Macro in Rust

What are macros?

Macros are pieces of code that write other pieces of code.

When called, they *expand* into more code.

Why not use functions?

Unlike functions:

- macros always expand at *compile time*, so they don’t cost any runtime
- macros can have a variable number of arguments

Multiple types of macros

There are two kind of macros in Rust: procedural and generative.

There are two ways of using macros in Rust:

- function-like style
- derive style
- attribute style

	Declarative Macros	Procedural Macros
Function-like macros	least powerful	can parse everything
Derive macros		useful to annotate structures
Attribute macros		useful to annotate code

That sounds complicated, why not use C macros?

C macros are based on “search and replace”. This is simple to understand, but extremely hard to use.

That sounds complicated, why not use C macros?

Naive macros are OK for simple cases

```c
#include <stdio.h>

#define MULTIPLY(a, b) a * b

int main() {
    int c = MULTIPLY(2, 5);
    printf("%d", c);
}
```

```c
#include <stdio.h>

int main() {
    int c = 2 * 5;
    printf("%d", c);
}
```

That sounds complicated, why not use C macros?

Naive macros are OK for simple cases, but broken for others

```c
#include <stdio.h>

#define MULTIPLY(a, b) a * b

int main() {
   int c = MULTIPLY(1+1, 5);
   printf("%d", c);
}
```

```c
#include <stdio.h>

int main() {
   int c = 1 + 1 * 5;
   printf("%d", c);
}
```

That sounds complicated, why not use C macros?

Naive macros are OK for simple cases, but broken for others, so you need parentheses.

```c
#include <stdio.h>

#define MULTIPLY(a, b) (a) * (b)

int main() {
    int c = MULTIPLY(1+1, 5);
    printf("%d", c);
}
```

```c
#include <stdio.h>

int main() {
    int c = (1 + 1) * (5);
    printf("%d", c);
}
```

That sounds complicated, why not use C macros?

Naive parentheses are OK for simple cases

```c
#include <stdio.h>

#define ADD(a, b) (a) + (b)

int main() {
    int c = ADD(1, 5);
    printf("%d", c);
}
```

```c
#include <stdio.h>

int main() {
    int c = (1) + (5);
    printf("%d", c);
}
```

That sounds complicated, why not use C macros?

Naive parentheses are OK for simple cases, but broken for others

```c
#include <stdio.h>

#define ADD(a, b) (a) + (b)

int main() {
   int c = 2 * ADD(1, 5);
   printf("%d", c);
}
```

```c
#include <stdio.h>

int main() {
   int c = 2 * (1) + (5);
   printf("%d", c);
}
```

That sounds complicated, why not use C macros?

Naive parentheses are OK for simple cases, but broken for others, so you need even more parentheses.

```c
#include <stdio.h>

#define ADD(a, b) ((a) + (b))

int main() {
    int c = 2 * ADD(1, 5);
    printf("%d", c);
}
```

```c
#include <stdio.h>

int main() {
    int c = 2 * ((1) + (5));
    printf("%d", c);
}
```

That sounds complicated, why not use C macros?

```c
#include <stdio.h>

// let's suppose we need a block here
#define MAKE_ZERO(x) \
    {                \
        x = 0;       \
    }

int main() {
    int a = 4;
    MAKE_ZERO(a);
    printf("%d", a);
}
```

```c
#include <stdio.h>

int main() {
    int a = 4;
    {
        a = 0;
    }
    printf("%d", a);
}
```

That sounds complicated, why not use C macros?

```c
#include <stdio.h>

// let's suppose we need a block here
#define MAKE_ZERO(x) \
    {                \
        x = 0;       \
    }

int main() {
    int a = 4;
    if (a == 4)
        MAKE_ZERO(a);
    else
        a = 8;
    printf("%d", a);
}
```

```c
#include <stdio.h>

int main() {
    int a = 4;

    if (a == 4) {
            a = 0;
    }
    ;
    else
        a = 8;
    printf("%d", a);
}
```

That sounds complicated, why not use C macros?

```c
#include <stdio.h>

// let's suppose we need a block here
#define MAKE_ZERO(x) \
    do {             \
        x = 0;       \
    } while (0)

int main() {
    int a = 4;
    if (a == 4)
        MAKE_ZERO(a);
    else
        a = 8;
    printf("%d", a);
}
```

```c
#include <stdio.h>

int main() {
    int a = 4;

    if (a == 4)
        do { a = 0; } while (0)
    else
        a = 8;
    printf("%d", a);
}
```

Declarative Macros

Declarative Macros - simple use case

```
macro_rules! shout {
    ($msg:expr) => {
        println!("📣 {}", $msg.to_uppercase());
    };
}

fn main() {
    shout!("time for some macro");
}
```



```
fn main() {
    println!("📣 {}", $msg.to_uppercase());
}
```

Declarative Macros - simple use case

```
macro_rules! add {
    ($a:expr, $b:expr) => {
        $a + $b
    };
}


fn main() {
    let sum =  1 * add!(1, 5);
    println!("{sum}");
}
```

Declarative Macros - variable number of arguments

```
macro_rules! add {
    ($first:expr $(, $rest:expr)*) => {
        $first $(+ $rest)*
    };
}

fn main() {
    let sum = 1 * add!(1, 5, 10);
    println!("{sum}");
}
```

Declarative Macros - variable number of arguments

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Simplified version of the vec macro

How to see the result of a macro I’m writing?

```
mod macro_test {
    fn function() {
        let sum = 1 * add!(1, 5);
        println!("{sum}");
    }
}
```

cargo expand macro_test

```
mod macro_test {
    fn function() {
        let sum = 1 * (1 + 5);
        {
            ::std::io::_print(format_args!("{0}\n", sum));
        };
    }
}
```

How to see the result of a macro I’m writing?

To do it automatically every time you save your file:

1: cargo install --locked bacon

2: In bacon.toml:

```
[jobs.macro]
command = ["cargo", "expand", "macro_test"]
need_stdout = true
```

3: bacon macro

Good practices

Use the complete path of the functions instead of their names

- Always with ::
- This prevents collision
- The expanded code is ugly, but no one will see it

Don’t overdo it

- Macros are hard to read and maintain
- Your IDE might not fully support macros

Procedural Macros

Warning

This is a very advanced topic. These slides does not cover enough to understand it.

I would suggest having a look at this video if you want to understand what's going on: https://www.youtube.com/watch?v=SMCRQj9Hbx8

What are procedural macros

Proc-macros:

- are way more complex
- must be placed in their project
- increases your compile time
- that project must declare it exposes proc macro in their cargo.toml

But:

- They are very powerful

Let’s write a proc-macro builder

```
use builder_macro::Builder;

#[derive(Builder)]
struct T {
    a: String,
    b: u32,
    c: u32,
}
```

What we need to use

```
use proc_macro::TokenStream;
use proc_macro2::{Ident, Span, TokenStream as TokenStream2};
use quote::quote;
use syn::{DeriveInput, Type, parse_macro_input};
```

The most important part: model with types

```
struct FieldData {
   original_field_name: Ident,
   original_field_type: Type,
   associated_generic_name: Ident,
   name_of_struct_provided: Ident,
   name_of_struct_not_provided: Ident,
}

struct FieldDatas {
   data: Vec<FieldData>,
}
```

Converting a type to a token stream

```
impl FieldDatas {
   fn compute_structs_for_each_fields(&self) -> Vec<TokenStream2> {
       self.data
           .iter()
           .map(|field| {
               let name_of_struct_provided = field.name_of_struct_provided.clone();
               let original_field_type = field.original_field_type.clone();
               let name_of_struct_not_provided = field.name_of_struct_not_provided.clone();
               quote! {
                   struct #name_of_struct_provided {
                       data: #original_field_type
                   }
                   struct #name_of_struct_not_provided {}
               }
           })
           .collect()
   }
}
```

Proc-macro - builder overview

```
#[proc_macro_derive(Builder)]
pub fn my_macro_derive(input: TokenStream) -> TokenStream {
    // Parse the input tokens into a syntax tree
    let input = parse_macro_input!(input as DeriveInput);

    let field_datas = match parse_to_fields(&input) {
        Ok(x) => x,
        Err(x) => {
            return x;
        }
    };

    let structs_for_each_fields = field_datas.compute_structs_for_each_fields();

    // Convert the expanded code into a TokenStream and return it
    let mut extanded = TokenStream2::new();
    for structs in structs_for_each_fields {
        extanded.extend(structs);
    }
    TokenStream::from(extanded)
}
```