Pattern Matching in Rust

Lecture 4

Slides: Neil Kaushikkar

Goals For Today

- Strings in Rust
- **match** statements
- Control Flow with **match**

Course Announcements

- HW1 released yesterday due 2/7 at 11:59 pm CT
- MP0 releasing today due 2/15 at 11:59 pm CT
- Homeworks will have a “Feedback Survey” to ask or say anything!

Strings in Rust

- **&str** vs **String**
- We will have a lecture dedicated to why there is a difference after the lectures on Rust's memory management mechanism
- Generally, you can modify (add to) a variable of type **String** in place (but not one of type **&str**)
- Both types have functions to parse, trim, extract, split, etc...
- You might see **&String** - think of this as a pointer to a **String**
    - More on this next week...

Creating Strings

- **&str**
  - Just type out your string surrounded by double quotes
  - Ex: **let y = “CS 128 Honors”;**
- **String**
  - **String::from(“hello”)**
  - **“hello”.to_string()**
  - **String::new()** - Creates an empty string
- Converting variables/values of numeric (and many other) types to **String**s
  - **<value>.to_string()**
  - Ex: **5.to_string()**
  - Ex: **let x = 3.1415; x.to_string()**

Converting Between Types

- **String** to **&str**
  - **.as_str()**
  - There are many more ways, but you will understanding those after the next couple lectures
- **&str** to **String**
  - **String::from(“hello”)**
  - **“hello”.to_string()**

Format Strings

- Use the **format!(...)** macro to compose **String**s from other values
- Ex: **format!("First arg goes here: {} and second goes here: {}", arg1, arg2);**
- Use curly brackets in the format string to specify where to place arguments
- Arguments are placed in chronological order**
    - There are more rules and conventions about variable placement, but you can read those on your own at the link at the bottom of the slide

Read More:

- https://doc.rust-lang.org/std/fmt/

The match Keyword

- Allows you to:
  - Compare a value against a series of patterns
  - Then execute code based on which pattern matches
  - Each case in your match statement is called an “arm”
- Patterns can be made up of literals, variable names, wildcards, more…
- Similar to the switch functionality in other languages
- **match** statements are an expression which means they *can* evaluate to a value, so we can assign the result of the **match** to some variable

Reference:
- https://doc.rust-lang.org/book/ch06-02-match.html
- https://doc.rust-lang.org/rust-by-example/flow_control/match.html

Examples of match

```rust
let course = get_course_number();

let professor = match course {
    124 => "Prof. Challen",
    128 => "Prof. Nowak",
    173 => "Prof Fleck",
    225 => "Prof Evans",
    _ => ""
};

println!("{} teaches CS {}", professor, course);
```

```rust
let call: String = get_random_call();

let response = match call.as_str() {
    "ILL" => "INI!",
    "To infinity" => "And beyond!",
    "Hakuna" => "Matata!",
    "Marco" => "Polo!"
    _ => "I don't know how to respond to that"
};

println!("{}", response);
```

IMPORTANT: &str is a pattern BUT Strings are NOT patterns!
Use a conversion method like .as_str() to get a &str from a String

match Rules

- The patterns you try to **match** **must** be **exhaustive**
  - **match** arms must cover all possibilities of the data type being matched
- Match **arms** a comma-separated list where each arm is composed as follows:
  - pattern => expression
  - Expressions are any piece of code that **evaluates** to some value
  - Expressions can be a some value, any code within curly braces, another match statement, an if statement, a loop, etc...
- **ALL** match arms that evaluate to a type must evaluate to the same type*
  - Some (or all) match arms may also **return** from the current function

Reference:
- https://doc.rust-lang.org/book/ch06-02-match.html
- https://doc.rust-lang.org/rust-by-example/flow_control/match.html

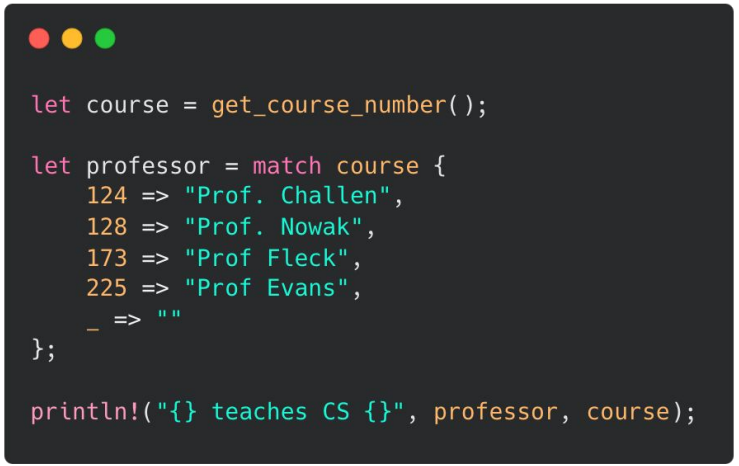
match Wildcards

- **REMEMBER:** The patterns you try to **match** **must be exhaustive**
  - We use the wildcard, an underscore, (the _ symbol) to match all of the remaining possibilities not covered by the patterns previously listed
- We also use the wildcard to indicate that we do not care about a specific value
  - **This will come up in tomorrow's lecture on **enum**s

Reference:

- https://doc.rust-lang.org/book/ch06-02-match.html
- https://doc.rust-lang.org/rust-by-example/flow_control/match.html

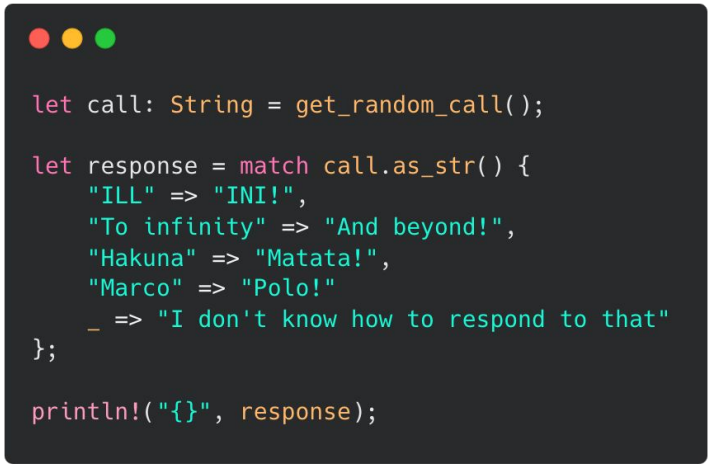
match Wildcards



```rust
let course = get_course_number();

let professor = match course {
    124 => "Prof. Challen",
    128 => "Prof. Nowak",
    173 => "Prof Fleck",
    225 => "Prof Evans",
    _ => ""
};

println!("{} teaches CS {}", professor, course);
```



```rust
let call: String = get_random_call();

let response = match call.as_str() {
    "ILL" => "INI!",
    "To infinity" => "And beyond!",
    "Hakuna" => "Matata!",
    "Marco" => "Polo!"
    _ => "I don't know how to respond to that"
};

println!("{}", response);
```

Tuple Destructuring & Variable Binding

- If the type we are matching can be decomposed, we can choose to match all or specific parts of the type
  - Applies to tuples, structs, enums (but more on those in future lectures)
- We use the wildcard (the _ symbol) to indicate that we do not care about matching a **specific part** of the data type
- We use the range notation, 2 periods, (the .. notation) to indicate that we do not care about matching the **portion that the range covers** in the data type
- We can bind values in the match statement to variables when we want to continue using the matched value
- Variables act just like the wildcard
  - They cover the "remaining" patterns)
  - BUT we can then go on to use the value bound to the variable in some code snippet

Complex Matching - Destructuring & Binding


```
match triple {
    (1, 2, 3) => println!("Got 1, 2, 3"),
    (_, 2, 3) => println!("Ends in 2 and 3"),
    (42, _, 42) => println!("Meaning of life"),
    (199, 128, _) => println!("CS 128 Honors!"),
    (128, ..) => println!("We only care that the first item is 128"),
    (.., 2002) => println!("We only care that the last item is 2002"),
    (a, 1, 1) => println!("got {} and two 1s", a),
    (x, y, z) => println!("triple adds to {}", x + y + z)
}
```

More match Syntax

- Use **value1 ..= value2** to match an **inclusive** range of values
- Use a vertical bar (the | symbol) to match any pattern separated by the bar and execute the same expression for each pattern
- Use the **<variable> @ <expression>** syntax to bind a variable to some range of values
  - Ex: **num @ 0..=100 => …** to match the case the some number is between 0 and 100 (inclusive) and bind that value to the variable "num"
  - May need to use parentheses:
  - Ex: **small_prime @ (2 | 3 | 5 | 7) => …** binds the variable "small_prime" to the either 2, 3, 5, or 7, depending on the value being matched

Complex Matching

```
let msg: String = match course {
    0 ..= 99 => "INVALID NUMBER".to_string(),
    128 | 225 | 341 => "Teaches C or C++".to_string(),
    100 ..= 199 => "100 Level".to_string(),
    level @ 100 ..= 399 => {
        let hundreds_digit: u32 = level / 100;
        format!("{}00 level course", hundreds_digit)
    },
    num @ 400 ..= 499 => if num == 461 {
            "My favorite class".to_string()
        } else {
            "Upper level electives".to_string()
        },
    500 ..= 599 => "Graduate level".to_string(),
    n => format!("CS {} is not a value course!", n)
};
```

That’s All Folks!