Enums and Errors

Jonathan Dönszelmann & Vivian Roest

Delft University of Technology

2024-11-18

Last lecture:

- References
- Mutability
- Slices
- Enums?

Today:

- Enums!
- Patterns
- Errors
- Utilities for error handling

Enums

Enums

- Enums are a list of values that are possible to represent

```Rust
enum ArithmeticOperation {
  Add,
  Sub,
  Div,
  Mul,
}
```

Each possibility is called a “variant”

Enums refresher

- Counting the number of possible values
- `structs` are “Product Types”
- `enums` are “Sum Types”

Enums

- Counting the number of possible **constructors**, “ways of constructing a value”
- `structs` have *one* constructor (with parameters)
- `enums` have *multiple* constructors (with parameters)

Some examples:

- A boolean has **2** constructors (true and false)
- A `struct X(bool, bool)` has **1** top-level constructor and **4** total constructors
- `ArithmeticOperation` has **4** constructors as well

Rust

```rust
enum ArithmeticOperation {
    Add,
    Sub,
    Div,
    Mul,
}
```

Enums

Question:

What’s the minimum number of constructors a type needs?

Enums

Question:

How many possible values does a struct without fields have?

```rust
struct X {}
```

Enums

Question:

How many possible values does a `struct` without fields have?

```rust
struct X {}
```

Enums

Question:

How many possible values does an `enum` without variants have?

```rust
enum X {}
```

Enums

Rust

```rust
fn example() {
  // ...
}
```

Enums

Rust

```rust
// secretly returns `()`
fn example() -> () {
  // ...
}
```

Enums

```Rust
// secretly returns `()`
fn example() -> () {
  // ...
}
```

Parentheses make a tuple:

```Rust
let x: (i32, i32) = (4, 4);
```

Enums

```rust
// secretly returns `()`
fn example() -> () {
  // ...
}
```

So `()` is a tuple with no elements

```rust
let x: () = ();
```

Enums

```rust
// secretly returns `()`
fn example() -> () {
  // ...
}
```

So `()` is a tuple with no elements

Like a struct without fields

```rust
struct X {

};

let x: X = X {};
```

Enums

Rust

```rust
// secretly returns `()`
fn example() -> () {
  // ...
}
```

- The smallest unit for information is a bit, or a `bool`.
- It has two constructors
- A value with one constructor communicates no information (*)
- A function that returns **something** returns **some information**

A logical type for a function that returns **nothing** is a type that communicates **no information**

Enums

Rust

```rust
// secretly returns `()`
fn example() -> () {
  // ...
}
```

“A value with one constructor communicates no information”

Question:

If a function returns `()`, we do gain **one** piece of information. What’s that?

Enums

Rust

```rust
// secretly returns `()`
fn example() -> () {
  // ...
}
```

“A value with one constructor communicates no information”

Question:

If a function returns `()`, we do gain **one** piece of information. What’s that?

`()` has one constructor. It cummunicates no information, but it’s a value *we can make*.

Enums

Rust

```rust
enum NoConstructors {}

fn example() -> NoConstructors {
  // what can we do here to return a value of type `NoConstructors`??
}
```

- We sure can’t “construct” it….

Enums

```Rust
enum NoConstructors {}

fn example() -> NoConstructors {
  // but we can avoid having to construct it...
  loop {}
}
```

- By going in an infinite loop, we don't *have* to construct it

Enums

We call `NoConstructors` the “never type” and you can write it as `!`

Rust

```rust
fn example() -> ! {
  // never returns
  loop {}
}
```

- `panic!()` returns `!`
- `std::process::exit()` returns `!`
- `loop{}` returns `!`
- When there’s no operating system, `main` returns `!`

Patterns

Patterns

A literal is a piece of syntax that constructs a value.

- `1` is a literal
- `"hello"` is a literal
- `(1, 2, 3)` is a literal (with more literals inside)
- `[1, 2, 3]` is a literal
- `IpAddr::v4(127, 0, 0, 1)` is a literal

Patterns

A literal is a piece of syntax that constructs a value.

- `1` is a literal
- `"hello"` is a literal
- `(1, 2, 3)` is a literal (with more literals inside)
- `[1, 2, 3]` is a literal
- `IpAddr::v4(127, 0, 0, 1)` is a literal

You always find literals on the *right hand side* of an expression

Rust

```rust
let x =    3;
// literal ^
```

Patterns

Patterns are literals on the *left hand side* of an expression

Rust

```rust
let a = 3;
//  ^ pattern
```

- And they act like the inverse of literals
- Patterns can match some value
- The pattern `a` apparently matches the value `3`

Patterns

Patterns are literals on the *left hand side* of an expression

```rust
let a = 3;
//  ^ pattern
```

- And they act like the inverse of literals
- Patterns can match some value
- The pattern `a` apparently matches the value `3`

Patterns

Patterns are literals on the *left hand side* of an expression

Rust

```rust
let a = (1, 2);
//  ^ pattern
```

- `a` also matches `(1, 2)`
- In fact, a variable name matches any value
- And assigns that value to itself

Patterns

Patterns are literals on the *left hand side* of an expression

```rust
let (a, b) = (1, 2);
//  ^^^^^^ pattern
```

- `(a, b)` also matches `(1, 2)`
- Acting like the opposite of a literal, it destructs the value
- `1` matches against `a` (and assigns)
- `2` matches against `b` (and assigns)

Patterns

Patterns are literals on the *left hand side* of an expression

```rust
let _ = (1, 2);
//  ^ pattern deleting the value
```

_ discards a value

Patterns

Patterns are literals on the *left hand side* of an expression

Rust

```rust
//   v keeps a but discards the value 2
let (a, _) = (1, 2);
//  ^^^^^^ pattern
```

Patterns

Patterns are literals on the *left hand side* of an expression

```rust
struct Example {
  a: u32,
  b: u8,
  c: ()
}

//   v keeps a but discards the value 2
let (a, _) = (1, 2);
//  ^^^^^^ pattern
```

Patterns

Patterns are literals on the *left hand side* of an expression

```rust
struct Example {
  a: u32,
  b: u8,
  c: (u32, u32),
}

let x = Example {
  a: 3,
  b: 8,
  c: (1, 2),
};

let Example {a, c: (x, _), ..} = x;
println!("{a}, {x}");
```

Fallible patterns

Patterns are literals on the *left hand side* of an expression

```rust
let x = Example {
    a: 3,
    b: 8,
    c: (1, 2),
};

// Can we assign this? if so give me b (compile error)
let Example {a: 3, b, ..} = x;
println!("{b}");
```

Fallible patterns

Patterns are literals on the *left hand side* of an expression

Rust

```rust
let x = Example {
  a: 3,
  b: 8,
  c: (1, 2),
};

// Can we assign this? if so give me b
if let Example {a: 3, b, ..} = x {
  println!("{b}");
} else {
  println!("couldn't assign x");
}
```

`if let` means “try to assign and run some code if we could”

Fallible patterns

Match tries many patterns at the same time:

Rust

```rust
let x = (1, 2);

match x {
   (2, 3) | (3, 4) => { /*...*/ }
   (1, a) => println!("{a}"),
   (2, 5 | 6 | 7..9) => { /*...*/ }
   (a, 4) => { /*...*/ }
}
```

Question:

What happens with `x = (10, 20)`?

Fallible patterns

Match tries many patterns at the same time:

```rust
let x = (10, 20);

match x {
  (2, 3) | (3, 4) => { /*...*/ }
  (1, a) => {/* ... */},
  (2, 5 | 6 | 7..9) => { /*...*/ }
  (a, 4) => { /*...*/ }
  // this pattern matches anything
  other => { /*...*/ }
}
```

Fallible patterns

Match tries many patterns at the same time:

```rust
let x = (10, 20);

match x {
  (2, 3) | (3, 4) => { /*...*/ }
  (1, a) => {/* ... */},
  (2, 5 | 6 | 7..9) => { /*...*/ }
  (a, 4) => { /*...*/ }

  // this pattern discards everything
  _ => { }
}
```

Fallible patterns

Match tries many patterns at the same time:

Rust

```rust
let x = (10, 20);

match x {
  (2, 3) | (3, 4) => { /*...*/ }
  (1, a) => {/* ... */},
  (2, 5 | 6 | 7..9) => { /*...*/ }
  (a, 4) => { /*...*/ }

  // this pattern discards everything
  _ => { }
}
```

Enums and Patterns

- Enums have multiple constructors
- Patterns can discern between constructors of a type

Rust

```rust
enum WeekDay {
  Monday,
  Tuesday,
  // ...
  Friday,
}

let x = WeekDay::Friday;

match x {
  WeekDay::Saturday | WeekDay::Sunday => println!("weekend"),
  _ => println!("not weekend"),
```

Enums and Patterns

```
}
```

Enums and Patterns

- So this is how you can actually use options!

Rust

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

fn print_option(x: Option<u32>) {
  match x {
    Some(value) => println!("the value inside is {value}"),
    None => println!("there was no value"),
  }
}
```

Enums and Patterns

- So this is how you can actually use options!

Rust

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

fn print_option(x: Option<u32>) {
  if let Some(value) = option {
    println!("the value inside is {value}")
  } else {
    println!("there was no value")
  }
}
```

Assignment: 10 minutes

Weblab: Assignments - Week 2 - enums and matching - Creating Patterns

Put a single `match` block in each function

Assignment: 10 minutes

Build a list!

Template:

Rust

```rust
enum List {
  End,
  Item(u32, Box<List>)
}

impl List {
  fn create(&[u32]) -> List;
  fn length(&self) -> usize;
}
```

error handling

Another `enum` in the standard library:

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

It’s like `Option`, but “None can have a value too”

error handling

Another `enum` in the standard library:

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

You’ll often see functions like this:

```rust
fn do_thing(some_input: X) -> Result<Y, Error> {
    // if it goes well: Ok
    // if it goes bad: Err
}
```

error handling

Imagine reading from a file:

```Rust
fn read_from_file(filename: &Path) -> Result<String, Error> {
  // what could go wrong?
}
```

error handling

Imagine reading from a file:

```rust
enum Error {
  DoesntExist,
  PermissionDenied,
  MemoryFull,
  HarddriveGone,
  ComputerOnFire,
  // ... etc
}

fn read_from_file(filename: &Path) -> Result<String, Error> {
  // what could go wrong?
}
```

error handling

Imagine reading from a file:

Rust

```rust
enum Error {
  DoesntExist,
  ComputerOnFire,
  // ... etc
}
fn read_from_file(filename: &Path) -> Result<String, Error> {
  // note: in real life the OS checks this for us and we wouldn't check like this in advance
  if !check_if_file_exists(filename) {
    return Err(Error::DoesntExist)
  }
  // ... some more checks
  // and then finally:
  Ok(contents)
}
```

error handling

If something returns a result, the operation might go wrong.

We can check what went wrong:

```Rust
match read_from_file("meow.txt") {
    Ok(contents)             => println!("the file contains {contents}"),
    Err(Error::DoesntExist) => println!("the file doesn't exist"),
    Err(e)                   => println!("another error occurred: {e}"),
}
```

error handling

A common pattern:

Rust

```rust
fn do_big_thing() -> Result<Output, Error> {
  let outcome1 = match do_small_thing1() {
    Ok(i) => i,
    Err(e) => return Err(e),
  };
  let outcome2 = match do_small_thing2(outcome1) {
    Ok(i) => i,
    Err(e) => return Err(e),
  };

  Ok(outcome2)
}
```

error handling

A common pattern:

Rust

```rust
fn do_big_thing() -> Result<Output, Error> {
  let outcome1 = match do_small_thing1() { Ok(i) => i, Err(e) => return Err(e)};
  let outcome2 = match do_small_thing2(outcome1) { Ok(i) => i, Err(e) => return Err(e)};
  let outcome3 = match do_small_thing3(outcome2) { Ok(i) => i, Err(e) => return Err(e)};
  let outcome4 = match do_small_thing4(outcome3) { Ok(i) => i, Err(e) => return Err(e)};
  let outcome5 = match do_small_thing5(outcome4) { Ok(i) => i, Err(e) => return Err(e)};
  Ok(outcome5)
}
```

At each step:

- If we have an error, immediately stop
- Otherwise, do some more work

ugh so much writing…

error handling

A common pattern:

- Use `?` to do exactly the same:

Rust

```rust
fn do_big_thing() -> Result<Output, Error> {
  let outcome1 = do_small_thing1()?;
  // means
  let outcome1 = match do_small_thing1() {
    Ok(i) => i,
    Err(e) => return Err(e)
  };

  Ok(outcome5)
}
```

error handling

A common pattern:

So the big example from before becomes:

Rust

```rust
fn do_big_thing() -> Result<Output, Error> {
    let outcome1 = do_small_thing1()?;
    let outcome2 = do_small_thing2(outcome1)?;
    let outcome3 = do_small_thing3(outcome2)?;
    let outcome4 = do_small_thing4(outcome3)?;
    let outcome5 = do_small_thing5(outcome4)?;

    Ok(outcome5)
}
```

error handling

`?` means:

- if the the value is an error, return immediately
- otherwise, it’s `Ok`, and `?` returns the value inside `Ok`

Question:

Is this a correct program?

Rust

```rust
fn example() {
  do_small_thing()?;
}
```

error handling

Error handling in real life:

Rust

```rust
pub enum FromFileError {
      Io(io::Error),
      Json(serde_json::Error),
      Yaml(serde_yaml::Error),
      Plist(plist::Error),
}

fn from_file(path: &Path) -> Result<Something, FromFileError>;
```

- This is an error for a single function `from_file`
- A different function might have different ways to fail
- It can fail in 4 different major ways
- Each of the 4 error options contain more information

error handling in real life

```Rust
use thiserror::Error;

#[derive(Error, Debug)]
pub enum FromFileError {
    #[error(transparent)]
    Io(#[from] io::Error),
    #[error("json deserialization")]
    Json(#[from] serde_json::Error),
    #[error("yaml deserialization")]
    Yaml(#[from] serde_yaml::Error),
    #[error("xml deserialization")]
    Plist(#[from] plist::Error),
}
```

error handling in real life

Rust

```rust
#[derive(Error, Debug)]
pub enum FromFileError {
    Io(#[from] io::Error),
    // ...
}

fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, FromFileError> {
  let contents = read_file()?; // does this work?

  // ...
}
```

error handling in real life

Rust

```
pub enum FromFileError {
    Io(#[from] io::Error),
}

fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, FromFileError> {
  let contents = match read_file() {
    Ok(i) => i,
    //  v-- io::Error
    Err(e) => return Err(e),
    //    FromFileError--^
  }
}
```

error handling in real life

Rust

```rust
pub enum FromFileError {
    Io(#[from] io::Error),
}

fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, FromFileError> {
  let contents = match read_file() {
    Ok(i) => i,
    //  v-- io::Error      vvvv-- automatically convert
    Err(e) => return Err(e.into()),
    //    FromFileError--^^^^^^^^
  }
}
```

error handling in real life

Rust

```rust
#[derive(Error, Debug)]
pub enum FromFileError {
    Io(#[from] io::Error),
    // ...
}

fn read_file() -> Result<String, io::Error> {/* ... */}

fn from_file() -> Result<String, FromFileError> {
  // this works! automatic conversion
  let contents = read_file()?;
}
```

error handling in real life

Note: `?` also works for options!

```rust
fn find_foo() -> Option<Foo> {
  // maybe we find it, or not?
}

fn get_and_check_foo() -> Option<Foo> {
  // try to find it
  let foo = find_foo()?;

  if !foo.is_correct() {
    return None
  }

  Some(foo)
}
```

Utilities for `Option` and `Result`

Utilities for Option and Result

There are methods defined on options and results

Rust

```rust
impl<T> Option<T> {
  // all kinds of functions to work with options
}
```

Utilities for Option and Result

There are methods defined on options and results

Important one: map

Rust

```rust
impl<T> Option<T> {
    fn map(self, /*...*/);
}
```

Utilities for Option and Result

When we want to work with what is **inside** an option:

```rust
fn double(input: Option<u32>) -> Option<u32> {
  if let Some(inner) = input {
    Some(inner * 2)
  } else {
    None
  }
}
```

Utilities for Option and Result

When we want to work with what is **inside** an option:

Rust

```
fn double(input: Option<u32>) -> Option<u32> {
  Some(inner? * 2)
}
```

Utilities for Option and Result

Map: translate the contents of an option

```rust
fn double(input: Option<u32>) -> Option<u32> {
  input.map(|inner| inner * 2)
}
```

Utilities for Option and Result

How to read the documentation

https://doc.rust-lang.org/stable/std/option/enum.Option.html

Utilities for Option and Result

How to read the documentation

- `map` means to translate what’s inside
- `unwrap` means to get what’ inside
- `or` means to do something specific when there’s nothing inside
- `is` gets some property of the value

Utilities for Option and Result

How to read the documentation

https://doc.rust-lang.org/stable/std/result/enum.Result.html