Where to put the turbofish

Nov 21, 2016

In a case that you need to specify the concrete type for a generic function, method, struct, or enum, rust has this special syntax that's called the turbofish. The rule of thumb is, whenever you see:

```
$ident<T>
```

in any kind of definition, then in an expression context you usually need to write:

```
$ident::<T>
```

to specify the type for the generic parameter. Here a few examples of common usage of this syntax. Divided into four sections.

Generic Function

The `std::mem::size_of()` has this signature:

```
pub fn size_of<T>() -> usize
```

Writing, for example:

```
std::mem::size_of::<u8>()
```

Will tell you what's the size of `u8` in bytes. It's kind of amazing that you can use generic in this way.

Generic Method

The `parse()` method for `str` is also one of the few times that you'll see the turbofish syntax being used. It's signature is:

```
fn parse<F>(&self) -> Result<F, F::Err> where F: FromStr
```

We can use the turbofish to tell the desired type that should be parsed from the `str`

```
"2048".parse::<u32>()
```

Another common example is `collect()` on `Iterator`. It's type signature:

```
fn collect<B>(self) -> B where B: FromIterator<Self::Item>
```

Because the compiler already know the type of `Self::Item` you're collecting, we usually didn't need to specify it. Instead we can write `_` to let the compiler infer it automaticallly. For example, collecting into a `Vec<u8>` from `Iterator<Item=u8>`can be written as:

```
[1u8, 2, 3, 4].iter().collect::<Vec<_>>()
```

And speaking of `Iterator`, you also probably forced to use the turbofish syntax when using `sum()` and `product()` method on it. Here the signature for those method:

```
fn sum<S>(self) -> S where S: Sum<Self::Item>
fn product<P>(self) -> P where P: Product<Self::Item>
```

The sum and product of the first four positive integers can be calculated as:

```
[1, 2, 3, 4].iter().sum::<u32>()
[1, 2, 3, 4].iter().product::<u32>()
```

Generic Struct

In case of the compiler can't infer enough information when creating a generic struct, you can also use the turbofish syntax. For example, `Vec` was defined as:

```
pub struct Vec<T> { /* fields omitted */ }
```

To use the turbofish syntax, for example when creating a new `Vec` with `Vec::new()`, you can write it as:

```
Vec::<u8>::new()
```

Take a note that we put the turbofish after the `Vec`, not the `new` method. Since the ones that generic is the struct, not the method. Bear with me for this one, since it's still follow the rule of thumb above.

And yes, you can do this to all rust collections. See `HashSet` for another example. Below we create a new `HashSet` with 10 capacity.

```
std::collections::HashSet::<u8>::with_capacity(10)
```

And the `HashSet` signature itself:

```
pub struct HashSet<T, S = RandomState> { /* fields omitted */ }
```

Still, it all follows the rule of thumb above.

Generic Enum

This is ones that didn't follow our rule of thumb. Historically speaking, enum in rust isn't scoped by their enum type, hence, we put the turbofish after the enum variant, not after their type. To make it clear let's see how we use turbofish for `Result` which is an enum defined as:

```
#[must_use]
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

We use it like this

```
Result::Ok::<u8, ()>(10)
Result::Err::<u8, ()>(())
```

See that we put the turbofish after `Ok` and `Err`, the variant, not after the `Result` type itself. And also, since that variant was available in prelude, actually we can just write it as follow:

```
Ok::<u8, ()>(10)
Err::<u8, ()>(())
```

The same also applies to `Option`. Which also just an enum defined as:

```
pub enum Option<T> {
    None,
    Some(T),
}
```

Unwrapping a `None` value, for example, can be written as:

```
None::<u8>.unwrap()
```

Closing

It's a nice thing to have this syntax, but remember, you can always bind the value to a variable using the `let` syntax and declare the type explicitly. For an alternative, there's also a feature in nightly that combine the best of those two world. See the type ascription tracking issue for details.

Reference

- Generics in the rust book
- Rust syntax index
- Type Ascription RFC, Rendered
- Tracking issue for Type Ascription
