Lecture 4

Enums & Matching

Goals For Today

- Match review
- Intro to Enums
- Special Enums: Result (Ok/Err) and Option(Some/None)
- Control flow with Enums and Match

Match

- Expressions that allow you to:
  - Compare a value against a series of patterns
  - Execute code based on which pattern matches
- Patterns:
  - Can be literals, variable names, wildcards, and more
  - Must be exhaustive

```
let course: i32 = 128;

let professor: &str = match course {
    124 => "Prof. Challen",
    128 => "Prof. Nowak",
    173 => "Prof. Fleck",
    225 => "Prof. Solomon",
    _ => "Idk bro"
};
// professor is now "Prof. Nowak"
```

Enums - A motivating example

- Suppose we want to model days of the week
- A defined, restricted set of values
- How should we represent it?
  - Integers: 1->7. What if the user inputs 10?
  - Strings: “Monday, Tuesday..”. What if the user inputs “Manday”?
- How do we handle these errors?
- Solution: Make the errors impossible to occur
- Enter Enums

What are Enums?

- Custom types with a restricted set of values
  - Days of the week (7 values)
  - Undergraduate student level (4 values)
  - Colors of the rainbow (7 values)
- Also called “sum types”
- Each value (Monday, Tuesday..) called a variant
- Can contain subvalues.

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

```
let thurs: DayOfWeek = DayOfWeek::Thursday;
```

```
enum Groups {
    Single(String),
    Pair(String, String),
    Trio(String, String, String),
    Gang(Vec<String>)
}
```

Matching on Enums

- Much like integers, &str, etc.. enums variants are also patterns

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

```
match today {
    DayOfWeek::Monday => println!("UGH!"),
    DayOfWeek::Saturday | DayOfWeek::Sunday => println!("Yay weekend"),
    _ => println!("Weekday"),
};
```

- Reminder: patterns must be exhaustive (must consider all variants, or defer to wildcard)

Enums (continued)

- What if we also want to model the number of classes on each day of the week?
- Enums variants can contain values

```
enum DayOfWeek {
    Monday(i8),
    Tuesday(i8),
    Wednesday(i8),
    Thursday(i8),
    Friday(i8),
    Saturday,
    Sunday,
}
```

```
let thurs: DayOfWeek = DayOfWeek::Thursday(3);
```

- Weekend don't have to contain values because we don't have classes

Matching on Enums (continued)

- Variants' values are also patterns
- Most common way to extract values from an enum variable
- Patterns must be exhaustive -> to extract values, need to consider all patterns

```
enum DayOfWeek {
    Monday(i8),
    Tuesday(i8),
    Wednesday(i8),
    Thursday(i8),
    Friday(i8),
    Saturday,
    Sunday,
}
```

```
match today {
    DayOfWeek::Monday(num_classes: i32) => println!("I have {} classes on Monday", num_classes),
    DayOfWeek::Tuesday(3) => println!("I have 3 classes on Tuesday"),
    DayOfWeek::Tuesday(4) => println!("I have 4 classes on Tuesday"),
    DayOfWeek::Tuesday(_) => println!("I have some other numbers of classes on Tuesday"),
    DayOfWeek::Saturday | DayOfWeek::Sunday => println!("Yay weekend"),
    _ => println!("Weekday"),
};
```

The Option enum

- Consider pointers from languages like C++
- Can either be a valid pointer (points to an address of an object), or a null pointer (doesn't point to anything)
- When using, developer must remember to check if pointer is NULL
- What if we have a type that forces us to perform this check?
- Sounds like the job for Enums and pattern matching

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

The Option enum (continued)

- A special, built in enum representing an optional value: variants of Option are Some (contains a value) or None (contains no value).
- Think weekday vs weekend variants from the DayOfWeek enum.
- Return values for functions that are not defined over the entire input range
- Note: Don't need Option::Some or Option::None because this enum is built in

```
let course: i32 = get_course_number();

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

The Result enum

- Consider exceptions from C++
- Used for operations where failure can occur
- May forget to catch exceptions. Bad things happen
- What if we have a type that forces us to catch these failures?
- Enums to the rescue! (again)

The Result enum (continued)

- A special, built in enum representing fallible operations, variants of Result are:
  - Ok(T) represents success, contains the operation's value
  - Err(E) represents failure, contains a value indicating the error
- A function returns a Result if failure is expected and recoverable
- In the standard library: most commonly used for I/O
- If no meaningful value for T or E: use the unit type () as a placeholder

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

Useful methods for Option and Result

- is_some()/is_ok(): Checks if Option/Result represents successful operations
- is_none()/is_err(): Checks if Option/Results represents failed operation
- unwrap_or(T): extracts the value from the successful operation, otherwise returns a default (value T)

Matching with Option & Result


```
let my_grade: String = match grades.get(0) {
    Some(val) => val.to_string(),
    None => "I'm not enrolled?".to_string()
};
```

```
match return_value: Result<String, String> {
    Ok(ret) => println!("Function returned: {}", ret),
    Err(err) => eprintln!("Function returned an error: {}", err)
};
```

USE THESE WITH CAUTION

- Or avoid if possible
- expect(msg: &str) - We are 100% sure the operation succeeded, so give me the success value. Panic if the operation failed and prints out msg.
- unwrap() - We are 100% sure the operation succeeded, so give me the success value. Panic if operation failed.