**Understanding Pattern Matching in Rust**

[[Claude Barde](https://claudebarde.medium.com/?source=post_page-----763f38785837--------------------------------)](https://claudebarde.medium.com/?source=post_page-----763f38785837--------------------------------)

[Claude Barde](https://claudebarde.medium.com/?source=post_page-----763f38785837--------------------------------)

·

Follow

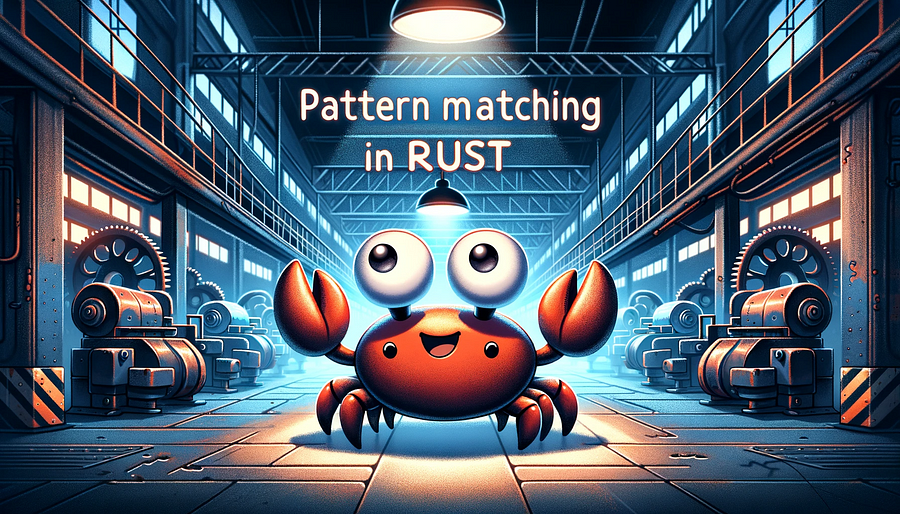
7 min read

·

1 day ago

26

Pattern matching is an amazing feature of the Rust language, read this to learn more about it!



**What is Pattern Matching?**

Pattern matching is a powerful and versatile feature in many programming languages that also exists in Rust. It is one of my favourite features of the language that I discovered first in OCaml and it’s one of the pillars of functional programming.

At its core, pattern matching allows developers to check a given value against a series of patterns and, depending on the match, execute code accordingly. This means that pattern matching focuses more on the shape of the data rather than on the data itself.

*This article assumes that you have a beginner-level understanding of Rust and will delve into the syntax and various applications of pattern matching.*

**The syntax of Pattern Matching in Rust**

In Rust, pattern matching is done using the match keyword. The basic syntax looks like this:

match value {  
 pattern1 => {/\* code to execute if value matches pattern1 \*/},  
 pattern2 => {/\* code to execute if value matches pattern2 \*/},  
 ...  
 \_ => {/\* code to execute if none of the above patterns match \*/}  
}

Between the curly braces, you find the different shapes the value can have and after the => you find the code that is executed if the pattern is matched. The code can be a single expression or a block of code surrounded by curly braces.

The \_ pattern is a catch-all that matches anything if none of the previous patterns match.

Pattern matching is a powerful feature as Rust will check for the exhaustiveness of the patterns, i.e it will make sure that all the patterns are handled and will warn you if you forgot to handle one.

**Examples of Pattern Matching**

**1. Matching numbers:**

let number = 3;  
  
match number {  
 1 => println!("One"),  
 2 => println!("Two"),  
 3 => println!("Three"),  
 \_ => println!("Something else"),  
}

**Functionality:**

* This example checks the value of number and prints the corresponding word for the number (if it's between 1 and 3).
* The catch-all pattern \_ serves as a default case for any number not specifically matched, i.e beyond 3. If this was missing, Rust would raise an error and tell you that the pattern is not exhaustive.

**Advantage:**

* Using pattern matching here simplifies the logic compared to multiple if-else statements. It's more concise and readable, especially for a fixed range of values.

**2. Matching strings**

let day = "thursday";  
  
match day {  
 "monday" => println!("first day of the week"),  
 "tuesday" => println!("second day of the week"),  
 "wednesday" => println!("third day of the week"),  
 "thursday" => println!("fourth day of the week"),  
 "friday" => println!("fifth day of the week"),  
 "saturday" => println!("sixth day of the week"),  
 "sunday" => println!("seventh day of the week"),  
 \_ => println!("this is not a valid day"),  
}

**Functionality:**

* This matches the string day against seven possibilities representing each day of the week, executing different code based on the match. The catch-all pattern at the end will catch a string that is not a valid day.

**Advantage:**

* Pattern matching is beneficial for handling specific known strings. It’s clearer and more direct than using a series of if-else statements.

**3. Using options**

let some\_option: Option<i32> = Some(5);  
  
match some\_option {  
 Some(number) => println!("Number is {}", number),  
 None => println!("No number"),  
}

**Functionality:**

* This example handles an Option<i32> type, which might contain an integer (Some) or nothing (None).
* The match statement either prints the number (if it exists) or a message saying there’s no number.
* This way of handling the absence of value is safer than using something like undefined or nil in other languages as None clearly indicates that the value is absent but the code works as expected, not that another kind of problem happened.

**Advantage:**

* Pattern matching is ideal for Option types as it elegantly handles both cases (Some and None) in a safe, concise manner while keeping your code deterministic.

**4. Matching enums:**

enum Direction {  
 Up,  
 Down,  
 Left,  
 Right,  
}  
  
let dir = Direction::Up;  
match dir {  
 Direction::Up => println!("Going up!"),  
 Direction::Down => println!("Going down!"),  
 Direction::Left => println!("Going left!"),  
 Direction::Right => println!("Going right!"),  
}

**Functionality:**

* The example defines an enum Direction with four variants.
* The match statement checks the value of dir and prints a message corresponding to the direction.
* Enums are preferable to strings for pattern matching as the range of values is finite. In this example, only 4 values are possible, if we had used "up", "down", "left", "right", we would have had to use at least the catch-all pattern at the end and handle a case that might never happen.

**Advantage:**

* Pattern matching is particularly useful with enums as it allows handling each variant distinctly and clearly in one block of code, ensuring that all cases are addressed.

Enums are an even more powerful pattern as they can hold complex data as illustrated by this example:

enum WebEvent {  
 // A page load event with URL as a String.  
 PageLoad(String),  
 // A page unload event with URL as a String.  
 PageUnload(String),  
 // A key press event with the key code as u32.  
 KeyPress(u32),  
 // A mouse click event with x and y coordinates.  
 MouseClick { x: i64, y: i64 },  
 // No data is needed for a screen refresh.  
 ScreenRefresh,  
}  
  
let event = WebEvent::KeyPress(32); // Example event  
  
match event {  
 WebEvent::PageLoad(url) => println!("Page loaded: {}", url),  
 WebEvent::PageUnload(url) => println!("Page unloaded: {}", url),  
 WebEvent::KeyPress(code) => println!("Key pressed: {}", code),  
 WebEvent::MouseClick { x, y } => println!("Mouse clicked at: ({}, {})", x, y),  
 WebEvent::ScreenRefresh => println!("Screen refreshed"),  
}

**Functionality:**

* Enum definition:  
  WebEvent has five variants, each representing a different type of event. Variants like PageLoad and PageUnload contain a String, KeyPress contains a u32, and MouseClick contains two i64 values wrapped in a struct for x and y coordinates.
* Pattern matching:  
  In the match statement, each arm corresponds to a different variant of WebEvent.  
  For PageLoad and PageUnload, it prints the URL.  
  For KeyPress, it prints the key code.  
  For MouseClick, it destructures the struct to get x and y coordinates and prints them.  
  For ScreenRefresh, which carries no additional data, it simply prints a message.

**Advantage:**

* Pattern matching on enums with values allows for succinctly handling different types of data encapsulated within each enum variant. By directly deconstructing each variant, the code becomes cleaner and more readable compared to using nested if-else statements or other methods. This approach also ensures that all possible cases (variants of the enum) are handled, making the code robust and exhaustive.

**5. Complex patterns:**

let pair = (0, -2);  
  
match pair {  
 (0, y) => println!("Y axis: {}", y),  
 (x, 0) => println!("X axis: {}", x),  
 \_ => println!("Somewhere on the plane"),  
}

**Functionality:**

* This example deals with a tuple pair containing two integers.
* The match checks if either of the integers is zero and identifies which axis the pair lies on, or otherwise acknowledges it's somewhere on the plane.

**Advantage:**

* Complex pattern matching like this is extremely useful for destructuring and handling various data types. It’s more efficient and readable than nested if-else statements, especially when dealing with multi-component data structures like tuples.

let array = [1, 2, 3];  
  
match array {  
 [0, ..] => println!("Array starts with 0"),  
 [1, 2, 3] => println!("Array contains 1, 2, 3"),  
 [\_, \_, \_] => println!("Array has three elements"),  
}

**Functionality:**

* Prefix match:  
  [0, ..] uses the .. pattern to match any array starting with 0. If array starts with 0 (like [0, 4, 5]), it prints "Array starts with 0".
* Exact match:  
  [1, 2, 3] matches an array that exactly contains 1, 2, 3. If array is [1, 2, 3], it prints "Array contains 1, 2, 3".
* Length match:  
  [\_, \_, \_] matches any array with exactly three elements, regardless of their values. If array has three elements like [7, 8, 9], it prints "Array has three elements". This acts more or less as a catch-all pattern, as the length of the array is known before pattern matching it.

**Advantage:**

* Pattern matching over arrays can be a powerful tool to validate the shape of the array or the elements it contains without resorting to more complex methods of traversing the array like fold or map.

**Limitations and Consideration:**

Rust’s pattern matching on arrays is limited because we need to know the size of the array at compile time. The patterns must account for the array’s length, which can be restrictive compared to other data types like vectors, where the length can be dynamic. For small and fixed-size arrays, pattern matching can offer a concise and readable way to handle array-based logic, but it becomes more cumbersome to use for longer arrays.

**Why Prefer Pattern Matching over If Conditions**

Pattern matching is a powerful feature because it’s not just about checking equality — it can destructure data types like tuples or enums to extract values directly.

This makes code more concise, readable, and less error-prone compared to a series of if statements, especially when dealing with complex data structures.

Pattern matching also ensures that all possible cases are handled, either specifically or through a default case, making your code more robust.

In summary, pattern matching in Rust provides a clear, concise, and powerful way to handle conditional logic.

It shines particularly when working with Rust’s various data types like enums and options, allowing developers to write more readable and maintainable code.

Whether you’re dealing with simple values or complex data structures, pattern matching can greatly simplify your Rust code.

**Final words**

Pattern matching in Rust offers a structured and elegant way to handle different types of data and conditions.

Its ability to destructure and match against different data types and structures makes it a preferable choice over traditional conditional statements in many cases.

This leads to code that is not only more readable and maintainable but also ensures comprehensive handling of all possible cases in a safe and robust manner.

[Rust](https://medium.com/tag/rust?source=post_page-----763f38785837---------------rust-----------------)

[Programming](https://medium.com/tag/programming?source=post_page-----763f38785837---------------programming-----------------)

[Coding](https://medium.com/tag/coding?source=post_page-----763f38785837---------------coding-----------------)

[Functional Programming](https://medium.com/tag/functional-programming?source=post_page-----763f38785837---------------functional_programming-----------------)

[Development](https://medium.com/tag/development?source=post_page-----763f38785837---------------development-----------------)