EventHelix > Rust > Compare the Assembly Generated for Static vs Dynamic Dispatch in Rust

Compare the Assembly Generated for Static vs Dynamic Dispatch in Rust

Traits in Rust are similar to interfaces in other languages. Traits permit the implementation of a common interface for multiple types. Developers can then write code in terms of the traits rather than the concrete types.

When a trait specified method is called, the compiler will generate code to dispatch the call to the concrete method implementing the trait specification. Rust supports two types of dispatch:

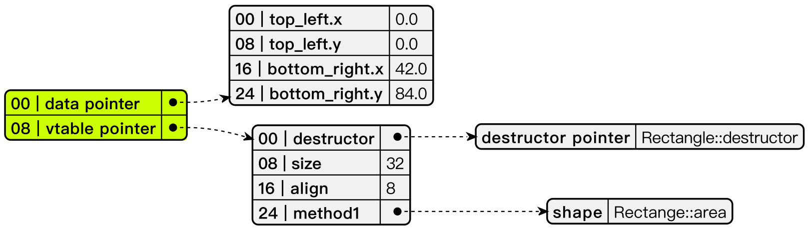
- **Static dispatch** is the default dispatch mode that is used when the concrete type can be determined at compile time.
- **Dynamic dispatch** is used when the concrete type implementing the trait is not known at compile time.

We will use the following example to illustrate the difference between static and dynamic dispatch.

```rust
// The Shape trait requires that the implementor have a method called `area`
// that returns the area as the associated type `Shape::T`.
pub trait Shape {
    type T;
    fn area(&self) -> Self::T;
}

// A generic Point for a specified type T.
pub struct Point<T> {
    x: T,
    y: T,
}

// A generic Rectangle for a specified type T.
pub struct Rectangle<T> {
    top_left: Point<T>,
    bottom_right: Point<T>,
}

// Implement the Shape trait for the Rectangle using a generic type T.
```

```
// The `T` is the return type of the `area` method. The where clause specifies
// that `T` must support subtraction, multiplication and ability to copy.
impl<T> Shape for Rectangle<T>
where
    T: std::ops::Sub<Output = T> + std::ops::Mul<Output = T> + Copy,
{
    type T = T;
    fn area(&self) -> T {
        let width = self.bottom_right.x - self.top_left.x;
        let height = self.top_left.y - self.bottom_right.y;

        width * height
    }
}

// A function that calculates the area of two shapes and returns a tuple containing the area
// This function requires that the two shapes implement the Shape trait. The function will c
// the area function via a static dispatch. Code will be generated for the function only if
// types are specified for `a` and `b`.
pub fn area_pair_static(a: impl Shape<T = f64>, b: impl Shape<T = f64>) -> (f64, f64) {
    (a.area(), b.area())
}

pub fn static_dispatch_pair(a: Rectangle<f64>, b: Rectangle<f64>) -> (f64, f64) {
    // The following line will generate code for the function as concrete types are specifie
    area_pair_static(a, b)
}

// This function performs the same function as `area_pair_static` but uses a dynamic dispatc
// will generate code for this function. The calls to `area` are made through the `vtable`.
pub fn area_pair_dynamic(a: &dyn Shape<T = f64>, b: &dyn Shape<T = f64>) -> (f64, f64) {
    (a.area(), b.area())
}
```

Type of dispatch	Parameters a and b (highlighted in green)
Static dispatch: The `static_dispatch_pair` expects `a` and `b` parameters as values with concrete types (`impl Shape<T = f64>` is syntactic sugar for generic parameters that implement the stated trait.). Thus the called function knows the layout of `a` and `b` at compile time. This also means any calls to the `area` method in the `Shape` trait can be made directly. These calls, as we will see, can also be inlined.	00 | top_left.x — 0.0 08 | top_left.y — 0.0 16 | bottom_right.x — 42.0 24 | bottom_right.y — 84.0

Type of dispatch	Parameters a and b (highlighted in green)
Dynamic dispatch: The `area_pair_dynamic` function expects `&dyn Shape<T = f64>` references to `a` and `b` . The function has no access to the concrete type implementing the trait. Even the calls to the `area` method have to be made via a function pointer. The function pointer is obtained via a `vtable` that points to a memory block that contains pointers to the methods implemented for the trait.	

Static dispatch

```rust
pub fn area_pair_static(a: impl Shape<T = f64>, b: impl Shape<T = f64>) -> (f64, f64) {
    (a.area(), b.area())
}

pub fn static_dispatch_pair(a: Rectangle<f64>, b: Rectangle<f64>) -> (f64, f64) {
    area_pair_static(a, b)
}
```

The static dispatch code takes place in the call to `area_pair_static` function. The function requires that `a` and `b` should belong to a type that implements the `Shape<T = f64>` trait. As mentioned earlier, the syntax used for `impl` based parameter passing is syntactic sugar to a generic based representation.

```rust
pub fn area_pair_static<S, T> (a: S, b: T) -> (f64, f64)
where
    S: Shape<T = f64>,
    T: Shape<T = f64>,
{
    (a.area(), b.area())
}
```

With the generic representation, `area_pair_static` function is generic over two types `S` and `T` . The function requires that `S` and `T` should implement the `Shape<T = f64>` trait. The function body calls the `area` method on the `a` and `b` parameters. The compiler will generate code for the function only if concrete types are specified for `a` and `b` . The function `static_dispatch_pair` does exactly that. `S` and `T` are defined as `Rectange<f64>` .

`Rectangle<f64>` layout is shown below. The `top_left` and `bottom_right` fields are of type `Point<f64>` . The `x` and `y` fields are of type `f64` . The layout of `Rectangle<f64>` also shows the byte offset from the starting address on the left side of each field. This will help us understand the assembly code generated for the function.

00 | top_left.x	0.0
08 | top_left.y	0.0
16 | bottom_right.x	42.0
24 | bottom_right.y	84.0

The following code is generated for the `static_dispatch_pair` function. The function is inlined and the code for the `area_pair_static` and `area` functions have been inlined. The assembly code is annotated with the comments that help map it to the Rust code. The generated code uses vector instructions to speed up the computation of the two area calculations. The two area tuples are returned in the `xmm0` and `xmm1` registers.

```
; Input parameters:
; a: Rectangle<f64> in rdi
; b: Rectangle<f64> in rsi
; Output parameters:
; (f64, f64) in (lower xmm0, lower xmm1)

example::static_dispatch_pair:
        movupd   xmm1, xmmword ptr [rdi + 8] ; Vector load a.top_left.y and a.bottom_right.x
                                             ; xmm1 = (a.top_left.y, a.bottom_right.x)
        movsd    xmm0, qword ptr [rdi + 24]  ; Load a.bottom_right.y into lower xmm0
        movhpd   xmm0, qword ptr [rdi]       ; Load a.top_left.x into upper xmm0
                                             ; xmm0 = (a.top_left.y, a.top_left.x)
        subpd    xmm1, xmm0                  ; xmm1 = (a.top_left.y - a.bottom_right.y, a.bot
                                             ; xmm1 = (a:height, a:width)
        movapd   xmm0, xmm1                  ; xmm0 = (a:height, a:width)
        unpckhpd xmm0, xmm1                  ; xmm0 = (a:width, a:width)
                                             ; xmm1 = (a:height, a:height)
        mulsd    xmm0, xmm1                  ; lower xmm0 = a:width * a:height = a:area

        movupd   xmm2, xmmword ptr [rsi + 8] ; Vector load b.top_left.y and b.bottom_right.x
                                             ; xmm2 = (b.top_left.y, b.bottom_right.x)
        movsd    xmm1, qword ptr [rsi + 24]  ; Load b.bottom_right.y into lower xmm1
        movhpd   xmm1, qword ptr [rsi]       ; Load b.top_left.x into upper xmm1
                                             ; xmm1 = (b.top_left.y, b.top_left.x)
        subpd    xmm2, xmm1                  ; xmm2 = (b.top_left.y - b.bottom_right.y, b.bot
                                             ; xmm2 = (b:height, b:width)
        movapd   xmm1, xmm2                  ; xmm1 = (b:height, b:width)
        unpckhpd xmm1, xmm2                  ; xmm1 = (b:width, b:width)
                                             ; xmm2 = (b:height, b:height)
        mulsd    xmm1, xmm2                  ; lower xmm1 = b:width * b:height = b:area
        ret                                  ; Return (a:area, b:area) in (xmm0, xmm1)
```

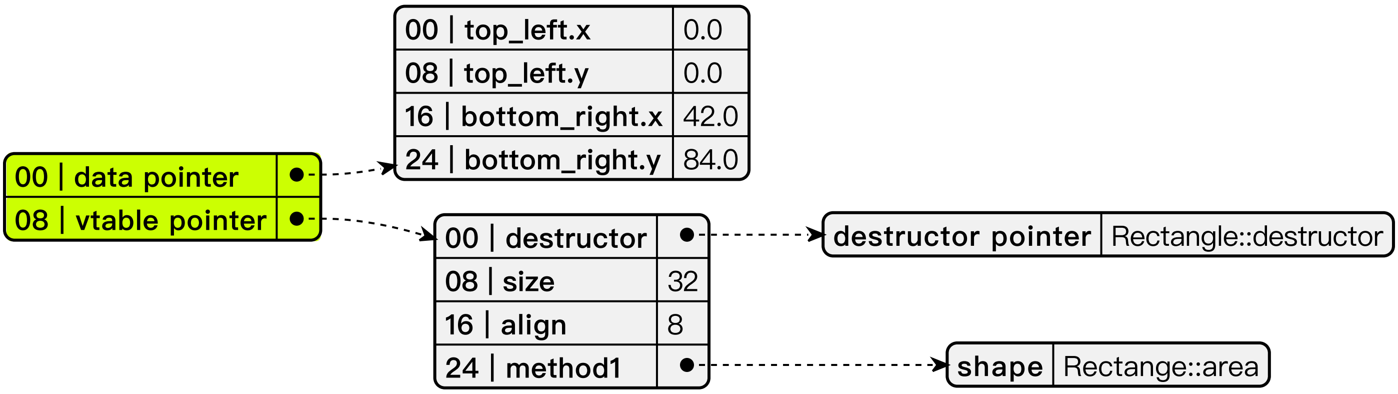
Dynamic dispatch

```
pub fn area_pair_dynamic(a: &dyn Shape<T = f64>, b: &dyn Shape<T = f64>) -> (f64, f64) {
    (a.area(), b.area())
}
```

The `area_pair_dynamic` function takes two trait objects as parameters. The function body calls the `area` method on the `a` and `b` parameters. The compiler generates code for the function referencing the trait object. There is no reference to the concrete types.

The layout of the trait object is shown below. The trait object is a tuple of two pointers. The first pointer is the address of the object. The second pointer is the address of the vtable of the object. The vtable contains the addresses of the methods of the trait implemented by the object.

In addition to the function pointers, the vtable also contains the a pointer to the drop function, the size, and the alignment of the concrete type. The drop function is called when the object goes out of scope. This typically happens when a trait object is contained in a smart pointer like `Box` . If the `Box` is dropped, the drop function is called on the object. The size and alignment of the concrete type are used to free memory for the object.



The following code is generated for the `area_pair_dynamic` function. The code has been annotated with comments that help map it to the Rust code.

The compiler passes each trait object pointer via two registers. One register carries the address of the object. The other register carries the address of the vtable of the object. The generated code obtains the address of the `area` function from the vtable of the object. It sets the `self` parameter to the address of the object. The `area` function is then called using the `area` function pointer. The `area` function is called twice, once for each object.

```
; Input parameters:
; rdi contains the address of the first object (a).
; rsi contains the address of the vtable of the first object (a)
; rbx contains the address of the second object (b).
; rcx contains the address of the vtable of the second object (b)
```

```
; Output parameters:
; Tuple<f64, f64> in (xmm0, xmm1)

example::area_pair_dynamic:
        push    r14                     ; Save previous r14
        push    rbx                     ; Save previous rbx
        push    rax                     ; Save previous rax
        mov     r14, rcx                ; r14 = address of the vtable of b
        mov     rbx, rdx                ; rbx = b (Get the address of b)

        ; Dynamic dispatch:
        ; rdi contains address of a (self)
        ; rsi contains address of the vtable for a's type
        call    qword ptr [rsi + 24]    ; Obtain the address of the area function from the v
        movsd   qword ptr [rsp], xmm0   ; Save a.area() return value in a local variable

        mov     rdi, rbx                ; rdi = address of b

        ; Dynamic dispatch:
        ; rdi contains address of b (self)
        ; r14 contains address of the vtable for b's type
        call    qword ptr [r14 + 24]    ; Obtain the address of the area function from the v
        movaps  xmm1, xmm0              ;Save b.area() return value in xmm1

        movsd   xmm0, qword ptr [rsp]   ;Get a.area() return value from a local variable
        add     rsp, 8                  ;Remove local variable from stack
        pop     rbx                     ;Restore previous rbx
        pop     r14                     ;Restore previous r14
        ret                             ;Return a.area() and b.area() as (xmm0, xmm1)
```

Key takeaways

The key takeaways can be summarized as follows:

Static dispatch	Dynamic dispatch
The compiler can inline the static dispatch code for the concrete type.	The code generated for a dynamic dispatch is inefficient as the compiler has no knowledge of the concrete type and it invokes the called trait method via a function pointer contained in the `vtable` associated with the concrete type.
The compiler generates code that is specific to the concrete type. This can result in code bloat if static dispatch is being used for a lot of types that implement the applicable trait.	The compiler generates code that is works for all types that implement the referenced trait.

Static dispatch	Dynamic dispatch
On a 64-bit system, the compiler generates code that passes the concrete type as a 64-bit pointer.	On a 64-bit system, the trait object is referenced via a 64-bit fat pointer that is a tuple of the the 64-bit pointer to the object and a 64-bit pointer to the `vtable` associated with the concrete type.
In our example, we saw that the compiler inlined the calls to the `area` method.	In our example, we saw that the compiler generated code that called the `area` method via a function pointer obtained from the `vtable` .

Experiment with the Compiler Explorer

Edit the code in the Compiler Explorer see the assembly generated for the code.

Freeing memory for trait objects

Wrap the trait objects in a `Box` and see the assembly generated for the `area_pair_dynamic` function. You will see that the compiler generates code to free the memory for the trait objects. This is because the `Box` is dropped when the function returns.

```
pub fn area_pair_dynamic(a: Box<&dyn Shape<T = f64>>, b: Box<&dyn Shape<T = f64>>) -> (f64,
    (a.area(), b.area())
}
```

Auto conversion of dynamic dispatch to static dispatch

Add the following function in the code and see the assembly generated for it. You will see that in this case, the compiler is able to optimize the code and generate the same assembly as the static dispatch version. This is because the compiler knows that the type of the arguments is `Rectangle<f64>` and it can inline the code for the `area` function.

```
pub fn dynamic_dispatch_pair(a: Rectangle<f64>, b: Rectangle<f64>) -> (f64, f64) {
    area_pair_dynamic(&a, &b)
}
```

Medium • GitHub • Twitter • LinkedIn • Facebook