Intro to Rust, Cargo, Crates

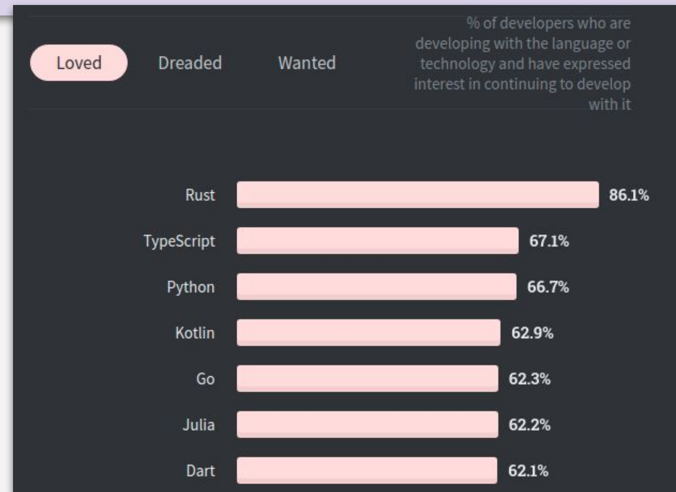
Mustakimur R. Khandaker

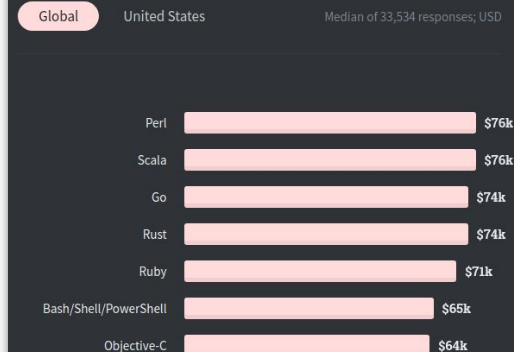
Rust: Memory Safety and Low-level Control

Begun in 2006 by Graydon Hoare.
- Sponsored as full-scale project and announced by Mozilla in 2010.

Takes ideas from **functional** and **OO** languages, and **recent research**.

Key properties: Type safety despite use of concurrency and manual memory management.
- And: No data races.





There's growing interest in the use of memory-safe Rust for systems programming to build major platforms, in particular at Microsoft, which is exploring it for Windows and Azure with the goal of wiping out memory bugs in code written in C and C++. Amazon Web Services is also using Rust for performance-sensitive components in Lambda, EC2, and S3.

Google notes, "The Fuchsia Platform Source Tree has had positive implementation experience using Rust" but it has opted not to support it for end-developers because none of its current end-developers use it and it's not a widely used language.

Google: "Over time we will continue to invest in Rust and see which [Android] system components are better off being written in Rust"

Sudhi Herle, Head of Android Platform Security, says in yesterday's Android Developer weekly video:

> Over time we will continue to invest in Rust and see which system components are better off being written in Rust. We believe Rust will end up fundamentally making the platform safe for all of our users.

Features of Rust

- Lifetimes and Ownership.
  - Key feature for ensuring safety.
- Traits as core of object(-like) system.
- Variable default is immutability.
- Data types and pattern matching.
- Type inference.
  - No need to write types for local variables.
- Generics (aka parametric polymorphism).
- First-class functions.
- Efficient C bindings.

Rust: Installation

Installing rustup on Linux or macOS:

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

Updating to latest version:

```
# rustup update
```

To check whether you have installed Rust correctly:

```
# rustc --version
```

Manually compiling Rust source:

```
# rustc main.rs
```

Executing Rust program:

```
# ./main
```

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

Hello, Cargo!

Check Cargo installation:

```
# cargo --version
```

Creating a project with Cargo:

```
# cargo new hello_cargo
# cd hello_cargo
```

Building and running a Cargo Project (for release: `cargo build --release`

```
# cargo build
  Compiling hello_cargo v0.1.0 (file:///projects/hello_cargo)
    Finished dev [unoptimized + debuginfo] target(s) in 2.85 secs
```

Only execute the project (if you had modified your source code, Cargo would have rebuilt the project before running it):

```
# cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
     Running `target/debug/hello_cargo`
Hello, world!
```

Cargo: Project Structure

A Cargo project structure would look like:

```
$ cd hello_world
$ tree .
.
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

First, let's check out Cargo.toml (manifest):

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name
<you@example.com>"]
edition = "2018"

[dependencies]
```

The last line, [dependencies], is the start of a section for you to list any of your project's dependencies. In Rust, packages of code are referred to as crates.

`Cargo new` has also initialized a new Git repository along with a .gitignore file.

Rust: Dependencies

crates.io is the Rust community's central package registry that serves as a location to discover and download packages.

- cargo is configured to use it by default to find requested packages.

The following example adds a dependency of the `time` and `regex` crate:

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Your Name <you@example.com>"]
edition = "2018"

[dependencies]
time = "0.1.12"
regex = "0.1.41"
```

```
$ cargo build
      Updating crates.io index
   Downloading memchr v0.1.5
   Downloading libc v0.1.10
   ….
   Downloading regex v0.1.41
     Compiling memchr v0.1.5
     Compiling libc v0.1.10
     ….
     Compiling regex v0.1.41
     Compiling hello_world v0.1.0
(file:///path/to/package/hello_world)
```

Let’s use them

```rust
use regex::Regex;

fn main() {
    let re = Regex::new(r"^\d{4}-\d{2}-\d{2}$").unwrap();
    println!("Did our date match? {}", re.is_match("2014-01-01"));
}
```

```
$ cargo run
   Running `target/hello_world`
Did our date match? true
```

Cargo: Standard Package Layout

```
.
├── Cargo.lock
├── Cargo.toml
├── src/
│   ├── lib.rs
│   ├── main.rs
│   └── bin/
│       ├── named-executable.rs
│       ├── another-executable.rs
│       └── multi-file-executable/
│           ├── main.rs
│           └── some_module.rs
├── benches/
│   ├── large-input.rs
│   └── multi-file-bench/
│       ├── main.rs
│       └── bench_module.rs
├── examples/
│   ├── simple.rs
│   └── multi-file-example/
│       ├── main.rs
│       └── ex_module.rs
└── tests/
    ├── some-integration-tests.rs
    └── multi-file-test/
        ├── main.rs
        └── test_module.rs
```

- Cargo.toml and Cargo.lock are stored in the root of your package (package root).
- Source code goes in the src directory.
- The default library file is src/lib.rs.
- The default executable file is src/main.rs.
  - Other executables can be placed in src/bin/.
- Benchmarks go in the benches directory.
- Examples go in the examples directory.
- Integration tests go in the tests directory.

Note: If a binary, example, bench, or integration test consists of multiple source files, place a main.rs file along with the extra modules within a subdirectory of the src/bin, examples, benches, or tests directory.

< Intro to Rust />