CS128 Honors

Lecture 4

Enums and Matching in Rust

Slides: Neil Kaushikkar

Goals For Today

- Introduction to Enums
- **Result** (Ok/Err) & **Option** (Some/None)
- Control Flow with Enums and **match**
- Custom Enums in Rust
- Introduce HW1 and MP0

Course Announcements

- HW1 due 2/8 at 11:59 pm CT
- HW2 releasing today due 2/10 at 11:59 pm CT
- MP0 releasing today due 2/11 at 11:59 pm CT

- We are more than happy to grant extensions when requested, but we do require that you've made some progress on the HW or MP (unless you have some valid excuse)
- Homeworks will have a "Feedback Survey" to ask or say anything!

Homework 1 Correction

The list of items is: computer, pizza, bread, Welby, panda, pancake, Eustis, giraffe, cat, Neil, Spiderman, Interstellar, banana, television, microwave, spaghetti, elephant, Ferris

THE NAME'S SPIDER-MAN--
--AND DON'T FORGET THE HYPHEN!
THWIPP!

What are Enums?

- Custom types with a restricted set of values
    - Colors of the rainbow
    - Undergraduate student level
    - Day of the week
- They make you (the programmer) and your life easier

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...

Reference:

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

CS128 Honors

Option in Action!

The Result Enum

- From the docs: Type **Result<T, E>** is used for returning & propagating errors. It is an enum with the variants, **Ok(T)**, representing success & containing a value, and **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.

Reference:

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

CS128 Honors

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()** : We are 100% sure that the operation succeeded, so give me the value corresponding to success. **Panic** if the operation failed!

Reference:

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

More Useful Methods on Option & Result

- **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_or(default: T)** : Give me the value corresponding to success, otherwise, return some default value (**default**).

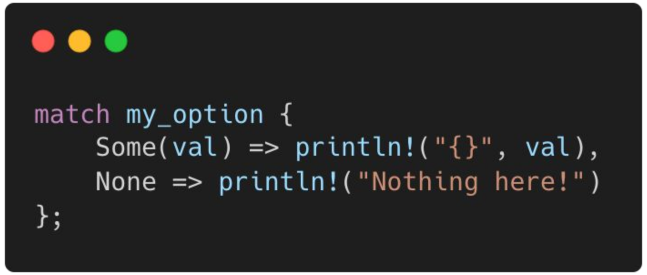
Reference:

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

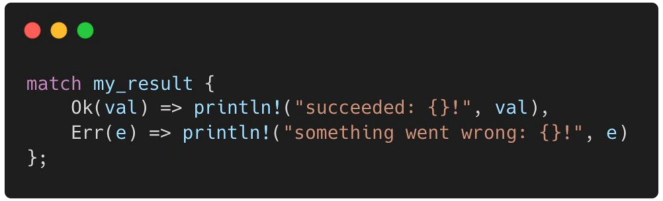
More Option Examples!

Matching Option and Result

- You can compare some value to a series of patterns, then execute some code based on which pattern **matches**
- 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!")
};
```



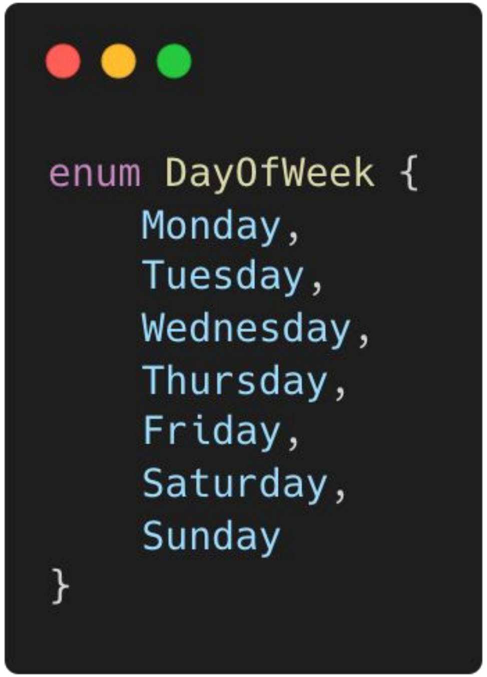
```
match my_result {
    Ok(val) => println!("succeeded: {}!", val),
    Err(e) => println!("something went wrong: {}!", e)
};
```

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

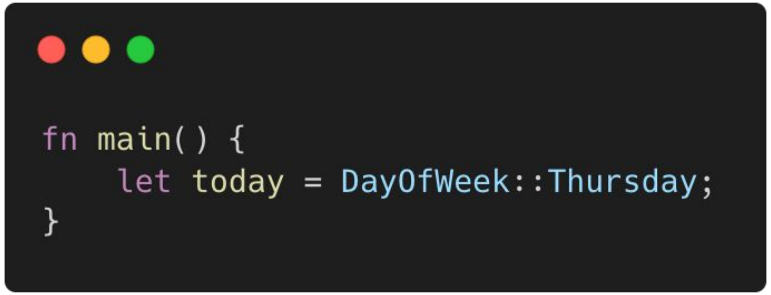
Matching Option & Result

Custom Enums

- The **enum** keyword!



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

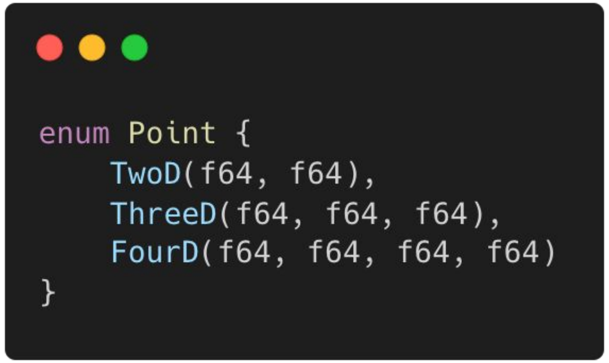


```rust
fn main() {
    let today = DayOfWeek::Thursday;
}
```

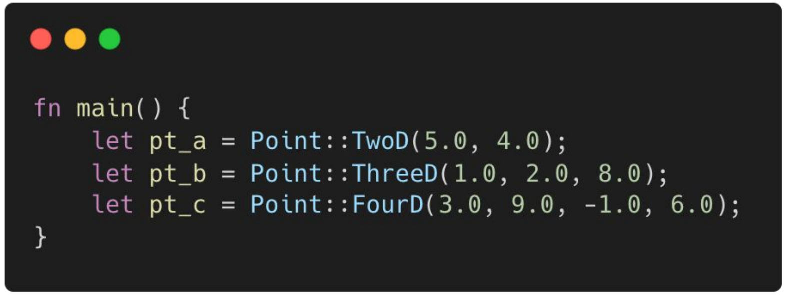
Reference:
- https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

Tuple Enums

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



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

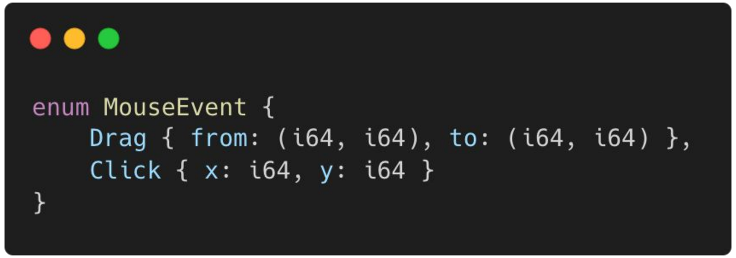


```rust
fn main() {
    let pt_a = Point::TwoD(5.0, 4.0);
    let pt_b = Point::ThreeD(1.0, 2.0, 8.0);
    let pt_c = Point::FourD(3.0, 9.0, -1.0, 6.0);
}
```

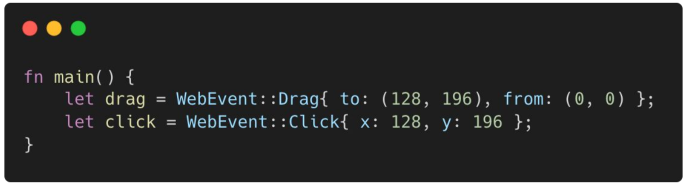
Reference:
- https://doc.rust-lang.org/book/ch06-01-defining-an-enum.html

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)



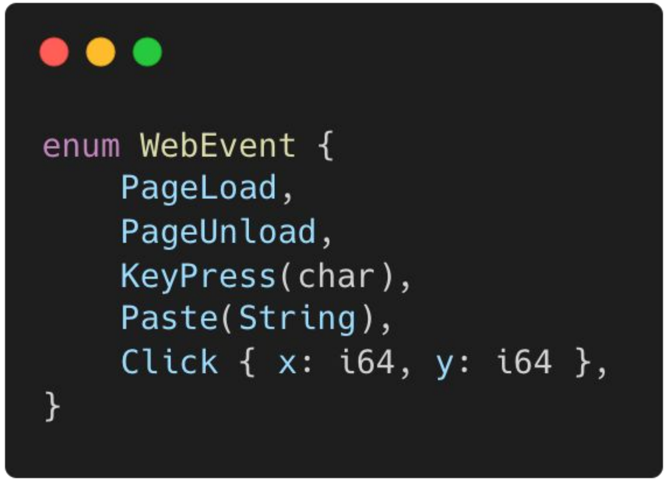
```rust
enum MouseEvent {
    Drag { from: (i64, i64), to: (i64, i64) },
    Click { x: i64, y: i64 }
}
```



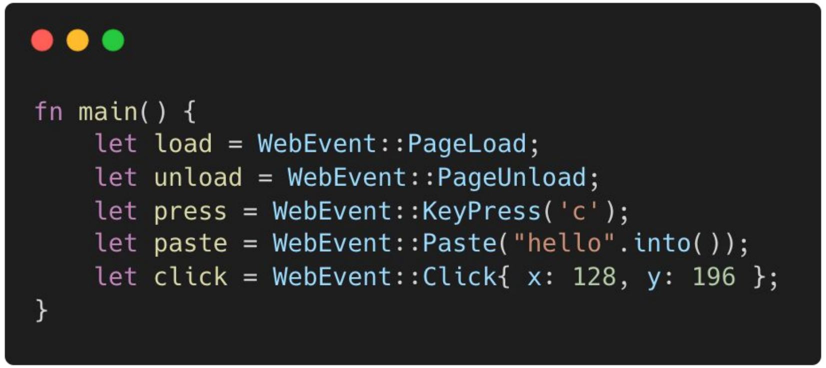
```rust
fn main() {
    let drag = WebEvent::Drag{ to: (128, 196), from: (0, 0) };
    let click = WebEvent::Click{ x: 128, y: 196 };
}
```

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



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



```rust
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

CS128
Honors

Order Status Example

Testing Your Code

- Create a module for your tests with the **mod** keyword
  - Mark the module with the **#[cfg(test)]** procedural macro
- Create functions within the test module to test out your code
  - Mark each test function with the **#[test]** procedural macro
- Use assertion function-like macros to check for expected behavior
  - **assert!(statement: bool)** - check if some condition is true
  - **assert_eq!(value1: T, value2: T)** - check if two values are equal
  - **assert_neq!(value1: T, value2: T)** - check if two values are NOT equal
- Assertions will **panic** when they fail. Panics in tests indicate that the test failed.

Reference:

- https://doc.rust-lang.org/book/ch11-01-writing-tests.html

Testing Our Example

HW1 and MP0 Walkthrough