# Type Conversions



## Understand type conversions

- In general, Rust does not perform automatic conversion between types.
- This includes integral types, even when the transformation is "safe":

```
fn foo() {
    let x: i16 = 42;
    let y: i32 = x;
}
```

# Three Categories of Type Conversions

Rust type conversions fall into three categories:

1. **semi-automatic**: explicit casts between values using the `as` keyword
2. **automatic**: implicit *coercion* into a target type
3. **manual**: user-defined type conversions provided by implementing the `From` and `Into` traits

- 1. and 2. above don't apply to conversions of user defined types (with a couple of exceptions such as C-like enums).



# User-Defined Type Conversions

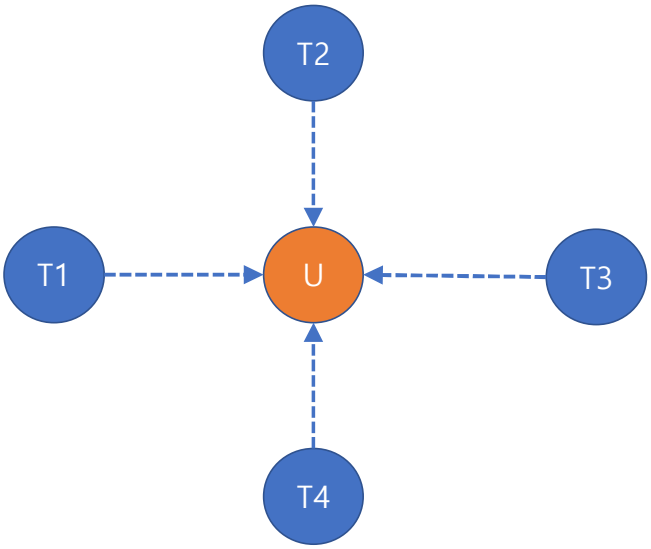
- Conversions between user-defined types is encapsulated as a set of related generic traits.

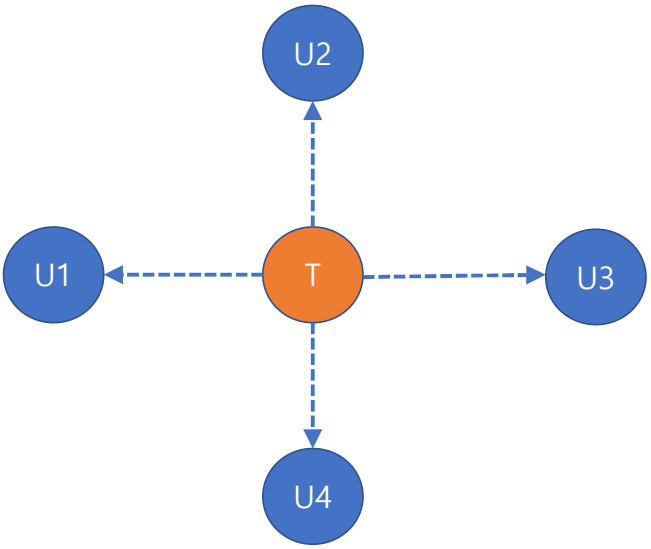
  The four relevant traits:
  - `From<T>`: Items of this type can be built from items of type T.
  - `Into<T>`: Items of this type can converted into items of type T.
  - `TryFrom<T>`
  - `TryInto<T>`

- Check the standard library to see already defined **impl From<T>**s.

```
impl From<&str> for String
impl<T: Clone> From<&[T; N]> for Vec<T>
impl<T: Clone> From<&[T]> for Vec<T>
impl From<&str> for Vec<u8>
...
```

# From *vs.* Into



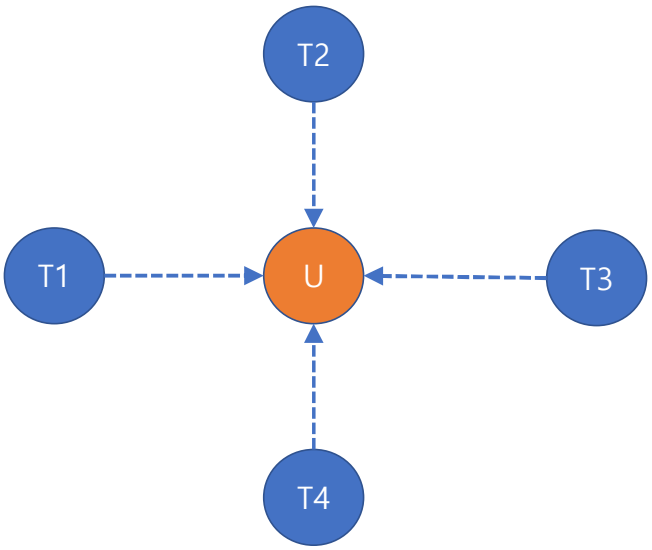




`impl<T, U> From<T> for U`

`impl<T, U> Into<U> for T`



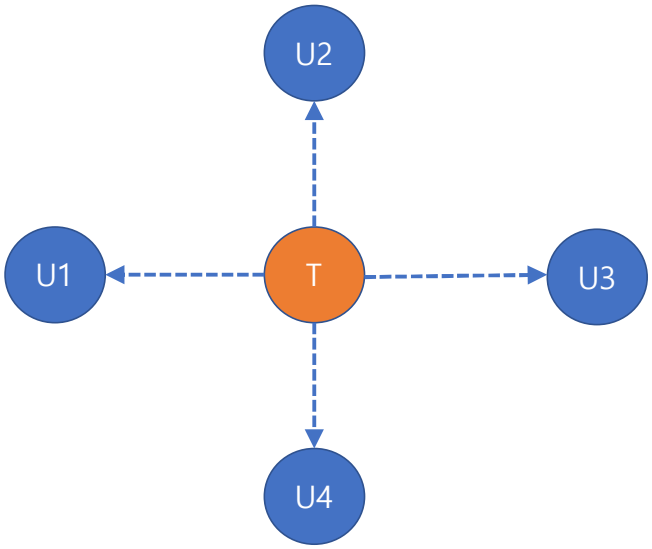
# From trait




```rust
impl<T, U> From<T> for U {
    // associated function
    fn from(value: T) -> U {
        todo!()
    }
}
```

`impl<T, U> From<T> for U`

# Into trait




```
impl<T, U> Into<U> for T {
    // method
    fn into(self) -> T {
        todo!()
    }
}
```

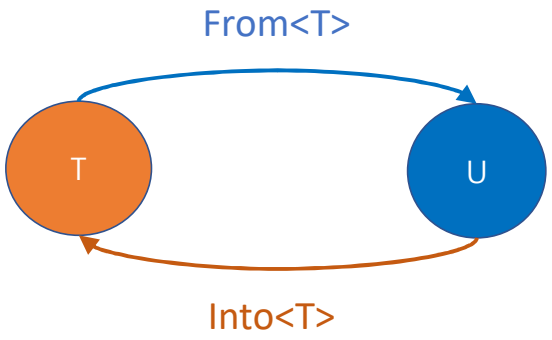
`impl<T, U> Into<U> for T`



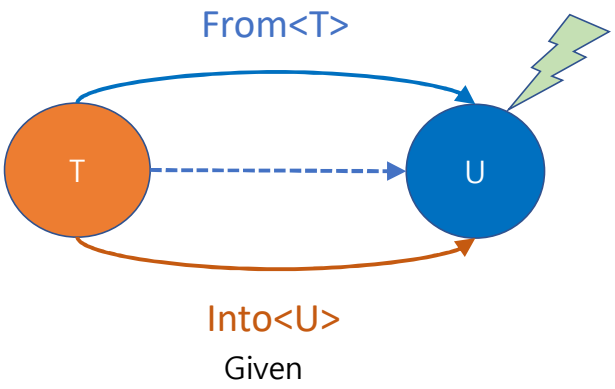
# Symmetry Trap

- The type conversion traits have an obvious symmetry:
  - if T ⟹ U is possible, it should also be possible to U ⟹ T.

Is it true???



# Which one to implement for T -> U?



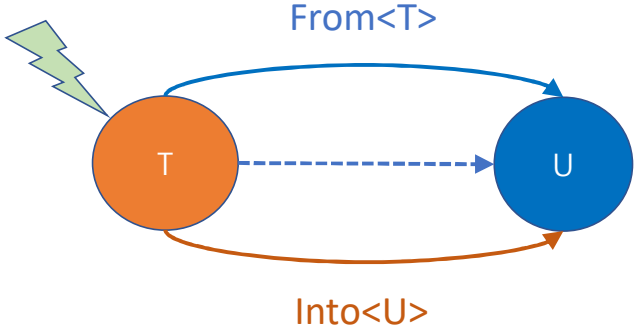

Given

```
impl<T, U> From<T> for U
where
    T: Into<U>,
{
    fn from(value: T) -> U {
        value.into()
    }
}
```



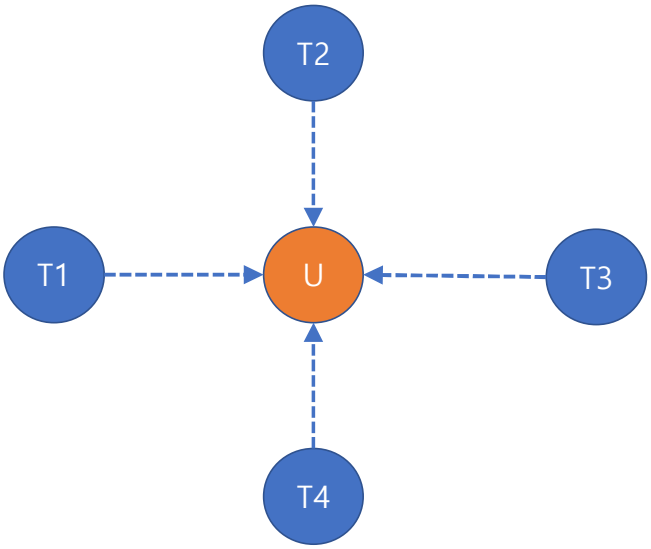
# Which one to implement for T -> U?

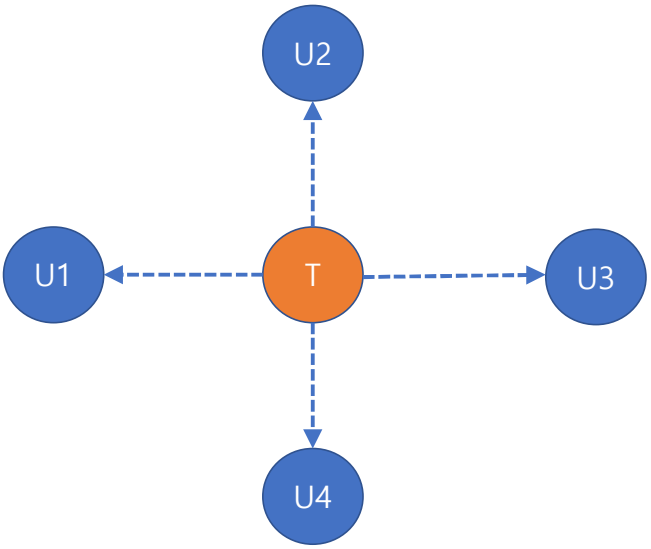
Given




```
impl<T, U> Into<U> for T
where
    U: From<T>,
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

# Which one to implement for T -> U?




```
impl<T, U> From<T> for U
```

```
impl<T, U> Into<U> for T
```



# Implement One, Get One free

## (Only implement the **From** trait)

- Rust automatically provides `Into` from a `From` implementation.

```
impl<T, U> Into<U> for T
where
    U: From<T>,
{
    fn into(self) -> U {
        U::from(self)
    }
}
```

Given → `U: From<T>,`

Compiler automatically generates

"I can implement `Into<U>` for a type T whenever U already implements `From<T>`".

# Reflexive Implementation

- "Given a T, I can get a T"

```
impl<T> From<T> for T {
    fn from(t: T) -> T {
        t
    }
}
```

- What good this is for?



# Casts by `as`

*For consistency and safety you should prefer* ***`from / into` conversions to `as` casts***

- The `as` keyword to perform *safe* explicit casts between some pairs of types.
  - Casting between any numeric-types is always valid.

```
let x: u32 = 9;
let y = x as u64; // widening
let z: u64 = x.into();

let x: u32 = std::u32::MAX; // 4294967295
let y = x as u16;           // 65535
let y: u16 = x.into(); // oops! From(u32) not implemented for u16
```

# Casts by `as` Examples

- The pairs of types that can be converted using `as` is a fairly limited set.
  - The only user-defined types it includes are "C-like" enums (those that have an associated integer value).

```
let one = true as u8;
let at_sign = 64 as char;
let two_hundred = -56i8 as u8;
```

```
enum Fruit { Apple = 10,
             Banana, Orange }
let x = Fruit::Apple as u32;
```

  - Perhaps surprisingly, it is safe to cast *raw pointers* to and from integers, and to cast between pointers to different types subject to some constraints.

```
let a = 300 as *const char; // `a` is a pointer to location 300.
let b = a as u32;
```



# Implicit Coercion

- Coercion between types is implicit and has no syntax of its own.
- Any implicit coercion can be forced with an explicit `as`, but the converse is not true.
- Most of the coercions involve silent conversions of pointer and reference types in ways that are sensible and convenient for the programmer.

# Implicit Coercion Cases (1)

- Removing mutability from a reference
  - Converting `&mut T` to `&T`
  - Converting `*mut T` to `*const T`
- Converting a reference to a raw pointer
  - `&T` to `*const T`
  - `&mut T` to `*mut T`
- Converting a closure without capture into a bare function pointer

```
let fp: fn(i32) -> i32 = |x| x + 1;
```

- Converting an array to a slice

```
let slice: &[i32] = &[1, 3, 5, 7, 9];
```



# Implicit Coercion Cases (2)

- Converting a concrete item to **trait object**

```
let animal: &dyn Animal = &Dog;
```

- Converting an item lifetime to a "shorter" one

```
fn choose_first<'a: 'b, 'b>(first: &'a i32, _: &'b i32) -> &'b i32 {
    first
}
```

- Converting a reference to the smart pointer into a reference to an item the smart pointer contains (due to transitive *Deref coercion*)

```
let boxed_string = Box::new(String::from("hello"));
let str_ref: &str = &boxed_string;
```