TD-5

Rust as a *safe language*: Pattern Matching, Handling Results/Errors, Options and Simple macros

Pierre Cochard and Tanguy Risset

5TC-RUST 2025

Summary

- Pattern matching in Rust
- The `Result` enum type
- The `Option` enum type
- Simple macros with `macro_rules!`

Pattern matching in Rust

- Pattern matching is the act of checking a given sequence of tokens or expressions for the presence of one or more specific patterns.
- Implemented in many programming languages (Rust, Haskell, Swift, etc.) and tools.
- In Rust, *patterns* and *pattern matching* constitute a specific syntax, that is used in different places in the language (`match` **statements**, `if let` expressions, function parameters, **simple macros**, etc.)

Pattern

A *pattern* consists of some combination of the following:

- Literals
- Destructured arrays, enums, structs, or tuples
- Variables
- Wildcards and Placeholders (most used: `_` meaning *Anything*, and `..` meaning *The rest*). A wildcard matches values; a placeholder stands in for something not yet specified (often a type).

`match` statements

The primary, and most explicit, use of pattern matching in Rust is done through the **`match` statement**, which can be perceived as the Rust-equivalent of a `C` `switch`, but with additional features.

If a given pattern matches, its associated instructions are executed, and the `match` block terminates (it does not check for other matching patterns, the first matching pattern is chosen).

```
// C basic switch case:
enum Colors {
    Red, Blue, Green, Yellow, Orange
};

bool match_color_orange(Colors color) {
    switch (color) {
        case Orange: {
            printf("Orange!\n");
            return true;
        }
        case Red:
        case Blue: {
            printf("Not orange :(\n"));
            return false
        }
        default: {
            printf("Still not orange\n");
            return false;
        }
    }
}
```

```
enum Colors {
    Red, Blue, Green, Yellow, Orange
}

fn match_color_orange(color: Colors) -> bool {
    match color {
        Colors::Orange => {
            println!("Orange!");
            true
        }
        Colors::Red | Colors::Blue => {
            println!("Not orange :(");
            false
        }
        _ => {
            println!("Still not orange...");
            false
        }
    }
}
```

`match` statements can be directly bound to variables with `let` statements:

```
enum Colors { Red, Blue, Green, Yellow, Orange }

let color = Colors::Red;

let is_color_warm = match color {
    Colors::Orange => true,
       Colors::Red => true,
    Colors::Yellow => true,
                _ => false
};
```

Matching **ranges** is also supported:

```
fn match_number(number: i32) {
    match number {
          50..=99 => println!("Between 50 and 99"),
       100..=1000 => println!("Between 100 and 1000"),
                _ => println!("Other value")
    }
}
```

Any other type of expression can be matched! from string types:

```
fn match_str(s: &'static str) {
    match s {
        "Orange" => println!("Orange!"),
        "Yellow" => println!("Not orange"),
        _ => println!("Something else...")
    }
}
```

to other kinds of collections:

```
fn match_tup(tup: (i32, i32)) {
    match tup {
        (0, 0) => println!("Zeroes!"),
        (1, 1) => println!("Ones"),
        _ => println!("Something else...")
    }
}

match_tup((1, 1));
match_tup((0, 1));

fn match_array(arr: [u8; 3]) {
    match arr {
        [0, 1, 2] => println!("Array match!"),
        _ => println!("No match")
    }
}
```

Question 1)

Write a `match` statement which applies to any `i32` number. It **should only match the two following patterns**:

1. The value is **below** `100` (including negative numbers);
2. The value is **equal or higher** than `100`.

`match` statements: "flexible" patterns

the `match` statement can test any kind of value, and it also extends to custom and composite types, **including `struct` instances**. A custom `struct` can be indeed either matched by its contents in a very precise manner:

```
struct Point { x: isize, y: isize}

let point = Point {x: 0, y: 100};

match point {
    // Only match Point if its 'x' member is equal to 0
    // and 'y' is equal to 100:
    Point {x: 0, y: 100} => println!("Match!"),
    _ => println!("No match!")
}
```

Or, in a more *flexible* way, using, for instance, `ranges` for its member values:

```
struct Point { x: isize, y: isize}

let point = Point {x: 25, y: 100};

match point {
    // Only match if 'x' is between 0 and 100,
    // and 'y' is between 50 and 100
    Point {x: 0..=100, y: 50..=100} => println!("Match!"),
    _ => println!("No match!")
}
```

Finally, the `_ =>` expression can be extended to any kind of value (or field value) that we want **to ignore**. This can also be done using the `..` syntax, which will ignore all the following values or field values:

```
struct Point { x: isize, y: isize }

let p = Point {x: 0, y: 100};

match p {
    // Only match if 'x' is between 0 and 100,
    // and ignore the 'y' field:
    Point {x: 0..=100, y: _} => println!("Match!"),
   // Only match if 'x' is between 101 and 1000,
    // Similarly, the '..' syntax will ignore all the struct fields after 'x':
    Point {x: 101..=1000, ..} => println!("Match!"),
    _ => println!("No match!")
}
```

For compound/collection types, the `..` syntax may be followed by other patterns:

```
let tup = (0, 1, 2, 3, 4);

match tup {
    // The first element of the tuple should be '0',
    // and the last should be '4',
    // we ignore the values in-between:
    (0, .., 4) => println!("Match!"),
    _ => println!("No match")
}
```

Question 2)

Implement a `match` statement on a `&[i32]` slice. It should match all of the following patterns:

1. The slice's **first value** should be `0`;
2. The slice's **second value** should be either `10` or `20`;
3. The slice's **final value** should be `100`;
4. The slice can have **an arbitrary size**.

```
fn match_slice(s: &[i32]) -> bool {
    match s {
        ...
    }
}
```

You can use the following `assert!` statements to test your code:

```
fn match_slice(s: &[i32]) -> bool;

assert_eq!(match_slice(&[1, 20, 20, 30, 100]), false);
assert_eq!(match_slice(&[0, 5, 20, 30, 100]), false);
assert_eq!(match_slice(&[0, 10, 20, 30, 99]), false);
assert!(match_slice(&[0, 10, 20, 30, 40, 100]));
assert!(match_slice(&[0, 20, 20, 30, 50, 60, 70, 80, 90, 100]));
```

Pattern everywhere

Pattern can be used

- In `let` statements: `let (x,y)=(1,2)`
- In `if let` statments (`while let` and for loops)

```
if let Some(color) = favorite_color {[...]} else {[...]}
```

- In function parameters `fn print_coordinates(&(x, y): &(i32, i32)) {[...]}`

In `let` , functions and for loops, patterns must be *irrefutable* (i.e. cannot fail)

The `Result` enum type

To handle and propagate **runtime errors**, Rust relies on a simple but efficient mechanism based on an `enum` : the `Result<T, E>` enum, which is defined as:

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

The generics `T` and `E` types have no trait implementation predicate whatsoever, they could be anything, for instance:

```
// This function returns a u8 slice if there is no error, a Vec<f32> if there is.
// This is probably not very useful, but it's still perfectly valid code:
fn my_function(i: i32) -> Result<&[u8], Vec<f32>> {...}

// This too (empty tuple type for both):
fn my_function(i: i32) -> Result<(),()> {
    Ok(())
}
```

The `unwrap()` method can be used to extract the `Ok` argument from a `Result<...>`. This will be explained in more detail further, but you will need it to test your function:

```
let u = my_function(3).unwrap();
```

Question 3)

Write a function called `positive()` that takes an `i32` as argument and checks whether it is (strictly) positive. It should return a `Result` with the same argument value if it is positive, or a `String` with an error message if the argument is negative or equal to zero.

Propagating Errors

An Error in Rust can be propagated down the call stack by using the question mark operator: `?`

```
fn my_function(i: i32) -> Result<i32, ()> {...}

fn my_other_function() -> Result<i32, ()> {
    // Append the '?' operator right after the function call:
    let mut i = my_function(1)?;
    // Do something with 'i':
    i += 1;
    // Return an 'Ok' result with the modified 'i' value:
    Ok(i)
}
```

Here, the `my_function(1)?` function call means:

- *if the result enum value is `Err` (an error), then propagate the error now, by returning the same `Err` from `my_function()`, otherwise, continue with the rest of the code*.

This code could also be implemented with an equivalent `match` statement, but is a bit more verbose:

```
fn my_other_function() -> Result<i32, ()> {
    let mut i = match my_function(1) {
        Ok(i) => i,
        Err() => return Err(())
    };
    i += 1;
    Ok(i)
}
```

Question 4)

Implement the same mechanism for the previous `positive(i: i32)` example, using the `?` syntax, and test it with both positive and negative values **in a new function with the same return type**, in order to see what happens.

```
// Your 'positive' function:
fn positive(i: i32) -> Result<i32, String> {...}

// Define a new function, and call 'positive(...)' from here:
fn check_positive() -> Result<i32, String> {
    ... // <- test positive & negative values here using the '?' syntax
}

fn main() {
    println!("{:?}", check_positive());
}
```

Returning a `Result` from the `main()` function

In general, returning with or without error from a `main` function, such as in the `C` or `C++` programming languages, is done by returning an **integer exit code** (`0` for success, error otherwise):

```
int main(void) {
    // No error, return 0:
    return 0;
}
```

In Rust, the `main()` function **has no return type by default**, and returning a `i32` is not accepted by the compiler. For instance, the following code is invalid:

```
fn main() -> i32 {
    0
}
```

In this case, the compiler prints the following:

```
error[E0277]: `main` has invalid return type `i32`
 --> src/main.rs:3:14
  |
3 | fn main() -> i32 {
  |              ^^^ `main` can only return types that implement `Termination`
  |
  = help: consider using `()`, or a `Result`
```

The `Termination` trait documentation indeed indicates that only the following types are valid:

```
impl Termination for Infallible;
impl Termination for !;
impl Termination for ();
impl Termination for ExitCode;
impl<T: Termination, E: Debug> Termination for Result<T, E>;
```

Therefore, we can see that propagating a `Result` down to `main()` is possible, but is still a bit of a specific case. The type `T` held by the `Ok` enum value must implement the `Termination` trait, and the type held by the `Err` enum value must implement the `Debug` trait.

The following still does not work because `i32` does not implement the `Termination` trait:

```
fn main() -> Result<i32, String> {
    Ok(0)
}
```

But the following works:

```
// Using an empty tuple as the 'Ok' result:
fn main() -> Result<(), String> {
    Ok(())
}

// Or using the ExitCode type:
fn main() -> Result<std::process::ExitCode, String> {
    Ok(std::process::ExitCode::from(0))
}
```

Using a valid `Result` type in the `main()` function, propagate the `Result` of our `positive()` function down.

```
fn positive(i: i32) -> Result<i32, String> {...}

fn check_positive() -> Result<i32, String> {...}

// Use a valid Result type here:
fn main() -> Result<?, ?>{
    // Call the 'check_positive' function here,
    // and propagate its Result as the main() return type:
}
```

Handling `Result` types immediately

In some cases - when propagating an error is not possible, or unconvenient - dealing immediately with a `Result` type is preferrable. This is why certain methods, such as `.unwrap()` or `.expect()` are natively implemented, and quite commonly used:

The `.unwrap()` method, for instance, will induce a `panic!` call and will exit the program immediately when encountering an error. Otherwise it will return the `Ok` value safely:

```
// Get the current working directory:
let dir: std::path::PathBuf = std::env::current_dir().unwrap();
println!("{:?}", dir);
```

The `.expect()` method is really similar, but will allow the user to print a custom `&str` message on error, which will be prepended to the actual display of the `Err` contents:

```
fn my_function() -> Result<(), i32> {
    Err(1)
}

my_function().expect("Error! Now exiting program with error code");
```

Other similar helper methods also exist, with different behaviors:

- `.unwrap_or(other: T)` returns the value `other` in case of an `Err`
- `.unwrap_or_default()` returns the type's default value in case of an `Err`
- `.unwrap_or_else(func: Fn)` executes a custom function in case of an `Err`
- etc.

The `panic!(msg)` macro interrupts the program immediately and prints a custom error message:

```
fn main() {
    use std::io;
    // Read input from stdin:
    let mut buffer = String::new();
    println!("Please enter password:");
    io::stdin().read_line(&mut buffer).unwrap();
    // Panic if password is not long enough:
    if buffer.len() < 8 {
        panic!("Password should be at least 8 characters");
    } else {
        println!("{buffer}");
    }
}
```

The `assert!(bool)`, `assert_eq!(lhs, rhs)`, and `assert_ne!(lhs, rhs)`, verify a boolean statement or check equality between two elements:

```
fn main() {
    use std::io;
    // Read input from stdin:
    let mut buffer = String::new();
    println!("Please enter password:");
    io::stdin().read_line(&mut buffer).unwrap();

    // We assert that the password is at least 8 characters
    // This will cause a 'panic' if the assertion is false:
    assert!(buffer.len() >= 8, "Password should be at least 8 characters");

    // Here, we forbid choosing 'password' as a password:
    assert_ne!(buffer.trim(), "password", "Choosing 'password' as password is unsafe.");
}
```

The `Option` enum type

The `Option` enum in Rust is somewhat similar to the `Result` enum, but its main purpose is to indicate the **presence or absence of a value**, rather than an error. It has the following definition:

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

As an example, let's suppose we want to build a list of people to contact, with various information, such as the contact's name and address, and optionally phone number and/or e-mail, we could for instance define the following `struct` :

```
struct MyContact {
    name: &'static str,
 address: &'static str,
   email: Option<&'static str>,
   phone: Option<&'static str>,
}

let mut list: Vec<MyContact> = Vec::new();
list.push(MyContact {
       name: "Marlo Stanfield",
    address: "2601 E Baltimore St, Baltimore, MD 21224",
      email: None,
      phone: Some("+1 410-915-0909")
});
```

We can then take advantage of the `Option` enum and **pattern matching** to decide of the best way to contact each person in the list:

```
for contact in list {
    match (contact.email, contact.phone) {
        (Some(email), Some(phone)) => {
            if is_email_correct(email) {
                contact_by_email(email);
            } else {
                contact_by_phone(phone);
            }
        }
        (Some(email), None) => contact_by_email(email),
        (None, Some(phone)) => contact_by_phone(phone),
        (None, None) => send_mail_to_address(contact.address)
    }
}
```

As for the `Result` type, `Option` can be checked and handled immediately using the same `.unwrap()`, `.expect()` methods.

```
fn money_left() -> Option<i32>;
money_left().expect("No money left :(");
```

Question 6)

Implement a small database of books that would be used by a library. It should have a `search_book(...)` function which searches for a specific book using its name **and/or** the name of the author. The function should return a reference to the `Book` object if it has been found, or `None` otherwise.

```
struct Book {
      name: String,
    author: String,
}

#[derive(Default)]
struct LibraryDatabase {
    books: Vec<Book>
}

impl LibraryDatabase {
    // The function to implement:
    fn search_book(&self,
        name: Option<&'static str>,
        author: Option<&'static str>
    ) -> Option<&Book> {...}
}
```

You can use the following `#[test]` function to verify your code:

```
#[test]
fn test() {
    let mut database = LibraryDatabase::default();
    database.books.push(Book { name: String::from("Peter Pan"), author: String::from("Barrie")});

    assert!(database.search_book(Some("Peter Pan"), None).is_some());
    assert!(database.search_book(None, Some("Barrie")).is_some());
    assert!(database.search_book(None, None).is_none());
    assert!(database.search_book(Some("Barrie"), None).is_none());
    assert!(database.search_book(None, Some("Peter Pan")).is_none());
    assert!(database.search_book(Some("Alice in Wonderland"), Some("Barrie")).is_none());
    assert!(database.search_book(Some("Peter Pan"), Some("Lewis Carroll")).is_none());
}
```

Question 7)

Using `Result`, `Option` and all the previous examples, create a program which **parses a password**, with the following rules:

- be at least **8 characters long**;
- should contain at least **one of these special characters**: `!`, `?` or `_`;
- should contain at least **one number**;
- should **not** contain **any whitespace**.
- A **specific error message** should be displayed for each rule.

hint: in order to chain the tests in a functionnal way, one can use the **`then_some(self, value T) -> Option(T)`** called on a `bool` that returns `Some(value)` if the `bool` is true, or **`ok_or<E>(self, err: E)-> Result<T,E>`** that can transform an `Option` in a `Result`.

Note: in order to verify your program, you can implement a `#[test]` function, and then run `cargo test` on your program:

```
#[test]
fn password_test() {
    let pwd0 = String::from("pass");
    let pwd1 = String::from("password");
    let pwd2 = String::from("password!");
    let pwd3 = String::from("pass word!");
    let pwd4 = String::from("password!1");
    assert!(parse_password(&pwd0).is_err());
    assert!(parse_password(&pwd1).is_err());
    assert!(parse_password(&pwd2).is_err());
    assert!(parse_password(&pwd3).is_err());
    assert!(parse_password(&pwd4).is_ok());
```

Simple macros with `macro_rules!`

Unlike other programming languages, such as `c` or `c++`, Rust's macro system is based on **abstract syntax trees** (AST), instead of string preprocessing, which makes them a bit more complex to use, but also more reliable and powerful. macros are expanded before the compiler interprets the meaning of the code. The difference between a macro and a function is that macro definitions are more complex than function definitions because you’re writing Rust code that writes Rust code. Due to this indirection, macro definitions are generally more difficult to read, understand, and maintain than function definitions.

Simple macros vs other macros

Throughout this course, we have already encoutered a few of them, including `vec!`, `panic!`, `assert!`, and of course `println!`. These macros are defined by the `macro!(...)` syntax (don't forget the trailling exclamation mark) and are called *simple macros*, as opposed to Rust's more complex macro systems, such as *attribute* macros (for instance the `#[test]` function attribute), and *derive* macros (the `#[derive(Debug)]` statement on top of a `struct`), which we have already both seen as well.

Basic `macro_rules!` usage

Simple macros can be defined anywhere in our code, using `macro_rules!`:

```
macro_rules! hello_world {
    () => {
        println!("Hello World!")
    };
}

hello_world!();
```

Here, we defined a `macro!` that takes no argument, which is indicated by the `()` statement. Our `hello_world!()` macro call will be under the hood replaced by the contents that we defined within the `=> { ... }` block.

Advantages of using `macros`

Our `hello_world!()` example is of course not very useful, and in fact adds unnecessary noise to a very simple piece of code, but think of the `vec!` macro for instance:

```
let v1 = vec![1, 2, 3, 4, 5];
let v2 = vec!();
let v3 = vec![1];
```

Defining the three different vectors **by hand** would actually mean writing the following code:

```
let v1 = <[_]>::into_vec(Box::new([1, 2, 3, 4, 5]));
let v2 = Vec::new<i32>();
let v3 = std::vec::from_elem(1, 1);
```

Notice how these three vectors are each time created in a very different way? In this case, the `vec!` macro allows defining a more *practical* and *unified* way of instantiating a `Vec` object, without having to remember all the (sometimes complex) underlying code. Furthermore, as you can see with this example, a `macro!` can also accept a **variable number of arguments**, which is not the case with a Rust function.

The different types of arguments (or *fragment specifiers*)

As you may already have guessed with our first basic macro example, which uses the `=>` operator, `macro_rules!` relies on **pattern matching** to parse its arbitrary number of arguments.

`macro_rules!` can parse different kinds of patterns, including:

- `()` : the empty pattern, which means no argument (our previous example);
- `block` : a **block expression**, surrounded by `{ }` ;
- `expr` : any kind of Rust expression;
- `ident` : an identifier (the name of a variable, function, etc.);
- `literal` : a number/string or other kind of litteral;
- `ty` : a type
- `tt` : a TokenTree

Let's now try an example with an actual argument. Here, we will use the `ident` designator in order to create functions from a simple macro call:

```
macro_rules! define_fn {
    ($fn_name:ident) => {
        fn $fn_name() {
            println!(
                "This function's name (ident) is: '{}()'.",
                stringify!($fn_name)
            );
        }
    }
}
define_fn!(foo);
define_fn!(bar);
foo();
bar();
```

Let's break this code piece-by-piece:

```
($fn_name:ident) => {
```

→ Instead of an empty pattern `()`, we use the pattern `($fn_name:ident)`, in which `$fn_name` would be the name of the argument, and `ident` its type. The dollar sign ($) is used to declare a variable in the macro system that will contain the Rust code matching the pattern.

```
fn $fn_name() {
```

→ Within our generated code block, we declare a function with the name taken from our `$fn_name` ident argument.

```
    println!(
        "This function's name (ident) is: '{}()'.",
        stringify!($fn_name)
    );
```

→ We then define the function's *body*, with a `println!` call, in which we print the ident's name using a utility macro called `stringify!`. This very useful macro will transform our `$fn_name` identifier into a `&'static str` object;

Question 7)

What would be the generated code for the `define_fn!(foo)` macro call?

Question 8)

Create a similar macro, but this time the generated code should define a `struct` and its `impl` block like the following:

```
// All of this code should be generated by our new macro,
// but the name 'Foo' should be made variable:
struct Foo {
    print: &'static str
}
impl Foo {
    fn new() -> Foo {
        Foo {
            print: "Foo"
        }
    }
}
```

```
// The macro to implement:
macro_rules! define_struct {
    ...
}

// Use the following to verify that the macro is correct:
define_struct!(Foo);

let bar = Foo::new();
assert_eq!(bar.print, "Foo");
```

Pattern overloading

`macro_rules!` definitions can accept an arbitrary number of patterns, in a very simple way. Let's try it out on our `define_fn!` macro. We will add another pattern allowing to add arbitrary code expressions to the created `fn`.

```
macro_rules! define_fn {
    ($fn_name:ident) => {
        fn $fn_name() {
            println!(
                "This function's name (ident) is: '{}()'.",
                stringify!($fn_name)
            );
        }
    }; // <-- pattern blocks must end with a semicolon if they're followed by other blocks

    // Our new pattern:
    ($fn_name:ident, $additional_code:expr) => {
        fn $fn_name() {
            println!(
                "This function's name (ident) is: '{}()'.",
                stringify!($fn_name)
            );
            // Append the additional code 'expr' at the end of the defined fn:
            $additional_code
        }
    }
}
```

```
define_fn!(foo);
define_fn!(bar, println!("Additional code"));
foo();
bar();
```

Here, we added the `($fn_name:ident, $additional_code:expr)` pattern, which is composed of two arguments: the same `ident` argument, followed by an `expr` argument, which can be any valid Rust expression. The two arguments are separated by a comma `,` but the choice of a comma is completely arbitrary, it could be (almost) **any symbol**.

Question 8)

In our previous example, try replacing the `,` symbol between `$fn_name:ident` and `$additional_code:expr` with another one (e.g. + or *). Then, call the `define_fn!` macro with two arguments separated by the same new symbol, and see what it does.

Pattern matching for `macro_rules!`

Pattern matching for `macro_rules!` is quite different from pattern matching used in the `match` keyword. Macros can also easily deal with **pattern repetition** by using a special syntax, which resembles the one used for *regular expressions*. In particuler, the usual operators of regular expressions can be used: `'_'`, `'+'` or `'*'` (one object, a repetition of objects -- at least one, a repetition of objects - possibly 0). In the following example, we want to replace the `std::cmp::max()` function to take an arbitrary number of arguments:

```
let mut max = std::cmp::max(1, 2);
max = std::cmp::max(max, 3);
max = std::cmp::max(max, 4);
max = std::cmp::max(max, 5);
```

In this case, having a macro like the following could prove useful, and would lighten the code a lot:

```
max!(1, 2, 3, 4, 5, 6*7, 3*4);
```

The way to do this is to use the `$(...),+` syntax, as follows:

```
macro_rules! max {
    // Only one argument 'x', return 'x':
    ($x:expr) => {$x};
    // At least two arguments,
    // - 'x' being the first,
    // - 'y' being one or more additional argument(s),
    // which is defined by the '$(...),+' syntax:
    ($x:expr, $($y:expr),+) => {
        // We recursively call 'max!' on the tail 'y'
        std::cmp::max($x, max!($($y),+))
    }
}
```

The `+` in the `$($y:expr),+` syntax means **one or more instances** of the `($y)` expression, separated by a comma `,`.

> **Note:** The `*` symbol also exists, and means *zero or more instances of the pattern*.

Now, if we were to call the `max!` macro the following way, we would only match the first pattern `($x:expr) => {$x}`:

```
max!(1);
```

- With two arguments, we would match the second pattern `($x:expr, $($y:expr),+)` with a single additional argument:

```
max!(1, 2); // expands to:
std::cmp::max(1, max!(2)); // expands to:
std::cmp::max(1, 2);
```

- And with more arguments recursively:

```
max!(1, 2, 3); // expands to:
std::cmp::max(1, max!(2, 3)); // expands to:
std::cmp::max(1, std::cmp::max(2, max!(3))); // expands to:
std::cmp::max(1, std::cmp::max(2, 3));
```

As a summary:

- `$var` captures a value in a pattern.
- `$var:ident` specifies a type (could be `ident`, `expr`, `ty`, etc.).
- `$($(var:pat),*)` or `$($(var:pat),+)` captures repetitive sequences separated by commas.
- `$var` is replaced during macro expansion.

End of TD5