Rust programming

Module D: Trait objects and Rust patterns

In this module

- Introducing dynamic dispatch
- Work with design pattern in Rust
- Common anti-patterns to avoid

Learning objectives

- Understand the difference between static and dynamic dispatch
- Be able to create trait objects
- Understand the concept of 'object safety'
- Apply various commonly used design patterns
- Know which anti-patterns to avoid

Module D

Idiomatic Rust Patterns

Content overview

- Trait objects and dynamic dispatch
- Rust design patterns
- The deref polymorphism anti-pattern
- Project intro

Trait objects & dynamic dispatch

Trait… Object?

- We learned about traits in module A3
- We learned about generics and `monomorphization`

There's more to this story though…

Question: What was monomorphization again?

Monomorphization: recap

```
impl MyAdd for i32 {/* - snip - */}
impl MyAdd for f32 {/* - snip - */}

fn add_values<T: MyAdd>(left: &T, right: &T) -> T
{
  left.my_add(right)
}

fn main() {
  let sum_one = add_values(&6, &8);
  assert_eq!(sum_one, 14);
  let sum_two = add_values(&6.5, &7.5);
  println!("Sum two: {}", sum_two); // 14
}
```

Code is *monomorphized*:

- Two versions of `add_values` end up in binary
- Optimized separately and very fast to run (static dispatch)
- Slow to compile and larger binary

Dynamic dispatch

What if don't know the concrete type implementing the trait at compile time?

```
use std::io::Write;
use std::path::PathBuf;

struct FileLogger { log_path: PathBuf }
impl Write for FileLogger { /* - snip -*/}

struct StdOutLogger;
impl Write for StdOutLogger { /* - snip -*/}

fn log<L: Write>(entry: &str, logger: &mut L) {
    write!(logger, "{}", entry);
}

fn main() {
    let log_file: Option<PathBuf> =
        todo!("read args");
    let mut logger = match log_file {
        Some(log_path) => FileLogger { log_path },
        Nome => StdOutLogger,
    };

    log("Hello, world!🦀", &mut logger);
}
```

Error!

```
error[E0308]: `match` arms have incompatible types
  --> src/main.rs:19:17
   |
17 |       let mut logger = match log_file {
   |  ______________________-
18 | |         Some(log_path) => FileLogger { log_path },
   | |                           ----------------------- this is found to be of type `FileLogger`
19 | |         Nome => StdOutLogger,
   | |                 ^^^^^^^^^^^^ expected struct `FileLogger`, found struct `StdOutLogger`
20 | |     };
   | |_____- `match` arms have incompatible types
```

What's the type of `logger`?

Heterogeneous collections

What if we want to create collections of different types implementing the same trait?

```
trait Render {
    fn paint(&self);
}

struct Circle;
impl Render for Circle {
    fn paint(&self) { /* - snip - */ }
}

struct Rectangle;
impl Render for Rectangle {
    fn paint(&self) { /* - snip - */ }
}

fn main() {
    let mut shapes = Vec::new();
    let circle = Circle;
    shapes.push(circle);
    let rect = Rectangle;
    shapes.push(rect);
    shapes.iter().for_each(|shape| shape.paint());
}
```

Error again!

```
   Compiling playground v0.0.1 (/playground)
error[E0308]: mismatched types
  --> src/main.rs:20:17
   |
20 |     shapes.push(rect);
   |            ---- ^^^^ expected struct `Circle`, found struct `Rectangle`
   |            |
   |            arguments to this method are incorrect
   |
note: associated function defined here
  --> /rustc/2c8cc343237b8f7d5a3c3703e3a87f2eb2c54a74/library/alloc/src/vec/mod.rs:1836:12

For more information about this error, try `rustc --explain E0308`.
error: could not compile `playground` due to previous error
```

What is the type of `` `shapes` ``?

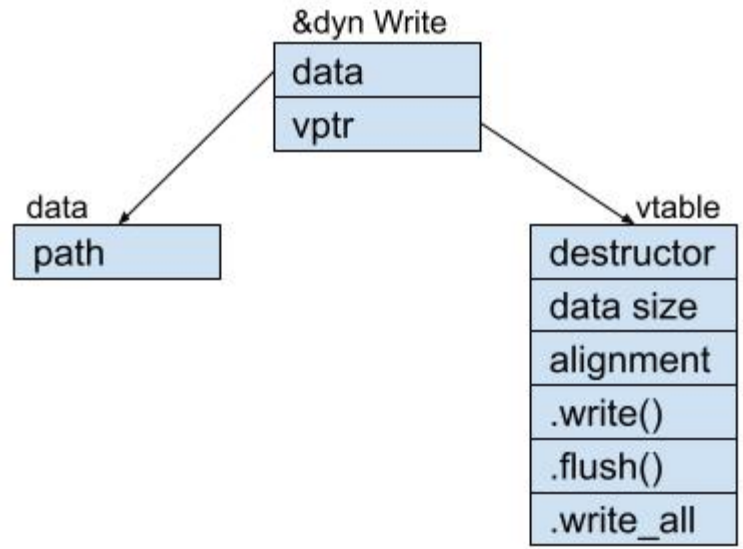
Trait objects to the rescue

- Opaque type that implements a set of traits
- Type description: `dyn T: !Sized` where `T` is a `trait`
- Like slices, Trait Objects always live behind pointers (`&dyn T`, `&mut dyn T`, `Box<dyn T>`, `...`)
- Concrete underlying types are erased from trait object

```
fn main() {
    let log_file: Option<PathBuf> =
        todo!("read args");
    // Create a trait object that implements `Write`
    let logger: &mut dyn Write = match log_file {
        Some(log_path) => &mut FileLogger { log_path },
        Nome => &mut StdOutLogger,
    };
}
```

Layout of trait objects

```
1   /// Same code as last slide
2   fn main() {
3       let log_file: Option<PathBuf> =
4           todo!("read args");
5       // Create a trait object that implements `Write`
6       let logger: &mut dyn Write = match log_file {
7           Some(log_path) => &mut FileLogger { log_path
},
8           Nome => &mut StdOutLogger,
9       };
10
11      log("Hello, world!", &mut logger);
12  }
```



- *Cost: pointer indirection via vtable → less performant*
- *Benefit: no monomorphization → smaller binary & shorter compile time!*

Fixing dynamic logger

- Trait objects `&dyn T`, `Box<dyn T>`, ... implement `T`!

```
// We no longer require L be `Sized`, so to accept trait objects
fn log<L: Write + ?Sized>(entry: &str, logger: &mut L) {
    write!(logger, "{}", entry);
}

fn main() {
    let log_file: Option<PathBuf> =
        todo!("read args");
    // Create a trait object that implements `Write`
    let logger: &mut dyn Write = match log_file {
        Some(log_path) => &mut FileLogger { log_path },
        Nome => &mut StdOutLogger,
    };

    log("Hello, world!🦀", logger);
}
```

And all is well!

Forcing dynamic dispatch

Sometimes you want to enforce API users (or colleagues) to use dynamic dispatch

```
fn log(entry: &str, logger: &mut dyn Write) {
    write!(logger, "{}", entry);
}

fn main() {
    let log_file: Option<PathBuf> =
        todo!("read args");
    // Create a trait object that implements `Write`
    let logger: &mut dyn Write = match log_file {
        Some(log_path) => &mut FileLogger { log_path },
        Nome => &mut StdOutLogger,
    };


    log("Hello, world!🦀", &mut logger);
}
```

Fixing the renderer

```
fn main() {
    let mut shapes = Vec::new();
    let circle = Circle;
    shapes.push(circle);
    let rect = Rectangle;
    shapes.push(rect);
    shapes.iter().for_each(|shape| shape.paint());
}
```

Becomes

```
fn main() {
    let mut shapes: Vec<Box<dyn Render>> = Vec::new();
    let circle = Box::new(Circle);
    shapes.push(circle);
    let rect = Box::new(Rectangle);
    shapes.push(rect);
    shapes.iter().for_each(|shape| shape.paint());
}
```

All set!

Trait object limitations

- Pointer indirection cost
- Harder to debug
- Type erasure
- Not all traits work:

Traits need to be 'Object Safe'

Object safety

In order for a trait to be object safe, these conditions need to be met:

- If `trait T: Y`, then `Y` must be object safe
- trait `T` must not be `Sized`: *Why?*
- No associated constants allowed*
- No associated types with generic allowed*
- All associated functions must either be dispatchable from a trait object, or explicitly non-dispatchable
  - e.g. function must have a receiver with a reference to `Self`

Details in The Rust Reference. Read them!

*These seem to be compiler limitations

So far...

- Trait objects allow for dynamic dispatch and heterogeneous
- Trait objects introduce pointer indirection
- Traits need to be object safe to make trait objects out of them

Design patterns in Rust

Why learn design patterns?

- Common problems call for common, tried an tested solutions
- Make crate architecture more clear
- Speed up development
- Rust does some patterns ever-so-slightly differently

Learning common Rust patterns makes understanding new code easier

What we'll do

```
const PATTERNS: &[Pattern] = &[
    Pattern::new("Newtype"),
    Pattern::new("RAII with guards"),
    Pattern::new("Typestate"),
    Pattern::new("Strategy"),
];
fn main() {
    for pattern in PATTERNS {
        pattern.introduce();
        pattern.show_example();
        pattern.when_to_use();
    }
}
```

1. The Newtype pattern

a small but useful pattern

Newtype: introduction

Wrap an external type in a new local type

```
1   pub struct Imei(String)
```

That's it!

Newtype: example

```rust
pub enum ValidateImeiError { /* - snip - */}

pub struct Imei(String);

impl Imei {
    fn validate(imei: &str) -> Result<(), ValidateImeiError> {
        todo!();
    }
}

impl TryFrom<String> for Imei {
    type Error = ValidateImeiError;

    fn try_from(imei: String) -> Result<Self, Self::Error> {
        Self::validate(&imei)?;
        Ok(Self(imei))
    }
}

fn register_phone(imei: Imei, label: String) {
    // We can certain `imei` is valid here
}
```

Newtype: when to use

Newtype solves some problems:

- Orphan rule: no `impl`s for external `trait`s on external types
- Allow for semantic typing (`url` example from mod B)
- Enforce input validation

2. The RAII guard pattern

More robust resource handling

RAII Guards: introduction

- Resource Acquisition Is Initialization (?)
- Link acquiring/releasing a resource to the lifetime of a variable
- Guard constructor initializes resource, destructor frees it
- Access resource through the guard

Do you know of an example?

RAII Guards: example

```rust
pub struct Transaction<'c> {
    connection: &'c mut Connection,
    did_commit: bool,
    id: usize,
}

impl<'c> Transaction<'c> {
    pub fn begin(connection: &'c mut Connection)
     -> Self {
        let id =
            connection.start_transaction();
        Self {
            did_commit: false,
            id,
            connection,
        }
    }

    pub fn query(&self sql: &str) { /* - snip - */}

    pub fn commit(self) {
        self.did_commit = true;
    }
}
```

```rust
impl Drop for Transaction<'_> {
    fn drop(&mut self) {
        if self.did_commit {
            self
                .connection
                .commit_transaction(self.id);

        } else {
            self
                .connection
                .rollback_transaction(self.id);
        }
    }
}
```

RAII Guards: when to use

- Ensure a resource is freed at some point
- Ensure invariants hold while guard lives

3. The Typestate pattern

Encode state in the type

Typestate: introduction

- Define uninitializable types for each state of your object

```
pub enum Ready {} // No variants, cannot be initialized
```

- Make your type generic over its state using `std::marker::PhantomData`
- Implement methods only for relevant states
- Methods that update state take owned `self` and return instance with new state

`PhantomData<T>` makes types act like they own a `T`, and takes no space

Typestate: example

```rust
pub enum Idle {} // Nothing to do
pub enum ItemSelected {} // Item was selected
pub enum MoneyInserted {} // Money was inserted

pub struct CoffeeMachine<S> {
    _state: PhantomData<S>,
}

impl<CS> CoffeeMachine<CS> {
    /// Just update the state
    fn into_state<NS>(self) -> CoffeeMachine<NS> {
        CoffeeMachine {
            _state: PhantomData,
        }
    }
}

impl CoffeeMachine<Idle> {
    pub fn new() -> Self {
        Self {
            _state: PhantomData,
        }
    }
}
```

```rust
impl CoffeeMachine<Idle> {
    fn select_item(self, item: usize) -> CoffeeMachine<ItemSelected> {
        println!("Selected item {item}");
        self.into_state()
    }
}

impl CoffeeMachine<ItemSelected> {
    fn insert_money(self) -> CoffeeMachine<MoneyInserted> {
        println!("Money inserted!");
        self.insert_money()
    }
}

impl CoffeeMachine<MoneyInserted> {
    fn make_beverage(self) -> CoffeeMachine<Idle> {
        println!("There you go!");
        self.into_state()
    }
}
```

Typestate: when to use

- If your problem is like a state machine
- Ensure *at compile time* that no invalid operation is done

4. The Strategy pattern

Select behavior dynamically

Strategy: introduction

- Turn set of behaviors into objects
- Make them interchangeble inside context
- Execute strategy depending on input

Trait objects work well here!

Strategy: example

```

trait PaymentStrategy {
    fn pay(&self);
}

struct CashPayment;
impl PaymentStrategy for CashPayment {
    fn pay(&self) {
        println!("□□");
    }
}

struct CardPayment;
impl PaymentStrategy for CardPayment {
    fn pay(&self) {
        println!("□");
    }
}
```

```

fn main() {
    let method: &str
        = todo!("Read input");
    let strategy: &dyn PaymentStrategy
        = match method {
        "card" => &CardPayment,
        "cash" => &CashPayment,
        _ => panic!("Oh no!"),
    };
    strategy.pay();
}
```

Strategy: when to use

- Switch algorithms based on some run-time parameter (input, config, ...)

Anti-patterns

What *not* to do

The Deref polymorphism anti-pattern

A common pitfall you'll want to avoid

Deref polymorphism: Example

```rust
use std::ops::Deref;

struct Animal {
    name: String,
}

impl Animal {
    fn walk(&self) {
        println!("Tippy tap")
    }
    fn eat(&self) {
        println!("Om nom")
    }
    fn say_name(&self) {
        // Animals generally can't speak
        println!("...")
    }
}
```

```rust
struct Dog {
    animal: Animal
}
impl Dog {
    fn eat(&self) {
        println!("Munch munch");
    }
    fn bark(&self) {
        println!("Woof woof!");
    }
}

impl Deref for Dog {
    type Target = Animal;

    fn deref(&self) -> &Self::Target {
        &self.animal
    }
}

fn main (){
    let dog: Dog = todo!("Instantiate Dog");
    dog.bark();
    dog.walk();
    dog.eat();
```

The output

```
Woof woof!
Tippy tap
Munch munch
...
```

Even overloading works!

Why is it bad?

- This is no 'real' inheritance: `Dog` is no subtype of `Animal`
- Traits implemented on `Animal` are not implemented on `Dog` automatically
- `Deref` and `DerefMut` are intended 'pointer-to-`T`' to `T` conversions
- Deref coercion by `.` 'converts' `self` from `Dog` to `Animal`
- Rust favours explicit conversions for easier reasoning about code

It will only add confusion: for OOP programmers it's incomplete, for Rust programmers it is unidiomatic

Don't do OOP in Rust!

What to do instead?

- *Move away from OOP constructs*
- Compose your structs
- Use facade methods
- Use `AsRef` and `AsMut` for explicit conversion

More anti-patterns

- Forcing dynamic dispatch in libraries
- `clone()` *to satisfy the borrow checker*
- `unwrap()` or `expect()` *to handle conditions that are recoverable or not impossible*

Summary

- Trait objects allow for heterogeneous collections and dynamic dispatch
- Use design patterns to address common problems
- Don’t do OOP in Rust!

Further reading on design patterns: Rust Design Patterns

Tutorial

- Project introduction
- Work on module D exercises
- Work on project proposal