Lesson 5 outline

- Rust standard library
- Rust code management
- Packets
- Modules
- Crates

Rust standard library

- Two different library provided by the language
- **Core** library provide the basic of the language
- **Std** library provide the minimal language ecosistem

Rust core library

Rust core library

- Designed to be used in **embedded** and **system** programming (#[no_std])
- No dependecies
- No libc or System libraries

Rust Core library

- All basic types modules i32, char, str, ecc.. (No String)
- Slice module for working with slices
- alloc api (but no implementation)
- Traits for working with borrow and mut borrow data
- Option and Result enums and related functions
- Atomics types (garantee lock-free) for sync programming

Rust Core library

- Time module (temporal quantification)
- Task types and primitives for working with asynchronous tasks

Rust Core library

- Time module (temporal quantification)
- Task types and primitives for working with asynchronous tasks
- Ops Traits (Operators)
- Traits for conversion between types

Rust Core library

- **No** memory management (alloc, realloc) memset and related functions are considered experimental
- **No** IO (file, networking ecc..)
- **No** libc functions, just the ones provided by the llvm compiler.

Rust Core library

- embedded programming
- System programming
- Accessible via the **core::*** package
- Target: #[no_std] environments

Rust Std library

Rust Std library

- Re-export the core library
- Portable
- Minimal and battle-tested functionalities
- Core types like Vec<T> and String
- Multithreading support
- I/O funcionalities
- Standard macros (println, vec, assert, panic ecc..)

Rust Std library

- Requires libc and operative system dependencies
- Provides memory management
- Provides networking primitives
- Implements high level types like lists and hashmaps
- Accessible via the **std::*** package

Rust Std library

```rust
use std::fs;

let contents: String = fs::read_to_string("/path/to/file")
    .expect("failed to read file content");
```

Rust Std library

```rust
use std::env;

for arg in env::args() {
  println!("process argument: {}", arg);
}
```

Rust Std library

```rust
use std::process::Command;

let output = Command::new("echo")
        .arg("Hello world")
        .output()
        .expect("Failed to execute command");
```

Rust Std library

```rust
use std::path::Path;

let path = Path::new("/tmp/foo/bar.txt");

let parent = path.parent(); // "/tmp/foo"

let file_stem = path.file_stem(); // "bar"

let extension = path.extension(); // "txt"
```

```rust
use std::time::{Duration, Instant};

let five_seconds = Duration::from_secs(5);
let one_second = Duration::from_millis(1000);
let now = Intant::now();
if now.elapsed() < five_seconds {
  println("less than 5 seconds from now");
}
```

```rust
use std::thread;

thread::spawn(move || {
    println!("I'm in a child thread");
});
```

Rust code management

Rust code management

- **Workspaces** (organize multiple packages)
- **Packages** for easily build, test and share crates
- **Crates**, a numbers of modules that produces library or executables
- **Modules** and **use** keyword
- Fine control for public and private code via **pub** keyword

pub and **use** keywords

- Every path it’s private by default (struct, enum, funcion, modules ..)
- **pub** keyword sign the path as **public**
- **use** keyword allows using **public** code on different modules
- **mod** keyword declares new modules

```rust
// MyStruct is private
// can be used just onthe same mod
struct MyStruct;

// MyCratePubStruct is public
// can be used by differents mod in my crate
// cannot be exported in other crates
pub (crate) MyCratePubStruct;

// MyPubStruct is public
// can bu used by different mods and other crates
pub MyPubStruct;
```

Make paths public for other crates in lib.rs file

```rust
// lib.rs
pub mod inner;
pub mod something;

// other crates use it as

use you_crate::inner::{InnerPubStruct, pub_inner_func};
use you_crate::something::*;
```

Prelude is a common pattern for public defaults

```rust
// prelude.rs
pub use inner::*;
pub use super::{MuPubStruct, pub_func, PubTrait};

// lib.rs
pub mod prelude;
```

```rust
// lib.rs
mod inner;

pub trait MyPubTrait {}

pub mod prelude {
  pub use inner::*;
  pub use MyPubTrait;
}

// other crate
use crate_io::prelude::*;
```

Private paths are private even in inner mods

```
struct MyStruct;

// error MyStruct is private
mod inner {
  use super::MyStruct;
}
```

mod need to be declared on lib.rs or main.rs to be used

```
// src/main.rs
//    /inner.rs
//    /io.rs

// main.rs
mod inner; // inner can be used on application
// mod io;  // module "io" is not compiled
```

Modules can contains multiple modules

- Two differents mode to declare modules inside an application or library
- 2015 edition (is still used)
- 2018 edition (semplificated)

2015 edition

Cargo.toml
src/main.rs
    /inner/mod.rs
    /inner/something.rs
    /inner/other_mod/mod.rs
    /inner/other_mod/other.rs

```rust
// src/inner/other_mod/mod.rs
pub mod other;
// src/inner/mod.rs
mod other_mod;
pub mod something;
pub use other_mod::other;
// main.rs
mod inner;
use inner::other;
```

2018 edition

```
Cargo.toml
src/main.rs
    /inner.rs
    /inner/something.rs
    /inner/other_mod.rs
    /inner/other_mod/other.rs
```

2018 edition

```
// src/inner/other_mod.rs
pub mod other;
// src/inner.rs
mod other_mod;
pub mod something;
pub use other_mod::other;
// main.rs
mod inner;
use inner::other; // is inner/other_mod/other.rs;
```

crates

- Available via cargo tool
- **Easy** to share on crates.io
- Can contains other crates or packages
- Allows to download and reuse existing code
- Can contains library and executables on the same crate
- Allows simple executable distributions (even with private crates repository)

crates

```
$ cargo new my_lib --lib
$ tree
.
├── Cargo.toml
└── src
    └── lib.rs
```

crates

```
$ cargo new my_app
$ tree
.
├── Cargo.toml
└── src
    └── main.rs
```

crates

```
[package]
name = "my_lib"
version = "0.1.0"
# author is take by your git config
authors = ["user.name <user.email>"]
edition = "2018"

[dependencies]
```

Cargo.toml

```toml
[package]
name = "my_lib"
version = "0.1.0"
# author is take by your git config
authors = ["user.name <user.email>"]
edition = "2018"

[dependencies]
```

dependencies

```
[package]
name = "my_app"
version = "0.1.0"
authors = ["user.name <user.email>"]
edition = "2018"

[dependencies]
log = "0.4.8"
```

dependencies

```
[package]
name = "my_app"
version = "0.1.0"
authors = ["user.name <user.email>"]
edition = "2018"

[dependencies]
log = { git = "https://github.com/rust-lang/log", branch = "master" }
```

dependencies

```
[package]
name = "my_app"
version = "0.1.0"
authors = ["user.name <user.email>"]
edition = "2018"

[dependencies]
log = "*"
```

dependencies

```
[package]
name = "my_app"
version = "0.1.0"
authors = ["user.name <user.email>"]
edition = "2018"

[dependencies]
# relative to Cargo.toml
utils = { path = "utils" }
```

Conditional compilation

```
[dependencies.my_lib]
version = "0.1.0"
default-features = false
features = ["log"]
```

Cargo.toml

Cargo is powerfull

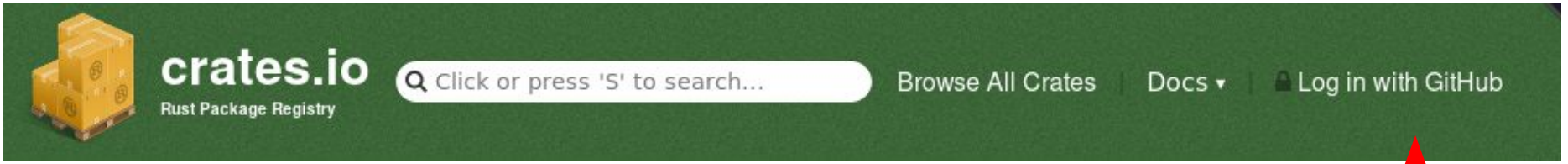
There is an entire book witch explains how to use it and how can be configurated

https://doc.rust-lang.org/cargo/

Publishing on crates.io

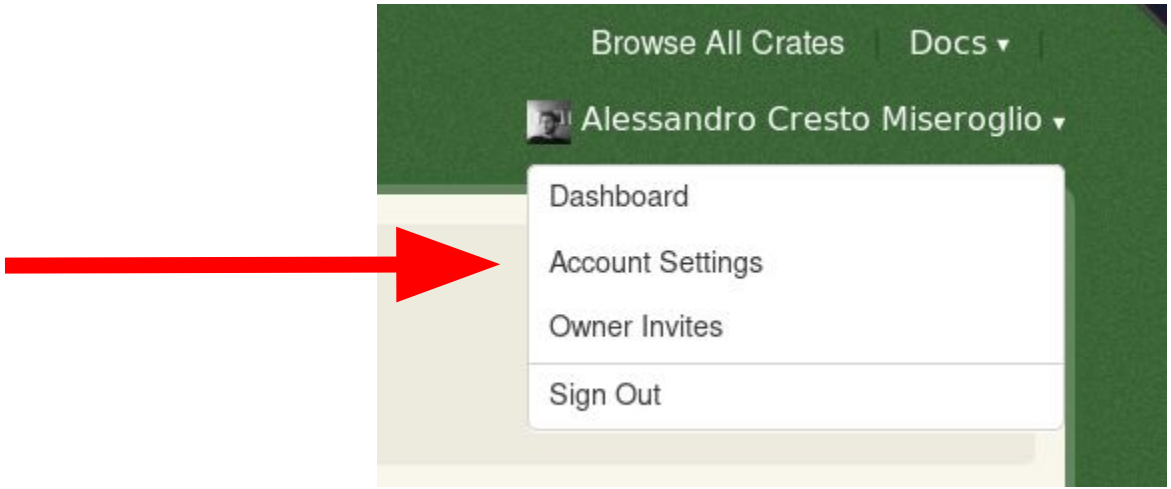
Go to https://crates.io

Log in with your github account



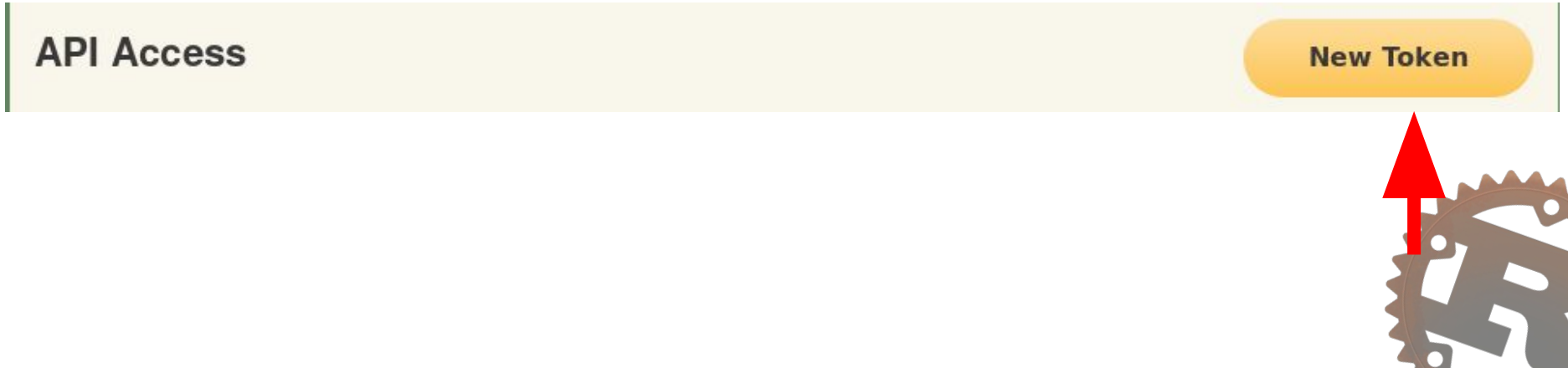
Publishing on crates.io

Click on your account and then to account settings



Publishing on crates.io

Create new token



Login with cargo and publish

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
$ cargo publish
```

Appendix: **workspace**

```
# Cargo.toml

[workspace]
members = [
    "my_crate",
    "my_other_crate",
]
```