

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

```
1  fn foo() -> impl Fn(i64, i64) -> i64 {
2      let z = 42;
3      move |x, y| x + y + z
4  }
5
6  fn bar() -> i64 {
7      // construct the closure
8      let f = foo();
9
10     // evaluate the closure
11     f(1, 2)
12 }
```

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

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

layout: section

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

```
1  impl MyAdd for i32 { /* - snip - */ }
2  impl MyAdd for f32 { /* - snip - */ }
3
4  fn add_values<T: MyAdd>(left: &T, right: &T) -> T
5  {
6      left.my_add(right)
7  }
8
9  fn main() {
10     let sum_one = add_values(&6, &8);
11     assert_eq!(sum_one, 14);
12     let sum_two = add_values(&6.5, &7.5);
13     println!("Sum two: {}", sum_two); // 14
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?

```
1  use std::io::Write;
2  use std::path::PathBuf;
3
4  struct FileLogger { log_path: PathBuf }
5  impl Write for FileLogger { /* - snip -*/}
6
7  struct StdOutLogger;
8  impl Write for StdOutLogger { /* - snip -*/}
9
10 fn log<L: Write>(entry: &str, logger: &mut L) {
11     write!(logger, "{}", entry);
12 }
13
14 fn main() {
15     let log_file: Option<PathBuf> =
16         todo!("read args");
17     let mut logger = match log_file {
18         Some(log_path) => FileLogger { log_path },
19         None => StdOutLogger,
20     };
21
22     log("Hello, world! 🐙", &mut logger);
23 }
```

Error!

```
1  error[E0308]: `match` arms have incompatible types
2    --> src/main.rs:19:17
3    |
4  17 |         let mut logger = match log_file {
5    |             _____-
6  18 | |             Some(log_path) => FileLogger { log_path },
7    | |                                     ----- this is found to be of type `FileLogger`
8  19 | |             None => StdOutLogger,
9    | |                     ^^^^^^^^^^^^^^^^^ expected struct `FileLogger`, found struct `StdOutLogger`
10  20 | |         };
11    | | _____- `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?

```
1  trait Render {  
2      fn paint(&self);  
3  }  
4  
5  struct Circle;  
6  impl Render for Circle {  
7      fn paint(&self) { /* - snip - */ }  
8  }  
9  
10 struct Rectangle;  
11 impl Render for Rectangle {  
12     fn paint(&self) { /* - snip - */ }  
13 }  
14  
15 fn main() {  
16     let mut shapes = Vec::new();  
17     let circle = Circle;  
18     shapes.push(circle);  
19     let rect = Rectangle;  
20     shapes.push(rect);  
21     shapes.iter().for_each(|shape| shape.paint());  
22 }
```

Error again!

```
1    Compiling playground v0.0.1 (/playground)
2    error[E0308]: mismatched types
3      --> src/main.rs:20:17
4      |
5  20 |     shapes.push(rect);
6      |           ---- ^^^^^ expected struct `Circle`, found struct `Rectangle`
7      |           |
8      |           arguments to this method are incorrect
9      |
10   note: associated function defined here
11     --> /rustc/2c8cc343237b8f7d5a3c3703e3a87f2eb2c54a74/library/alloc/src/vec/mod.rs:1836:12
12
13   For more information about this error, try `rustc --explain E0308`.
14   error: could not compile `playground` due to previous error
```

What is the type of `shapes` ?

Trait objects to the rescue

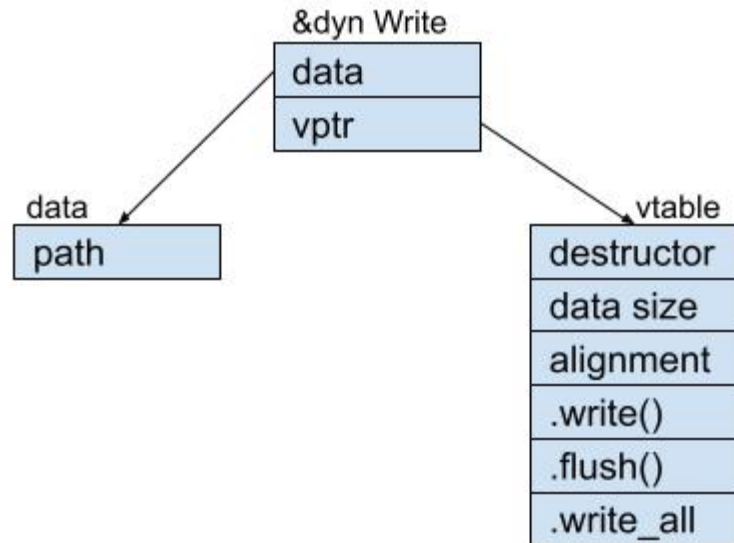
- Opaque type that implements a set of traits
- Type description: ``dyn T: !Sized`` where ``T`` is a ``trait``
- Like slices, Trait Objects always live behind pointers (``&dyn T``, ``&mut dyn T``, ``Box<dyn T>``, ``...``)
- Concrete underlying types are erased from trait object

```
1 fn main() {  
2     let log_file: Option<PathBuf> =  
3         todo!("read args");  
4     // Create a trait object that implements `Write`  
5     let logger: &mut dyn Write = match log_file {  
6         Some(log_path) => &mut FileLogger { log_path },  
7         None => &mut StdOutLogger,  
8     };  
9 }
```

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          None => &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`` !

```
1  // We no longer require L be `Sized`, so to accept trait objects
2  fn log<L: Write + ?Sized>(entry: &str, logger: &mut L) {
3      write!(logger, "{}", entry);
4  }
5
6  fn main() {
7      let log_file: Option<PathBuf> =
8          todo!("read args");
9      // Create a trait object that implements `Write`
10     let logger: &mut dyn Write = match log_file {
11         Some(log_path) => &mut FileLogger { log_path },
12         None => &mut StdOutLogger,
13     };
14
15     log("Hello, world! 🐞", logger);
16 }
```

And all is well!

Forcing dynamic dispatch

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

```
1  fn log(entry: &str, logger: &mut dyn Write) {
2      write!(logger, "{}", entry);
3  }
4
5  fn main() {
6      let log_file: Option<PathBuf> =
7          todo!("read args");
8      // Create a trait object that implements `Write`
9      let logger: &mut dyn Write = match log_file {
10         Some(log_path) => &mut FileLogger { log_path },
11         None => &mut StdOutLogger,
12     };
13
14
15     log("Hello, world! 🐙", &mut logger);
16 }
```


Fixing the renderer

```
1  fn main() {  
2      let mut shapes = Vec::new();  
3      let circle = Circle;  
4      shapes.push(circle);  
5      let rect = Rectangle;  
6      shapes.push(rect);  
7      shapes.iter().for_each(|shape| shape.paint());  
8  }
```

Becomes

```
1  fn main() {  
2      let mut shapes: Vec<Box<dyn Render>> = Vec::new();  
3      let circle = Box::new(Circle);  
4      shapes.push(circle);  
5      let rect = Box::new(Rectangle);  
6      shapes.push(rect);  
7      shapes.iter().for_each(|shape| shape.paint());  
8  }
```

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