

PLC 2025, Lecture 13, 25 February 2025

Enumerated types

- Simplest form, choice of constants

```
In [2]: enum IpAddrKind {  
        V4,  
        V6,  
      }  
  
let four = IpAddrKind::V4;  
let six = IpAddrKind::V6;
```

```
In [3]: four
```

```
[E0277] Error: `IpAddrKind` doesn't implement `Debug`
```

- Add special directive to get around the `Debug` issue

```
In [4]: #[derive(Debug)]  
enum IpAddrKind {  
        V4,  
        V6,  
      }  
  
let four = IpAddrKind::V4;  
let six = IpAddrKind::V6;
```

```
In [5]: four
```

```
Out[5]: V4
```

- Options can be parameterized

```
In [6]: #[derive(Debug)]  
enum IpAddr {  
        V4(String),  
        V6(String),  
      }  
  
let home = IpAddr::V4(String::from("127.0.0.1"));  
let loopback = IpAddr::V6(String::from("::1"));
```

```
In [7]: loopback
```

```
Out[7]: V6("::1")
```

- Options can have different types of associated values

```
In [10]: #[derive(Debug)]
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

```
In [11]: home
```

```
Out[11]: V4(127, 0, 0, 1)
```

- Like a `struct`, can attach functions to an `enum`

```
In [12]: #[derive(Debug)]
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

impl Message {
    fn call(&self) {
        // method body would be defined here
    }
}
```

- Typical usage

```
In [13]: let m = Message::Write(String::from("hello"));
m.call();
```

Option type

- A type that can hold a value of type `T`, or be undefined

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

- Like `Maybe` in Haskell
- Examples (`Option` is a built-in enum)
- Note that `None` has to be explicitly typed

```
In [14]: let some_number = Some(5);
let some_char = Some('e');
```

```
In [15]: let absent_number = None;
```

```
[E0282] Error: type annotations needed for `Option<_>`
[command_15:1:1]
1 | let absent_number = None;
  |                   ^
  |                   |
  |                   | error: type annotations needed for `Option
  |                   |
  |                   | help: consider giving `absent_number` an e
  |                   | xplicit type, where the type for type parameter `T` is specified: `: Option
  |                   | n<T>`
  |                   |
  |                   | type must be known at this point
```

```
In [16]: let absent_number: Option<i32> = None;
```

- Can't mix `T` and `Some(T)` in an expression

```
In [17]: let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y;
```

```
Error: consider importing this trait
[E0277] Error: cannot add `Option<i8>` to `i8`
[command_17:1:1]
4 | let sum = x + y;
  |           ^
  |           |
  |           | no implementation for `i8 + Option<i8>`
```

- Need to `unwrap()` the inner value from `Some(T)`

```
In [18]: let x: i8 = 5;
let y: Option<i8> = Some(5);

let sum = x + y.unwrap();
```

```
In [19]: y
```

```
Out[19]: Some(5)
```

```
In [20]: y.unwrap()
```

```
Out[20]: 5
```

- Unwrapping `None` is an error

```
In [21]: absent_number.unwrap()
```

```
thread '<unnamed>' panicked at src/lib.rs:158:15:
called `Option::unwrap()` on a `None` value
stack backtrace:
 0: rust_begin_unwind
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/std/src/panicking.rs:665:5
 1: core::panicking::panic_fmt
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/panicking.rs:76:14
 2: core::panicking::panic
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/panicking.rs:148:5
 3: core::option::unwrap_failed
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/option.rs:2009:5
 4: std::panic::catch_unwind
 5: run_user_code_17
 6: evcxr::runtime::Runtime::run_loop
 7: evcxr::runtime::runtime_hook
 8: evcxr_jupyter::main
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Matching

- `match` operator selects amongst optional values in an `enum`

```
In [22]: enum Coin {
        Penny,
        Nickel,
        Dime,
        Quarter,
    }

    fn value_in_cents(coin: Coin) -> u8 {
        match coin {
            Coin::Penny => {
                println!("Lucky penny!");
                1
            }
            Coin::Nickel => 5,
            Coin::Dime => 10,
            Coin::Quarter => 25,
        }
    }
```

```
In [23]: let c = Coin::Penny;
        value_in_cents(c)
```

Lucky penny!

```
Out[23]: 1
```

```
In [24]: let c = Coin::Quarter;
        value_in_cents(c)
```

```
Out[24]: 25
```

- `match` must return a consistent type

```
In [25]: let z = match y {
          None => None,
          Some(x) => y.unwrap(),
        };
```

[E0308] Error: `match` arms have incompatible types

```
[command_25:1:1]
1  let z = match y {
2      None => None,
3      Some(x) => y.unwrap(),
4  };
    this is found to be of type `Option<_>`
    expected `Option<_>`, found `i8`
    `match` arms have incompatible types
```

- This works

```
In [26]: fn plus_one(x: Option<i32>) -> Option<i32> {
          match x {
            None => None,
            Some(i) => Some(i + 1),
          }
        }

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

```
In [27]: five
```

```
Out[27]: Some(5)
```

```
In [28]: six
```

```
Out[28]: Some(6)
```

```
In [29]: none
```

```
Out[29]: None
```

- Can pattern match and operate within an enum
- **All** possible values must be covered -- error tells you which value was missed

```
In [30]: fn plus_one(x: Option<i32>) -> Option<i32> {
          match x {
            Some(i) => Some(i + 1),
```

```
}
}
```

[E0004] Error: non-exhaustive patterns: `None` not covered

```
[command_30:1:1]
2   match x {
3       |
    pattern `None` not covered
    Some(i) => Some(i + 1),
    help: ensure that all possible cases
are being handled by adding a match arm with a wildcard pattern or an explicit pattern as shown: `
    None => todo!()`
```

```
In [31]: fn plus_one(x: Option<i32>) -> Option<i32> {
        match x {
            None => None,
            Some(i) => Some(i + 1),
        }
    }
```

- `match` is not restricted to finite number of options
- `other` is a catch-all pattern (`other` is not a keyword, can use any label)

```
In [32]: let dice_roll = 3;
        match dice_roll {
            3 => add_fancy_hat(),
            7 => remove_fancy_hat(),
            other => move_player(other),
        }

        fn add_fancy_hat() {println!("Add fancy hat");}
        fn remove_fancy_hat() {println!("Remove fancy hat");}
        fn move_player(num_spaces: u8) {println!("Move {} spaces", num_spaces);}
```

Add fancy hat

- If we don't need the value, can use anonymous `_` instead of a label

```
In [33]: let dice_roll = 9;
        match dice_roll {
            3 => add_fancy_hat(),
            7 => remove_fancy_hat(),
            _ => reroll(),
        }

        fn add_fancy_hat() {println!("Add fancy hat");}
        fn remove_fancy_hat() {println!("Remove fancy hat");}
        fn reroll() {println!("Reroll");}
```

Reroll

Panic

- Rust "panics" when it encounters an unrecoverable error at run-time, such as an index out of bounds

```
In [34]: fn main() {
           let v = vec![1, 2, 3];

           v[99];
       }
```

```
In [35]: main()
```

```
thread '<unnamed>' panicked at src/lib.rs:5:6:
index out of bounds: the len is 3 but the index is 99
stack backtrace:
 0: rust_begin_unwind
    at /rustc/9fcb6b43126469e3858e2fe86cafb4f0fd5068869/library/std/src/panicking.rs:665:5
 1: core::panicking::panic_fmt
    at /rustc/9fcb6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/panicking.rs:76:14
 2: core::panicking::panic_bounds_check
    at /rustc/9fcb6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/panicking.rs:281:5
 3: <unknown>
 4: <unknown>
 5: evcxr::runtime::Runtime::run_loop
 6: evcxr::runtime::runtime_hook
 7: evcxr_jupyter::main
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

- Observe what happens if we use an array instead of a vector
- Rust is able to catch the potential panic at compile time!

```
In [36]: fn main() {
           let v = [1, 2, 3];

           v[99];
       }
```

```
[unconditional_panic] Error: this operation will panic at runtime
[command_36:1:1]
4 |         v[99];
  |         ^    index out of bounds: the length is 3 but the index is 99
```

- However, if we obfuscate the array access, we achieve a run-time panic

```
In [37]: fn check(x:usize) {
           let v = [1, 2, 3];

           v[x];
       }
```

```
fn main(){
    let x:usize = 22;
    check(x)
}
```

In [38]: `main()`

```
thread '<unnamed>' panicked at src/lib.rs:16:5:
index out of bounds: the len is 3 but the index is 22
stack backtrace:
 0: rust_begin_unwind
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/std/src/panicking.rs:665:5
 1: core::panicking::panic_fmt
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/panicking.rs:76:14
 2: core::panicking::panic_bounds_check
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/panicking.rs:281:5
 3: <unknown>
 4: <unknown>
 5: evcxr::runtime::Runtime::run_loop
 6: evcxr::runtime::runtime_hook
 7: evcxr_jupyter::main
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

- We can directly invoke `panic()` with an error message
- Useful if code detects an unrecoverable situation and wants to abort

In [39]:

```
fn main() {
    panic!("Crash and burn!");
}
```

In [40]: `main()`

```
thread '<unnamed>' panicked at src/lib.rs:3:5:
Crash and burn!
stack backtrace:
 0: rust_begin_unwind
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/std/src/panicking.rs:665:5
 1: core::panicking::panic_fmt
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/panicking.rs:76:14
 2: <unknown>
 3: <unknown>
 4: evcxr::runtime::Runtime::run_loop
 5: evcxr::runtime::runtime_hook
 6: evcxr_jupyter::main
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Result

- An enum to return informative error messages


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

- This code generates an error if `hello.txt` is not found in the current directory
- If the file can be opened, returns file handle for `hello.txt`

In [41]: `use std::fs::File;`

```
fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

In [42]: `main()`

```
thread '<unnamed>' panicked at src/lib.rs:7:23:
Problem opening the file: Os { code: 2, kind: NotFound, message: "No such
file or directory" }
stack backtrace:
 0: rust_begin_unwind
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/std/src/panicking.rs:665:5
 1: core::panicking::panic_fmt
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/panicking.rs:76:14
 2: <unknown>
 3: <unknown>
 4: <unknown>
 5: evcxr::runtime::Runtime::run_loop
 6: evcxr::runtime::runtime_hook
 7: evcxr_jupyter::main
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
```

- [Created `hello.txt`]
- Here is what happens if `hello.txt` opens fine

In [43]: `use std::fs::File;`

```
fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => {println!("Opened hello.txt"); file},
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

In [44]: `main()`

Opened hello.txt

Out[44]: ()

- [Deleted hello.txt]
- Can `match` on type of error
- Can have nested (chained) errors

```
In [45]: use std::fs::File;
use std::io::ErrorKind;

fn main() {
    let greeting_file_result = File::open("hello.txt");

    let greeting_file = match greeting_file_result {
        Ok(file) => file,
        Err(error) => match error.kind() {
            ErrorKind::NotFound => match File::create("hello.txt") {
                Ok(fc) => {println!("Created file!"); fc},
                Err(e) => panic!("Problem creating the file: {:?}", e),
            },
            other_error => {
                panic!("Problem opening the file: {:?}", other_error);
            }
        },
    };
}
```

In [46]: main()

Created file!

Out[46]: ()

- [Deleted hello.txt]
- Shortcut: `unwrap()` extracts the match for the `OK` branch

```
In [47]: use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt").unwrap();
}
```

In [48]: main()

```

thread '<unnamed>' panicked at src/lib.rs:66:49:
called `Result::unwrap()` on an `Err` value: Os { code: 2, kind: NotFound,
message: "No such file or directory" }
stack backtrace:
 0: rust_begin_unwind
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/std/src/panicking.rs:665:5
 1: core::panicking::panic_fmt
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/panicking.rs:76:14
 2: core::result::unwrap_failed
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/result.rs:1699:5
 3: <unknown>
 4: <unknown>
 5: <unknown>
 6: evcxr::runtime::Runtime::run_loop
 7: evcxr::runtime::runtime_hook
 8: evcxr_jupyter::main
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.

```

- `expect()` is triggered by the `Err` branch and panics with a message
- If no error, `unwrap()` is invoked implicitly, see later

```

In [49]: use std::fs::File;

fn main() {
    let greeting_file = File::open("hello.txt")
        .expect("hello.txt should be included in this project");
}

```

```

In [50]: main()

```

```

thread '<unnamed>' panicked at src/lib.rs:67:10:
hello.txt should be included in this project: Os { code: 2, kind: NotFound,
message: "No such file or directory" }
stack backtrace:
 0: rust_begin_unwind
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/std/src/panicking.rs:665:5
 1: core::panicking::panic_fmt
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/panicking.rs:76:14
 2: core::result::unwrap_failed
    at /rustc/9fc6b43126469e3858e2fe86cafb4f0fd5068869/library/core/src/result.rs:1699:5
 3: <unknown>
 4: <unknown>
 5: <unknown>
 6: evcxr::runtime::Runtime::run_loop
 7: evcxr::runtime::runtime_hook
 8: evcxr_jupyter::main
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.

```

Propagating errors

- The code below first tries to open `hello.txt`
- If this succeeds, it tries to read `username` from the file
- If either the file open or the file read fails, `Err(e)` is propagated to caller

```
In [51]: use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let username_file_result = File::open("hello.txt");

    let mut username_file = match username_file_result {
        Ok(file) => file,
        Err(e) => return Err(e),
    };

    let mut username = String::new();

    match username_file.read_to_string(&mut username) {
        Ok(_) => Ok(username),
        Err(e) => Err(e),
    }
}
```

- [Created `hello.txt` with contents `"Hello world"`]
- The example above, without an error
- `expect()` implicitly invokes `unwrap()` and file handle is stored in `greeting_file`

```
In [52]: use std::fs::File;
use std::io::{self, Read};

fn main() {
    let mut greeting_file = File::open("hello.txt")
        .expect("hello.txt should be included in this project");

    let mut username = String::new();

    greeting_file.read_to_string(&mut username);

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

```
In [53]: main()
```

Hello World

```
Out[53]: ()
```