Rust programming

Module 2: Foundations of Rust

Unit 5

Closures and Dynamic dispatch

Learning objectives

Closures

Closures

- Closures are anonymous (unnamed) functions
- they can capture ("close over") values in their scope
- they are first-class values

```
fn foo() -> impl Fn(i64, i64) -> i64 {
    z = 42;
    |x, y| x + y + z
}

fn bar() -> i64 {
    // construct the closure
    let f = foo();

    // evaluate the closure
    f(1, 2)
}
```

- very useful when working with iterators, `Option` and `Result`.

```
let evens: Vec<_> = some_iterator.filter(|x| x % 2 == 0).collect();
```

To do

Issue: tweedegolf/teach-rs#66

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

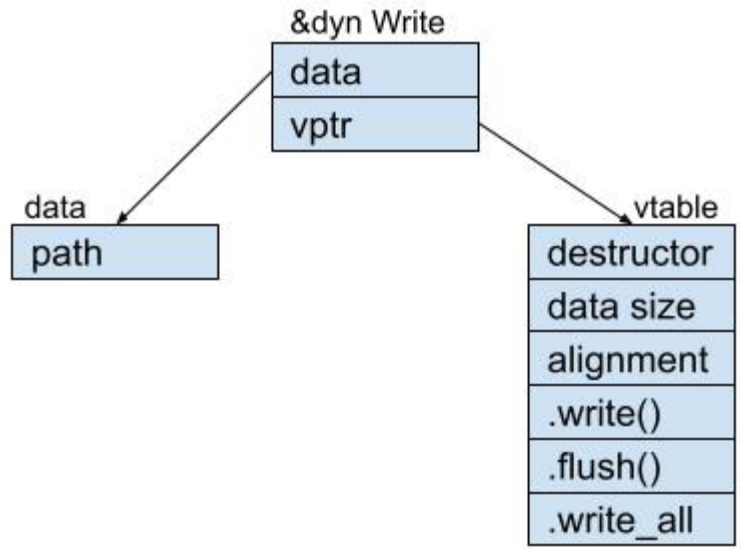
- Opaque type that implements a set of traits
- Type description: `dyn T: !Sized` where `T` is a `trait`
- Like slices, Trait Objects always live behind pointers (`&dyn T` , `&mut dyn T` , `Box<dyn T>` , `...`)
- Concrete underlying types are erased from trait object

```
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

```
/// Same code as last slide
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);
}
```



- 💸 *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 },
        None => &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

Summary