Lecture 15

Static Vs. Dynamic Dispatch

The Return of Polymorphism

Static and Dynamic Dispatch

- Rust has a very strong preference for **static dispatch** of function calls, which is where the function matching a call is determined at compile-time.
- **Dynamic dispatch**: function matching a call is determined at run-time.
- Static dispatch leads to faster performance while dynamic dispatch gives flexibility.

Dynamic Dispatch in Rust

- It can be easy to think Rust's traits alone imply dynamic dispatch, but that is often not the case.

```
trait T {
  fn m(&self) -> u64;
}

struct S {
  i: u64
}

impl T for S {
  fn m(&self) -> u64 { self.i }
}

fn main() {
  let s = S{i : 100};
  println!("{}", s.m());
}
```

Dynamic Dispatch in Rust

- Rust compiler statically resolves the call m() to the function T::m.
- However, Rust does allow dynamic dispatch.
- Let's assume we want to make a function which can take in any struct which implements the trait T and calls function m. We might try and write this:

```
fn f(x: T) {
  println!("{}", x.m())
}
```

Dynamic Dispatch in Rust

- However compiling that leads to the following error:

```
error[E0277]: the size for values of type `(dyn T + 'static)`
cannot be known at compilation time
  --> src/main.rs:21:6
   |
21 | fn f(x: T) {
   |      ^ doesn't have a size known at compile-time
   |
   = help: the trait `std::marker::Sized` is not implemented
for `(dyn T + 'static)`
```

dyn -- trait object

- We don't know how big any given struct that implements T will be, so we can’t generate a single chunk of machine code that handles it.
- Unknown size always implies – must be on heap!

```
fn f(x: Box<dyn T>) {
  println!("{}", x.m())
}
```

- Pointer from stack, to function on heap. Pick function at run-time

Trait objects – Polymorphism?

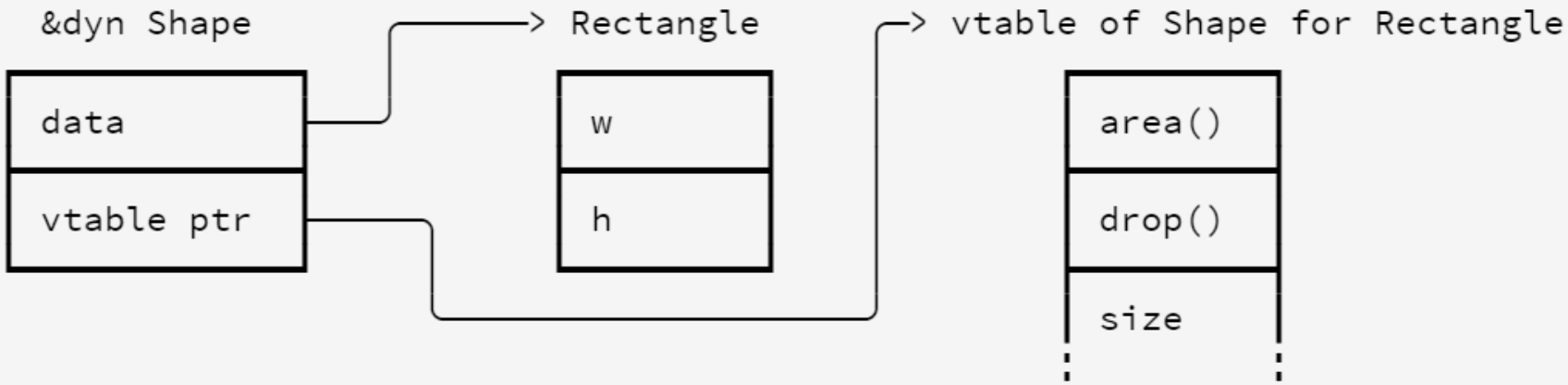
```
trait Shape
{
   fn area(&self) -> f32;
}
struct Rectangle { w: f32, h : f32 }

impl Shape for Rectangle {
   fn area(&self) -> f32 { self.w * self.h }
}
struct Circle { r: f32 }
impl Shape for Circle {
   fn area(&self) -> f32 { 3.14 * self.r * self.r }
}

fn total_area(list: &[&dyn Shape]) -> f32 {
    list.iter().map(|x| x.area()).fold(0., |a, b| a+b)
}
```

Virtual Pointers and Tables

- Here is a simplified representation of the memory layout



Polymorphism – really?

```
struct Service<T:Backend>{

 backend: Vec<T>  // Either Vec<TypeA> or Vec<TypeB>, not both

}
...

let mut backends = Vec::new();

backends.push(TypeA);

backends.push(TypeB);  // <---- Type error here
```

Fixing the problem

```
struct Service{

    backends: Vec<Box<dyn Backend>>

}

...

let mut backends = Vec::new();

backends.push( Box::new(TypeA{}) as Box<dyn Backend>);

backends.push( Box::new(TypeB{}) as Box<dyn Backend>);
```