Macros

An overview and examples for the diverse types of macros you can use in Rust.

- Declarative Macros
- Procedural Macros

Declarative Macros

Declarative macros look like a function: e.g. `println!()`, `format!()` or `todo!()`. They have an exclamation mark between the name and the parameter list.

Declarative macros are like "shortcuts", it is usually more code, packed in a code block, which will generated at the place where it is called. This has the benefit, that it is not a function call, hence does not add to the stack or needs to consider moving. It is like a template, which will be copied in place.

Procedural Macros

Procedural macros are "headers" added above e.g. a struct, **take those information and generate different code out of it**. They look like this: `#[example]`. They are way more complex, but give you the **ability to generate complex code**.

Procedural Macros takes an object and **produces a stream of tokens**, a "TokenStream". You then manipulate the token stream, so the code at this place will be the desired generated result.

Those generated results are not "copied in the source", they are cached. You only see your source code and the macro call. But there is a command to show, how the generated result looks like. But it resolves not only your macro, it resolves all macros, hence debugging using it can be helpful, but hard work.

Derive Macros

```
// Example

#[derive(Debug, Clone)]
struct Example {
  a: i32,
  b: i32,
}
```

Attribute-Like Macros

Attribute-Like macros modify **structs, functions or modules**.

```
//example

use my_macros::log_call;

#[log_call]
fn add(a: i32, b: i32) -> i32 {
    a + b
```

```
}
```

Function-Like Macros

Function-Like macros automatically **generate a function**.

```
// Example


create_hello_fn!(say_hello);


fn main() {
    say_hello();
}
```

They **look similar to declarative macros**, but they don't use a template, they use a TokenStream and you are able to analyse the syntax and generate the code (by using *syn and quote*).