Lecture 15

Static Vs. Dynamic Dispatch

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 and better safety checking while dynamic dispatch gives flexibility.
- Given, an option Rust says choose STATIC always!

Static Dispatch and Monomorphism

```
trait Foo {
    fn method(&self) -> String;
}
```

We can use this trait to perform static dispatch with trait bounds:

```
fn do_something<T: Foo>(x: T) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something(x);
    do_something(y);
}
```

Rust uses ‘monomorphization’ to perform static dispatch here

- This means that Rust will create a special version of `do_something()` for both `u8` and `String`, and then replace the call sites with calls to these specialized functions. In other words, Rust generates something like this:

```rust
fn do_something_u8(x: u8) {
    x.method();
}

fn do_something_string(x: String) {
    x.method();
}

fn main() {
    let x = 5u8;
    let y = "Hello".to_string();

    do_something_u8(x);
    do_something_string(y);
}
```

Trait Objects

```
fn main(){
    trait Animal {
        fn eat(&self);
    }

    struct Herbivore;
    struct Carnivore;

    impl Animal for Herbivore {
        fn eat(&self) {
            println!("I eat plants");
        }
    }
    impl Animal for Carnivore {
        fn eat(&self) {
            println!("I eat flesh");
        }
    }

    let h = Herbivore;
    h.eat();
    let c = Carnivore;
    c.eat();
}
```

Okay, h and c are trait objects.

But can a put h and c into a single container?

Dynamic Dispatch – and more animals

- The compiler needs to know how much space every function's return type requires. This means all your functions must return a concrete type.
- However, there's a workaround. Instead of returning a trait object directly, our functions return a Box which contains some type.
- Rust tries to be as explicit as possible whenever it allocates memory on the heap. So if your function returns a pointer-to-trait-on-heap in this way, you need to write the return type with the dyn keyword

For example

```
struct Sheep {}

struct Cow {}

trait Animal {  noise(&self) -> &'static str;}

impl Animal for Sheep {    fn noise(&self) -> &'static str {
"baaaaah!"    }}

impl Animal for Cow {    fn noise(&self) -> &'static str {
"moooooo!"    }}
```

Returning An (unknown) Animal

```
fn random_animal(random_number: f64) -> Box<dyn Animal>
{    if random_number < 0.5
          {       Box::new(Sheep {})    }
          else {       Box::new(Cow {})    }
}



fn main() {
let random_number = 0.234;
let animal = random_animal(random_number);
}
```

Applying to the h and c code

```
fn main() {
..........
   // Create a vector of Animals:
   let mut list: Vec<Box<dyn Animal>> = Vec::new();
   let goat = Herbivore;
   let dog = Carnivore;

   list.push(Box::new(goat));
   list.push(Box::new(dog));

   // Calling eat() for all animals in the list:
   for animal in &list{
      animal.eat();
   }
}
```

Trait objects – Polymorphism?

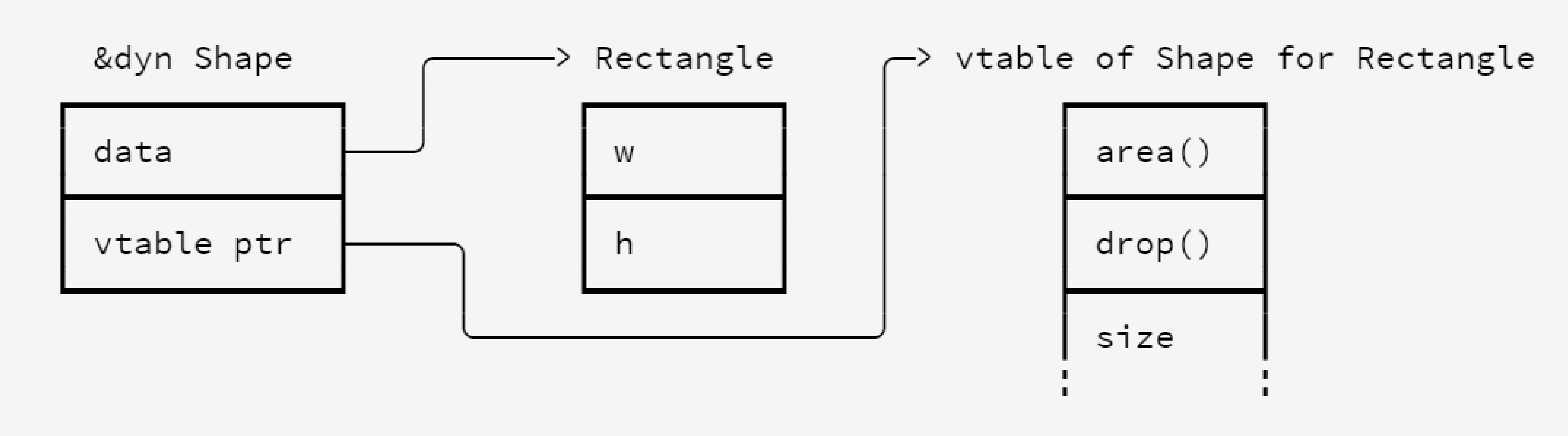
```
trait Shape
{
   fn area(&self) -> f32;
}
struct Rectangle { w: f32, h : f32 }

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



Everything on the heap, compiler sidelined

Do I really need a shape?

Or just an enum – rectangle or circle

Rust likes the enum option if viable