Enums & Matching in Rust

Lecture 5

Slides: Neil Kaushikkar

Goals For Today

- Review **match** statements
- Introduction to Enums
- **Result** (Ok/Err) & **Option** (Some/None)
- Control Flow with Enums and **match**

Course Announcements

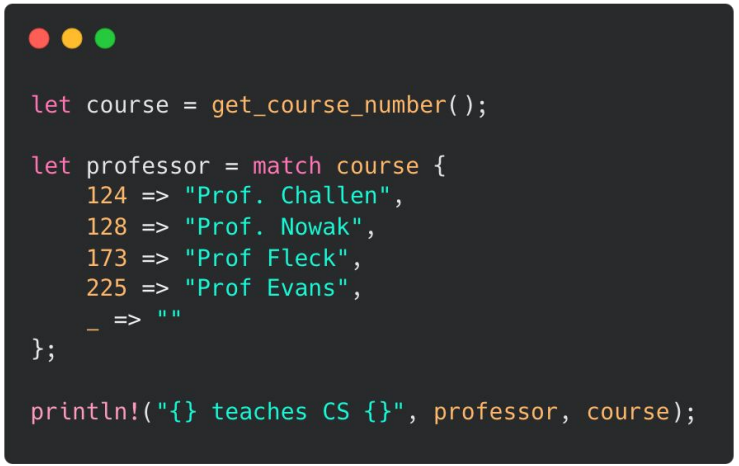
- HW1 due 2/7 at 11:59 pm CT
- HW2 releasing today due 2/9 at 11:59 pm CT
- MP0 released yesterday due 2/15 at 11:59 pm CT

The match Keyword

- Allows you to:
  - Compare a value against a series of patterns
  - Then execute code based on which pattern matches
- Patterns can be made up of literals, variable names, wildcards, more...
- The patterns you try to **match** **must** be **exhaustive**

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

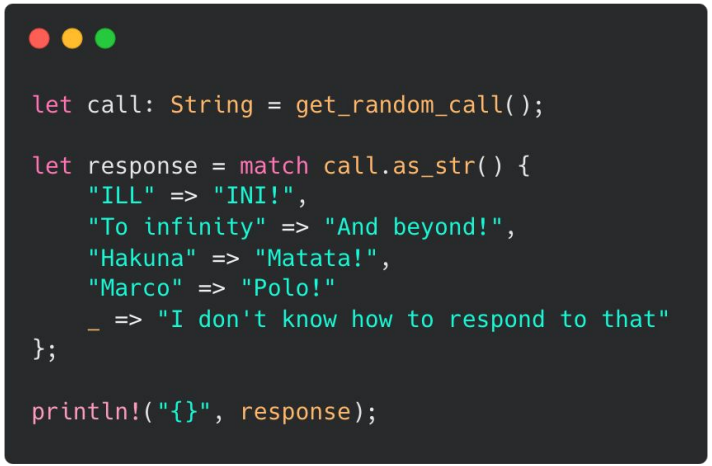
Examples of match



```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);
```

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)
}
```

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)
};
```

What are Enums?

- Custom types with a restricted set of values
  - Colors of the rainbow
  - Undergraduate student level
  - Day of the week
  - HTTP methods/HTTP Response Codes
- They make your life a whole lot easier
- Enums are also considered patterns (we can **match** them!)

Why do we enums?

- Take the “day of the week” example
  - Representing the day with a String? Integer?
    - “Monday”, “Tuesday”, ...
    - i.e. 1 is Monday, 2 is Tuesday, ...
  - What happens if a variable containing our “day” does not match what we expect it to be?
  - Do we throw an error? Do we crash the program?
  - Enums prevent this by restricting the values a type can take on to some small set of values specified by the programmer

Defining Custom Enums

- The **enum** keyword!

```
enum DayOfWeek {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

```
fn main() {
    let today = DayOfWeek::Thursday;
}
```

Reference:

- https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

Matching Enums

```rust
enum DayOfWeek {
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}
```

```rust
match day_of_week {
    DayOfWeek::Monday => "UGH!",
    DayOfWeek::Tuesday | DayOfWeek::Thursday => "128 Honors Lecture Drop!",
    DayOfWeek::Saturday | DayOfWeek::Sunday => "Weekend!",
    _ => "Weekday"
}
```

Reference:

- https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

The Option Enum

- From the docs: Type **Option** represents an optional value: every **Option** is either **Some** and contains a value, or **None**, and does not.
- Return values for functions that are not defined over their entire input range
- Similar usage as returning null/nullptr in Java/C++
  - No more NullPointerExceptions (!!)
  - Kind of...

```rust
let course = get_course_number();

let professor: Option<&str> = match course {
    124 => Some("Prof. Challen"),
    128 => Some("Prof. Nowak"),
    173 => Some("Prof Fleck"),
    225 => Some("Prof Evans"),
    _ => None
};

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

Reference:

- https://doc.rust-lang.org/std/option/
- https://doc.rust-lang.org/std/vec/struct.Vec.html#method.get

Option in Action!

The Result Enum

- From the docs: Type **Result<T, E>** is used for returning & propagating errors
- It is an enum has 2 variants
  - **Ok(T)** representing success & containing a value
  - **Err(E)** representing error & containing an error value.
- Functions return **Result** whenever errors are expected and recoverable
- In the **std** crate, **Result** is most prominently used for I/O
- If there is no meaningful value to be returned as **T** or **E**, we can use the unit type **()** in place of the success or error value
  - Ex: **Result<(), String>**

Reference:

- https://doc.rust-lang.org/std/result/
- https://doc.rust-lang.org/std/fs/struct.File.html#method.open

Result in Action!

Useful Methods on Option & Result

- **is_some()** / **is_ok()** : Check if the variable of type (Option / Result) contains a value corresponding to some successful operation
- **is_none()** / **is_err()** : Check if the operation returning the variable of type (Option / Result) failed
- **unwrap_or(default: T)** : Give me the value corresponding to success, otherwise, return some default value (**default**).

Reference:

- https://doc.rust-lang.org/std/option/
- https://doc.rust-lang.org/std/result/

USE THESE WITH CAUTION

- **expect(msg: &str)** : We are 100% sure that the operation succeeded, so give me the value corresponding to success. **Panic** if the operation failed and print out a useful error message (**msg**)!
- **unwrap()** : We are 100% sure that the operation succeeded, so give me the value corresponding to success. **Panic** if the operation failed!
  - Avoid as much as possible
  - Difficult and very annoying to debug

Reference:

- https://doc.rust-lang.org/std/option/
- https://doc.rust-lang.org/std/result/

Matching Option and Result

- You can compare some value to a series of patterns, then execute some code based on which pattern **match**es
- The patterns you **match** must be exhaustive
- Patterns for **Option<T>**:
  - **Some(T)**
  - **None**
- Patterns for **Result<T, E>**:
  - **Ok(T)**
  - **Err(E)**

```
match my_option {
    Some(val) => println!("{}", val),
    None => println!("Nothing here!")
};
```

```
match my_result {
    Ok(val) => println!("succeeded: {}!", val),
    Err(e) => println!("something went wrong: {}!", e)
};
```

Reference:

- https://doc.rust-lang.org/std/option/
- https://doc.rust-lang.org/std/result/

Matching Result

Tuple Enums

- Rust allows you to bundle additional information to your **enum** states
- We can create named tuples using enum variants

```
enum Point {
    TwoD(f64, f64),
    ThreeD(f64, f64, f64),
    FourD(f64, f64, f64, f64)
}
```

```
fn main() {
    let pt_a = Point::TwoD(5.0, 4.0);
    let pt_b = Point::ThreeD(1.0, 2.0, 8.0);
    let pt_c = Point::FourD(3.0, 9.0, -1.0, 6.0);
}
```

Reference:

- https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

Struct Enums

- We can assign more meaning to our **enum** states using **struct** declarations
- **struct** are similar to tuples:
  - Like tuples, the pieces of a **struct** can be different types
  - Unlike tuples, you name each piece of data so it's clear what values mean
  - As a result, **struct**s are more flexible than tuples
- (more on **struct**s later in the course)

```
enum MouseEvent {
    Drag { from: (i64, i64), to: (i64, i64) },
    Click { x: i64, y: i64 }
}
```

```
fn main() {
    let drag = WebEvent::Drag{ to: (128, 196), from: (0, 0) };
    let click = WebEvent::Click{ x: 128, y: 196 };
}
```

Reference:

- https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html
- https://doc.rust-lang.org/book/ch05-01-defining-structs.html

Mixing and Matching Variant Types

```
enum WebEvent {
    PageLoad,
    PageUnload,
    KeyPress(char),
    Paste(String),
    Click { x: i64, y: i64 },
}
```

```
fn main() {
    let load = WebEvent::PageLoad;
    let unload = WebEvent::PageUnload;
    let press = WebEvent::KeyPress('c');
    let paste = WebEvent::Paste("hello".into());
    let click = WebEvent::Click{ x: 128, y: 196 };
}
```

Reference:

- https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

That’s All Folks!