C/CPS 506

Compara **Languages**

Prof. Ale

Topic 11: Structs, enums, generic types, traits

Ryerson University

Course Administration

Reminder: Types & Literals

4 Scalar types:

Integer – u8, u16, u32, u64, usize, i8, i16, **i32**, i64, isize

Floating Point – f32, ___

Boolean – bool (true, f

Character – Unicode: ‘Z’, ‘a’, ‘&’ 5}’, etc

2 Compound types:

Tuple – heterogeneous

Arrays – homogeneous

struct

Structures: Similar to C/C++

Can contain heterogeneous data, just like tuples:

- Struct fields separated by commas.

in a comma.

- Declare struct as follows:
- Indicate values for each field
- Again, separate by commas.
- Need not declare in same order!

Structures

When declaring, must assign values to all fields:

Structures: Accessing Fields

- If we want to change struct
  lues, must declare using `mut`
  tire struct must be mutable
  the dot operator to access

Structures: As Return Values

- Return type is the struct name
- Function ends in an expression, returns a Person struct.

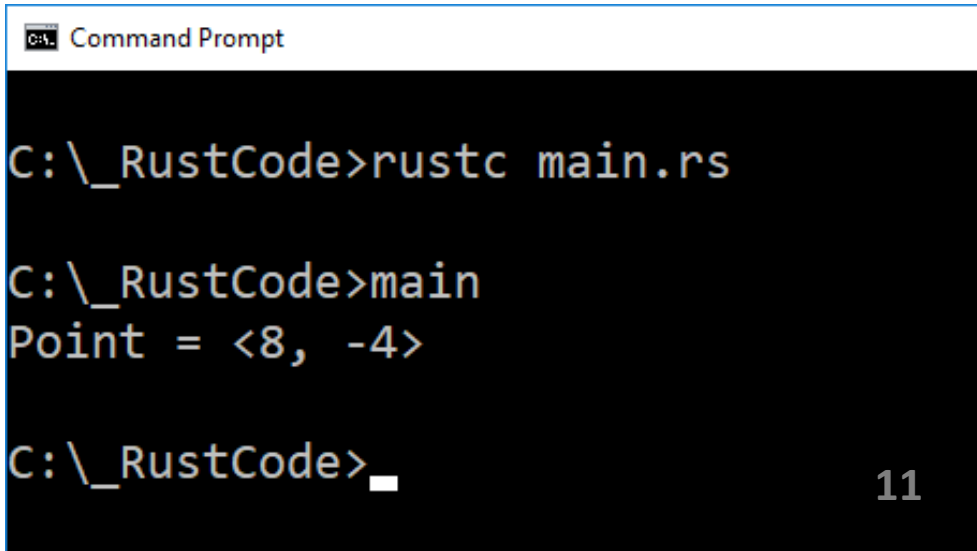
Structures: Parameter Shorthand

If the name of the parameter is the same as the struct field, we can create as seen here:

Tuple Structures

Notice:

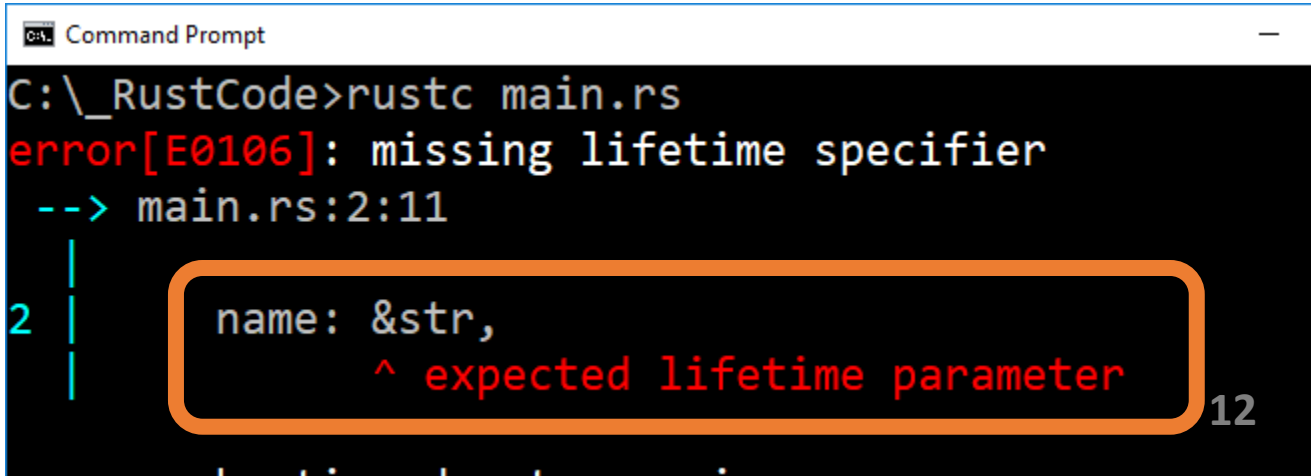
- Fields are not named.
- Access them using *numeric* indexes like a normal tuple.



Structures, References, Lifetimes

Notice:

name is declared as a
g slice (**&str**)
able to
ze it with a literal



Structures, References, Lifetimes

Danger:

- This reference could point to data owned lse
  angling pointer problem
- Struc be in scope, while the data referenced has gone out of scope.
- We can annotate lifetimes here

Structures, References, Lifetimes

All we're saying here is that an instance of Person won't outlive the ds in its name field.

In oth **`ser1`** cannot outlive **`"Tim"`**

Methods VS Functions

Like functions, methods can accept parameters, return a value, and co un when called.

Unlike functions, methods ar within the context of a struct (or trait, or enum, as we'll also see).

`Rect` struct containing two fields, width and height

- Method **`area`** *implemented* for **`Rect`** structs. Use `impl` block.
- `&self` refers to the calling struct **`width`** and `self.height`

ce st
`this` in Java.

- Call area method for r1.
- No args since we're just going to reference self

Why Methods?

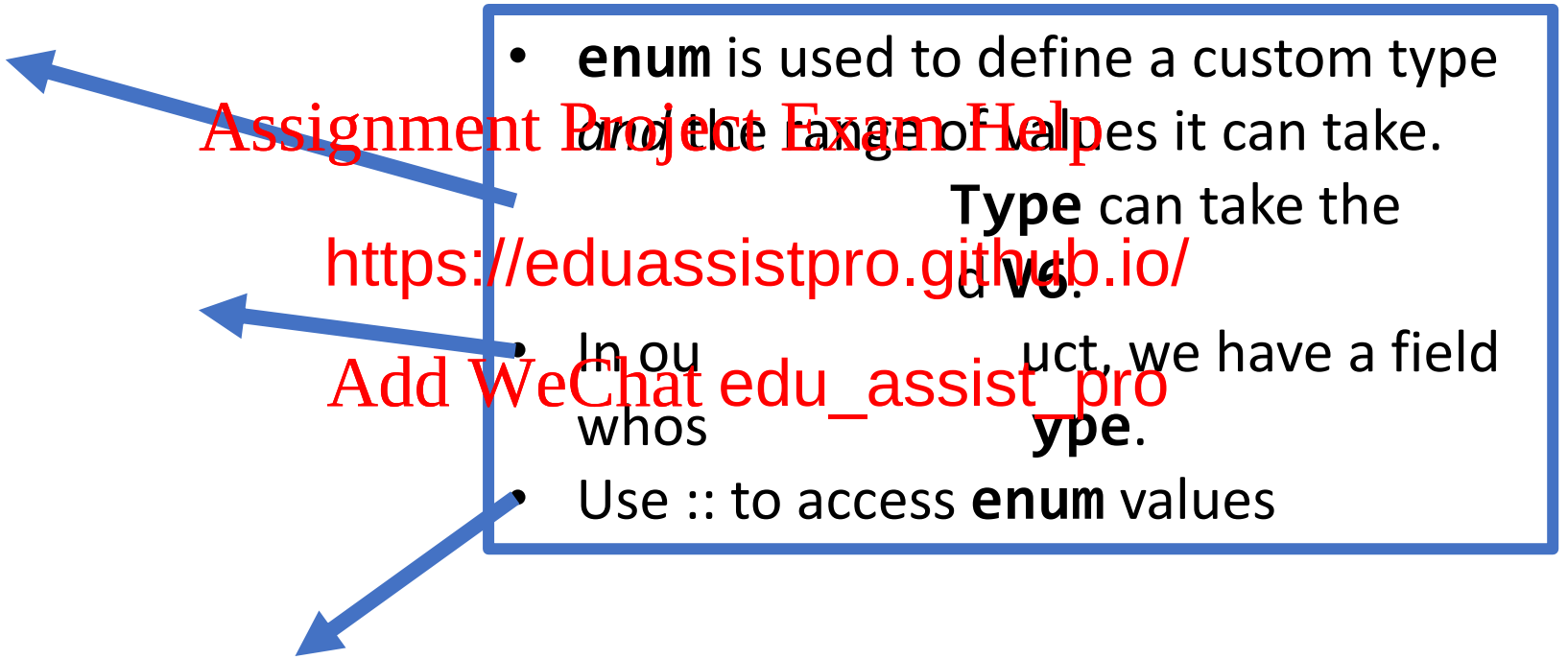
- Keep behavior specific to a struct organized.
- Simplify arguments through `self` keyword.

Function VS Method:

- Strictly speaking, we can have d *functions* as well
  rence s s that
  doesn't use the `&self` parameter, while a method does.

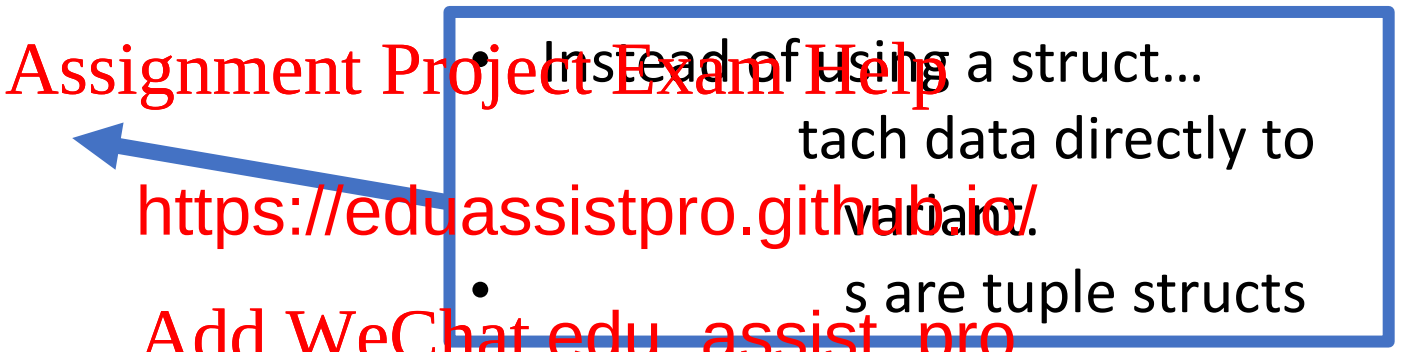
There’s more to methods, but we’ll leave it here for now. Ask about traits.

Enums



- **enum** is used to define a custom type and the range of values it can take.
- In our project, we have a field called **Type** which can take the values **Vo**.
- Use :: to access **enum** values

More Concisely



Pattern Matching

(with *Enums!*)

Pattern Matching with `match`

IpType enum from before

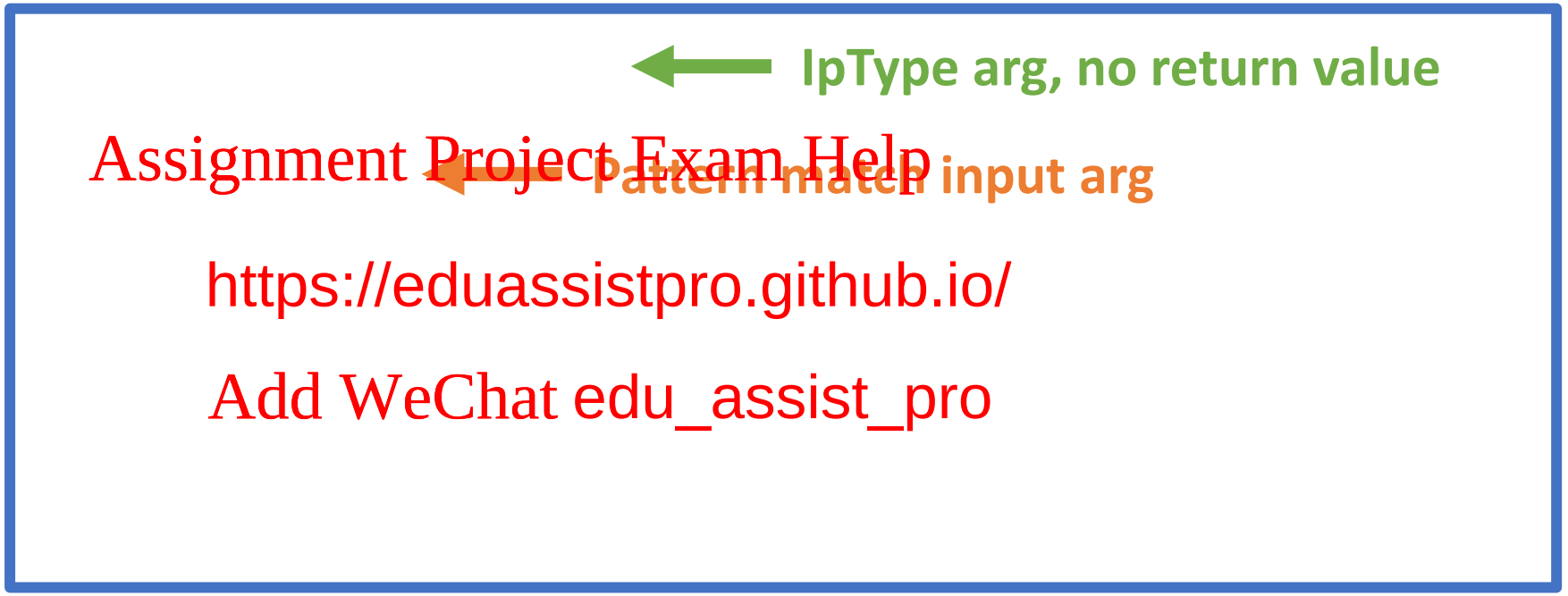
Declare two IpTypes, one V4 variant and one V6 variant.

Pattern Matching with `match`


```
print_ip(v4);
print_ip(v6);
```

- Call a function for printing IP addresses.
- **Problem:** How we print depends on version

Pattern Matching with `match`



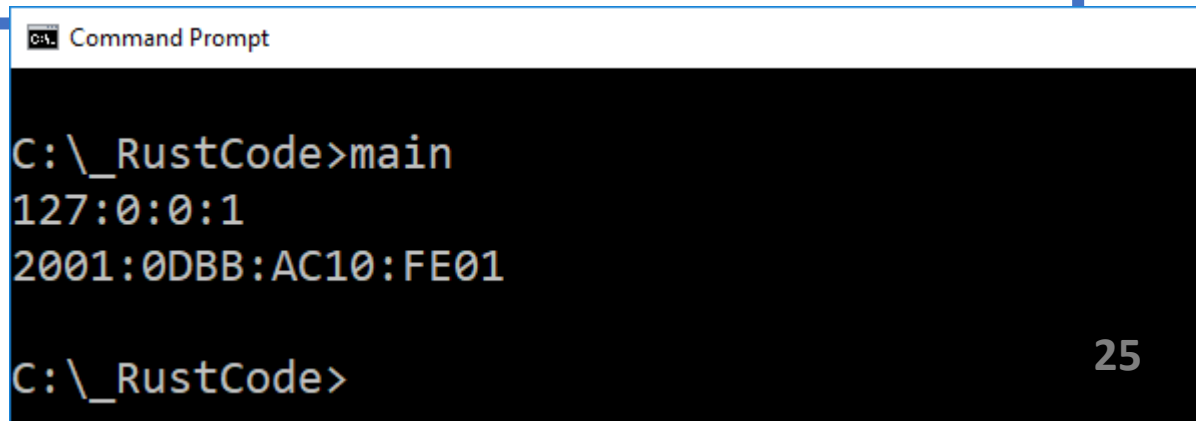
Pattern Matching with `match`

Each `match` branch can be an entire block

V4 case. Notice four values for members of V4

V6 case.

Single value for String



Rules: `match`

Like Haskell, match structures must be exhaustive:

Rules: `match`

Also like Haskell, underscores are wild:

Nested Clauses

- If/else inside a match case
- Result of if/else is an expression
- Will result in a String

Enum: `Option`

Rust defines an *Option* enum. Used to test if a value is *something* or *nothing*

Just like **`Maybe`** in Haskell!

What's wrong with **`NULL`**?

- It's easy to accidentally use a NULL value
- Dereference NULL pointers, use NULL value in computation
- May or may not cause run-time errors
- The concept of NULL is pervasive. Easy mistake to make.

Option

- NULL is a very useful concept.
- The problem is its implementation
- Default integer value instead of unique type.
- Haskell implements **Option**


```
data Maybe a
  = Just a
  | Nothing
```

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

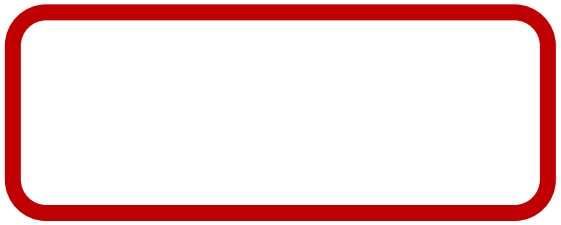
a is a type variable

T is the Rust equivalent, a generic type parameter

Option

It’s built into Rust, we can just use it

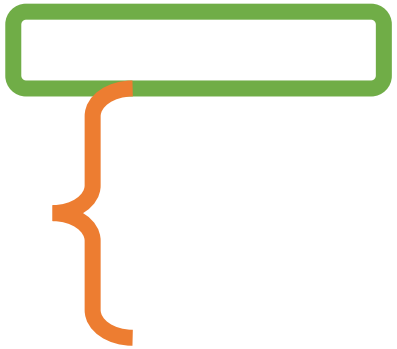
- g **None**, we specify a type.
- fer a type from **None**.



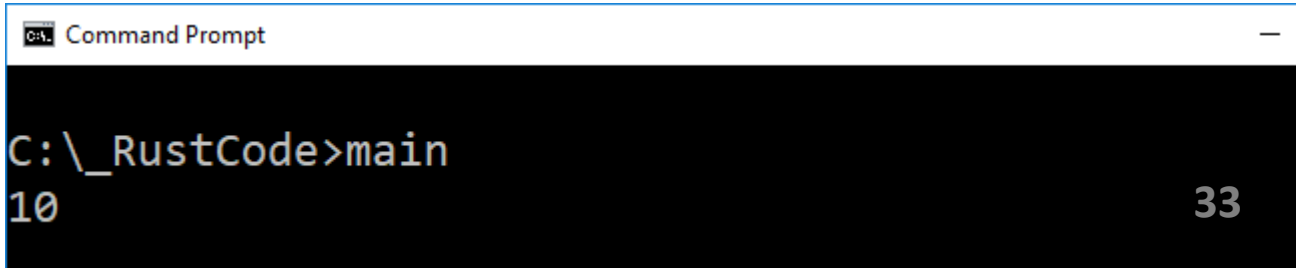
- This highlights the difference between u **`Option`** or **`Maybe`**:
- We can't implicitly convert **`Option`** for arithmetic or other operations.
- Thus, we get a compile error
- With **`NULL`**, code compiles perfectly fine because it's just a value.
- Causes runtime errors that are more dangerous, and trickier to track down.
- The burden is still on the programmer to identify when **`Option`** should be used.

Pattern Matching with `match`

`T` from **`Some(T)`**



- Very similar to Haskell
  alue x from Some(x)
  some default value for None
  , we say 0. Sensible because using v1_val in an addition.
- Notice we're treating the match structure as an expression
- Just like we saw with if/else-if/else



Generic Types

Generic Types

Consider a function that finds the largest item in an array:

- Rust is strongly, statically typed.
- function will only work for ys of type i32.

  about: i8, i16, i64, isize, u8, 32, u64, usize, f32, f64, or even char?
- ***Do we really need a different function for each type?***

With what we know of Rust, you'd be forgiven for thinking that we did.

Generic Types

It can be done!

- T is a generic data type.
- We're not done yet though:

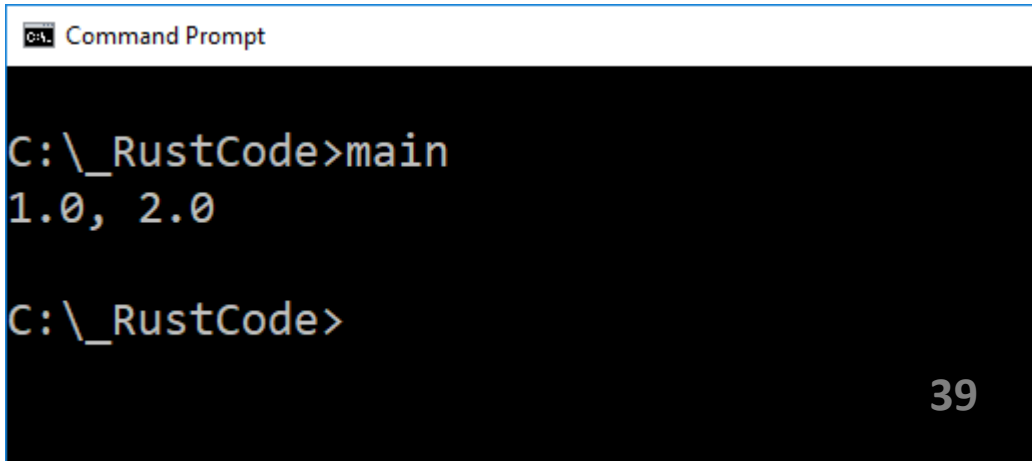
- Function won't work on all possible types T could take.
- We're doing comparison, T must be a type that can be ordered.
- **`std::cmp::PartialOrd`** is a ***trait***.
- **Recall:** Traits in Rust are directly inspired by type classes in Haskell

Let’s see some more about generic types and traits, then we’ll come back to this max_val function.

Generic Types: Structs

- Here, it doesn't matter what type T is
- We're not operating on it in any way

ints declared using int, float, and even Strings

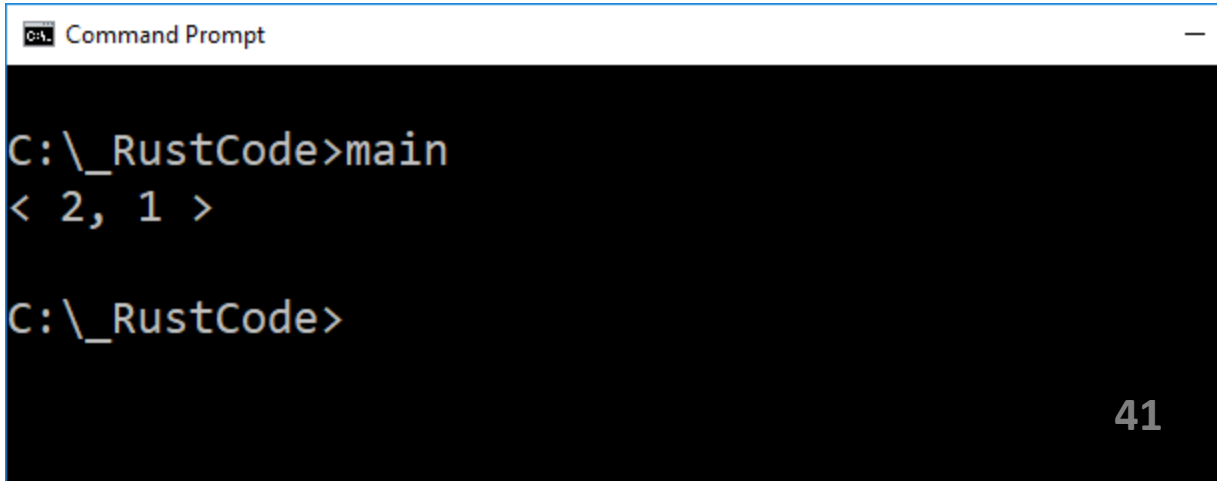


However…

Type of x and y must match

Generic Types: Struct Methods

- Here, we're moving ownership (no &)
  OK, we're sending a new Point
  o main
  't matter what type T is,
  lowed to create a new Point



Generic Types: Struct Methods

- Here, we implement a **vec_len** method for Point.
- However, it is only implemented when T is f64.
- If we left it generic, we'd get a compile error.

fined for all possible T

d…

e're calling it as a method, rather than a function.

- **sqrt()** is implemented over type f64

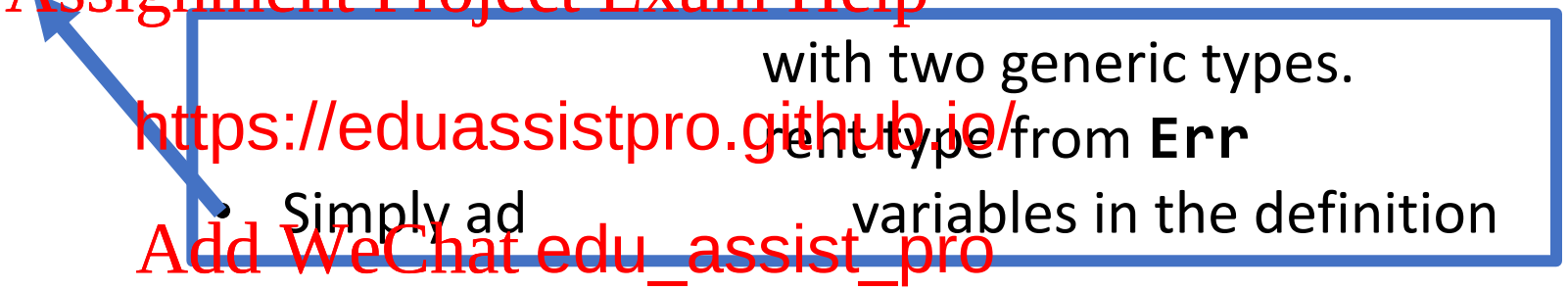
Struct with two values of generic type T

Method **`vec_len`** implemented for Point, but only when T is `f64`

Method **`swap_coords`** implemented for T. Old point is destroyed, new wapped coords is returned

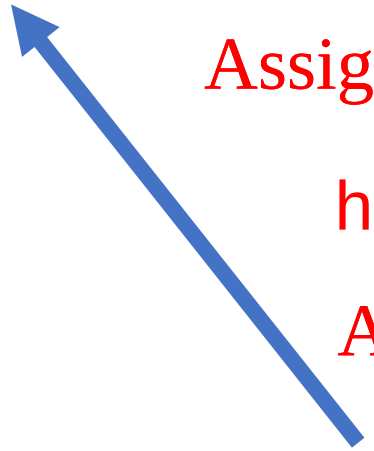
- Add as many as we want
- They can be implemented for specific types, or generic types.
- Whatever the case, operations on the type must be defined.

Generic Types: Enums



- **Problem:** These types can't stay generic forever.
- Rust is statically typed – has to know concrete type at compile time if we're using an instance of Result.

Generic Types: Enums



- By providing 5 with Ok, that tells rust the type of T.
- It says nothing about E.

Problem? Can Rust infer types?

Generic Types: Enums

Now, for each Result variable, we've
e type of T and E.

But… Why should it matter what type **`E`** is, if **`_r1`** is **`Ok(T)`**?

mut

- If `n1` is mutable, it can take
  T) ***or*** `Err(E)`
  know the type
  of eac
  time.

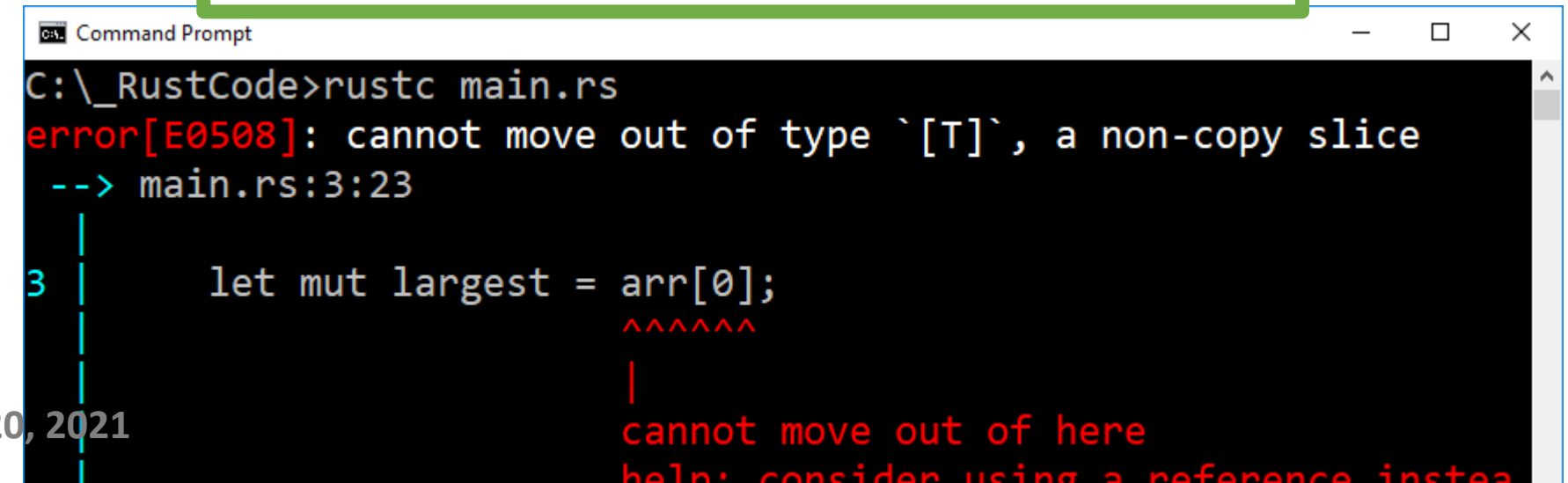
Traits

A trait tells the Rust compiler about functionality a particular type has

Let's revisit our max_val function.

- In **`max_val`**, we compare two values of type T.
- This behavior isn't defined for all possible values of T.
- Comparison is defined for types the **`PartialOrd`** trait. like Haskell's Ord type class

- Tell Ru ept any type T that implements rd trait.
- We're almost there…



```
error[E0508]: cannot move out of type `[T]`, a non-copy slice
 --> main.rs:3:23
  |
3 |     let mut largest = arr[0];
  |                       ^^^^^^
  |                       |
  |                       cannot move out of here
```


Roughly speaking:

- Not all types implementing PartialOrd n the stack
- There is still potential here for T being d on the heap (i.e. String)
- This error comes from us trying to (potentially) copy a heap variable.
- Recall, of Strings:


```
  = note: move occurs because `s` has type `std::string::String`,
which does not implement the `Copy` trait
```

```
 = note: move occurs because `s` has type `std::string::String`,
which does not implement the `Copy` trait
```

- Ownership moved from **s** to **word**!
- valid!
- different from any other
- la re used to.
- f appen with primitives

because they will simply be copied.

- The String is not copied! Its ownership is moved!

```
  = note: move occurs because `s` has type `std::string::String`,
which does not implement the `Copy` trait
```

We can fix this! We have the technology!

swap**`max_val`** will work on any type T that implements **`Copy`** and **`PartialOrd`** traits.

- Interesting, I thought Copy wasn't implemented for String?
- Except, these are not **Strings**. They're literals, which means they're **&str**
- **`Copy`** and **`PartialOrd`** ***are*** implemented for **&str**

Traits and Methods

Clun ate a method we can call that will print our Points.

- Makes sense right? Except…
- We're assuming that Rust knows how to print every type T.

  Turns out this isn't the case.

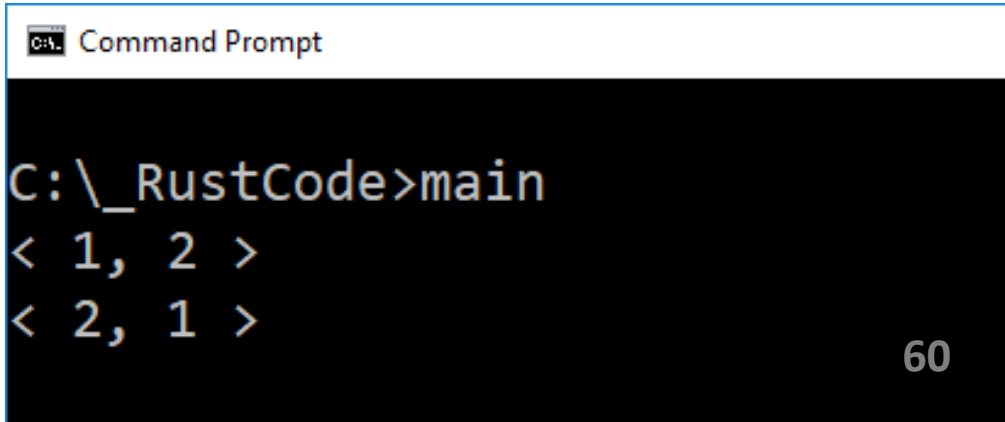
Not to worry! **`Display`** is a trait.

We know how to use traits.

First time seeing:

- Similar to a “using namespace” statement in C++

lude the Display trait in
definition.
the print method will
for any type T that
implements **`Display`**



Implement Traits For Custom Types?

Everything we've seen r as been about ***restricting*** functions or specific traits.

What if we want to create a ***new*** a trait for a certain type?

Custom Trait: `Pt3D`

```
trait PointOps {
    fn plus (&self, pt: &Self) -> Self;
}
```

- `&self` is a reference to the calling variable
- `Self` is the ***type***
- Keeps this trait ge
- Any type can implement it

- Create a new trait (using `trait` keyword) called **PointOps**.
- This trait will contain operations
  n our Pt3D data type
  nitially, we'll start with a simple
  thod for addition.
  is method is invoked from *some* type, accepts *that type* as an argument, and returns *that type*
- **`T = T.plus(T)`**
- This is just a method signature!
- What about the implementation?

Custom Trait: Pt3D

```
trait PointOps {
    fn plus (&self, pt: &Self) -> Self;
}
```


- We're implementing the PointOps trait for our Pt3D type
- Inside the `impl` we find our
  ethod definition(s)
- We create and return a new Pt3D
  ose elements are the sum of the
  D that invoked the method, and the Pt3D passed in as an argument.
- Notice this implementation is specific to Pt3D
- Sensible, since we'd need different behavior for different types

Custom Trait: `Pt3D`

```
trait PointOps {
    fn plus (&self, pt: &Self) -> Self;
}
```

- If we implement a trait for a particular data type, we are required to implement all methods
ompile error otherwise:

Custom Trait: Pt3D

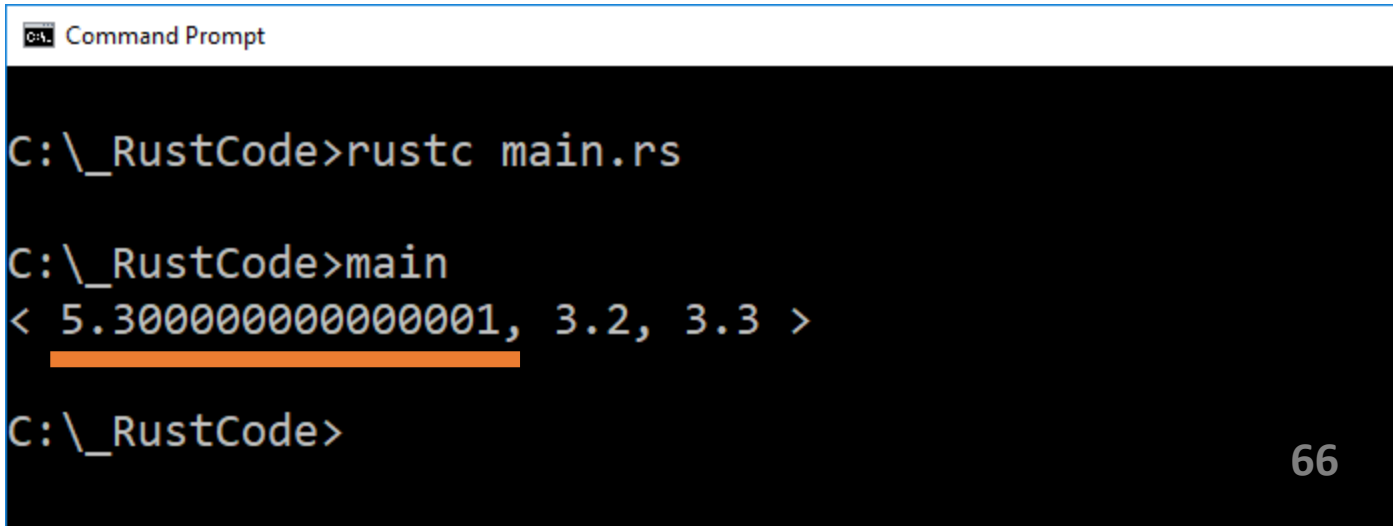
```
trait PointOps {
    fn plus (&self, pt: &Self) -> Self;
}
```

- Let's test our **plus** method
- Declare two points, call the plus method on p1, pass in p2 as rgument.
- tore the result in a new point, p3.
  nt p3.


```
fn main()
{
    let p1 = Pt3D {x: 1.1, y: 2.2, z: 3.3};
    let p2 = Pt3D {x: 4.2, y: 1.0, z: 0.0};
    let p3 = p1.plus(&p2);
    println!("< {}, {}, {} >", p3.x, p3.y, p3.z);
}
```

- **Unacceptable**. Let's create and implement a print method in our PointOps trait

Custom Trait: Pt3D



By the way:

- If we want formatted output a la printf, we can use format! Instead of println!
- Feel free to Google it, usage is fairly straightforward.

We've implemented our custom PointOps trait for Pt3D

Can we implemen **? How about Pt2D?**

Custom Trait: `Pt2D`

- Our trait can stay the same.
- It's already generic enough for Pt2D.
- `Self` can be anything!

Custom Trait: `Pt2D`

- This time, **impl** PointOps for Pt2D
- nctionality is similar
- e're dealing with a 2D point
- ead of 3D.

Custom Trait: `Pt2D`

Implement *Existing* Trait For Custom Types?

We can ***restrict*** functions or methods to types with specific traits.

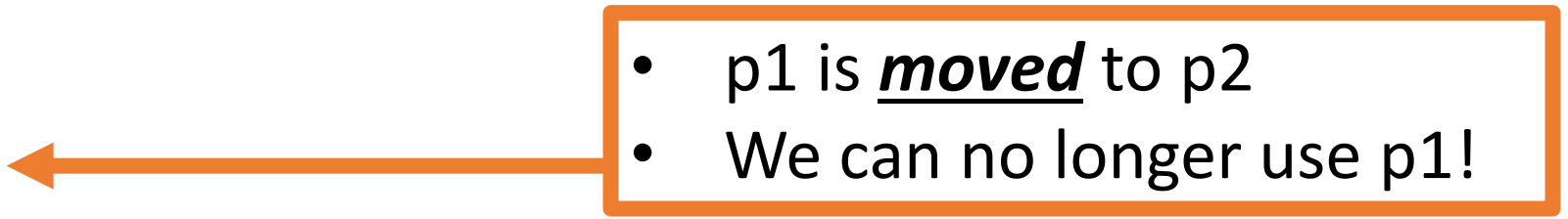
What if we want to ***implement*** an existing trait for a certain type?
For example, the Copy or Display trait for Pt

Two Options: `#derive`

Recall:

moved to p2

- We can no longer use p1!



Ownership - Three Rules:

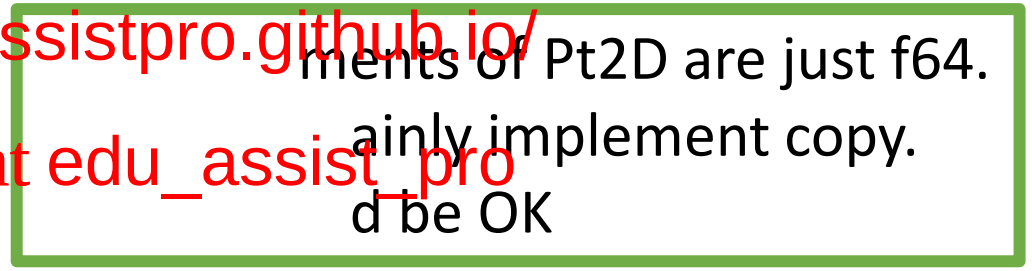
1. Each value in Rust has a variable that's called its *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value is dropped.

- p1 is ***moved*** to p2
- We can no longer use p1!

ove occurs because Pt2D
n't implement Copy.
ables are moved by default,
unless they implement Copy
- Can we implement Copy?

Implementing Copy

Rule: A struct can implement Copy if its *components* implement copy.



#derive

- Huh? We can't implement Copy without implementing Clone?
- Clone is a ***supertrait*** of Copy

#derive

- We're now *copying* instead of *moving*
- p1, p2 are different values in memory
- Can still make use of p1!

Rule: A struct can implement Copy if its components implement copy.

- We already know that String doesn't implement Copy.
- What happens if we try?

`#derive` Display?

Nope. But we can implement it ourselves!

Implement Display?

- Doing so requires us to know what methods the Display trait requires.
- The method signature for displaying a type using { } in a println! Is as follows:

Format your output as desired here

Traits are a powerful mechanism for achieving type polymorphism and custom type behavior in Rust.

Traits are directly inspired by Haskell's type classes, and it shows.

Haskell type classes

- Must (*should*) implement minimal set of operations
- Can derive existing type classes
- Can implement existing type class

Traits

- Must (*should*) implement methods described in trait definition
- Can derive existing traits
- Can implement existing traits

Fantastic Rust Reference:

https://doc.rust /second-edition/