

`CoerceUnsized` was implemented on something that isn't a struct.

Erroneous code example:

```
#![feature(coerce_unsized)]
use std::ops::CoerceUnsized;

struct Foo<T: ?Sized> {
    a: T,
}

// error: The type `U` is not a struct
impl<T, U> CoerceUnsized<U> for Foo<T> {}
```

`CoerceUnsized` can only be implemented for a struct. Unsized types are already able to be coerced without an implementation of `CoerceUnsized` whereas a struct containing an unsized type needs to know the unsized type field it's containing is able to be coerced. An [unsized type](#) is any type that the compiler doesn't know the length or alignment of at compile time. Any struct containing an unsized type is also unsized.

The `CoerceUnsized` trait takes a struct type. Make sure the type you are providing to `CoerceUnsized` is a struct with only the last field containing an unsized type.

Example:

```
#![feature(coerce_unsized)]
use std::ops::CoerceUnsized;

struct Foo<T> {
    a: T,
}

// The `Foo<U>` is a struct so `CoerceUnsized` can be implemented
impl<T, U> CoerceUnsized<Foo<U>> for Foo<T> where T: CoerceUnsized<U> {}
```

Note that in Rust, structs can only contain an unsized type if the field containing the unsized type is the last and only unsized type field in the struct.