
}
```

METHODS

Decorators for Structs and Enums

```
struct Person { name: String, age: u32 }
```

```
impl Person {
    fn to_csv(&self) -> String {
        format!("{},{}", self.name, self.age)
    }
    fn inc_age(&mut self) {
        self.age += 1;
    }
}
let mut person = Person {
        name: String::from("bob"),
        age: 23,
    };
```

```
println!("{}", person.to_csv());
//bob,23
person.inc_age();
```

```
println!("{}", person.to_csv());
//bob,24
```

TRAITS: SHARED BEHAVIOURS

FP that looks like OOP

```
impl FromStr for Person {
    type Err = Box<dyn error::Error>;

    fn from_str(s: &str) -> Result<Self, Self::Err> {
        let person_info: Vec<&str> = s.split(",").collect();
        let name: String = String::from(person_info[0]);
        let age: u32 = person_info[1].parse()?;
        Ok(Person { name: name, age: age })
    }
}
```

```
println!("{:?}", "Fred,32".parse::<Person>());
//Ok(Person { name: "Fred", age: 32 })
println!("{:?}", "Fred,ZZZ".parse::<Person>());
//Err(ParseIntError { kind: InvalidDigit })
```

OTHER GREAT LANGUAGE FEATURES

- Concurrency libraries
- Macro System

A LOVED LANGUAGE

- Modern tools for system programmers
- Memory/Concurrency safety
- "Zero-cost" abstractions
- Thoughtful Interoperability
  - FFI with other languages
  - Easy compilation for multiple platforms
- Active Community

RESOURCES

- Rust Book
- Rust by Example
- Rust Youtube channel