Note: this error code is no longer emitted by the compiler.

There are various restrictions on transmuting between types in Rust; for example types being transmuted must have the same size. To apply all these restrictions, the compiler must know the exact types that may be transmuted. When type parameters are involved, this cannot always be done.

So, for example, the following is not allowed:

```
use std::mem::transmute;

struct Foo<T>(Vec<T>);

fn foo<T>(x: Vec<T>) {
    // we are transmuting between Vec<T> and Foo<F> here
    let y: Foo<T> = unsafe { transmute(x) };
    // do something with y
}
```

In this specific case there's a good chance that the transmute is harmless (but this is not guaranteed by Rust). However, when alignment and enum optimizations come into the picture, it's quite likely that the sizes may or may not match with different type parameter substitutions. It's not possible to check this for *all* possible types, so `transmute()` simply only accepts types without any unsubstituted type parameters.

If you need this, there's a good chance you're doing something wrong. Keep in mind that Rust doesn't guarantee much about the layout of different structs (even two structs with identical declarations may have different layouts). If there is a solution that avoids the transmute entirely, try it instead.

If it's possible, hand-monomorphize the code by writing the function for each possible type substitution. It's possible to use traits to do this cleanly, for example:

```
use std::mem::transmute;

struct Foo<T>(Vec<T>);

trait MyTransmutableType: Sized {
    fn transmute(_: Vec<Self>) -> Foo<Self>;
}

impl MyTransmutableType for u8 {
    fn transmute(x: Vec<u8>) -> Foo<u8> {
        unsafe { transmute(x) }
    }
}

impl MyTransmutableType for String {
    fn transmute(x: Vec<String>) -> Foo<String> {
        unsafe { transmute(x) }
    }
}

// ... more impls for the types you intend to transmute

fn foo<T: MyTransmutableType>(x: Vec<T>) {
```

```
    let y: Foo<T> = <T as MyTransmutableType>::transmute(x);
    // do something with y
}
```

Each impl will be checked for a size match in the transmute as usual, and since there are no unbound type parameters involved, this should compile unless there is a size mismatch in one of the impls.

It is also possible to manually transmute:

```
# use std::ptr;
# let v = Some("value");
# type SomeType = &'static [u8];
unsafe {
    ptr::read(&v as *const _ as *const SomeType) // `v` transmuted to `SomeType`
}
# ;
```

Note that this does not move `v` (unlike `transmute`), and may need a call to `mem::forget(v)` in case you want to avoid destructors being called.