

`CoerceUnsized` was implemented on a struct which contains more than one field with an `unsized` type.

Erroneous code example:

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

struct Foo<T: ?Sized, U: ?Sized> {
    a: i32,
    b: T,
    c: U,
}

// error: Struct `Foo` has more than one unsized field.
impl<T, U> CoerceUnsized<Foo<U, T>> for Foo<T, U> {}
```

A struct with more than one field containing an `unsized` type cannot implement `CoerceUnsized`. This only occurs when you are trying to coerce one of the types in your struct to another type in the struct. In this case we try to impl `CoerceUnsized` from `T` to `U` which are both types that the struct takes. 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`.

`CoerceUnsized` only allows for coercion from a structure with a single `unsized` type field to another struct with a single `unsized` type field. In fact Rust only allows for a struct to have one `unsized` type in a struct and that `unsized` type must be the last field in the struct. So having two `unsized` types in a single struct is not allowed by the compiler. To fix this use only one field containing an `unsized` type in the struct and then use multiple structs to manage each `unsized` type field you need.

Example:

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

struct Foo<T: ?Sized> {
    a: i32,
    b: T,
}

impl <T, U> CoerceUnsized<Foo<U>> for Foo<T>
    where T: CoerceUnsized<U> {}

fn coerce_foo<T: CoerceUnsized<U>, U>(t: T) -> Foo<U> {
    Foo { a: 12i32, b: t } // we use coercion to get the `Foo<U>` type we need
}
```