C/CPS 506

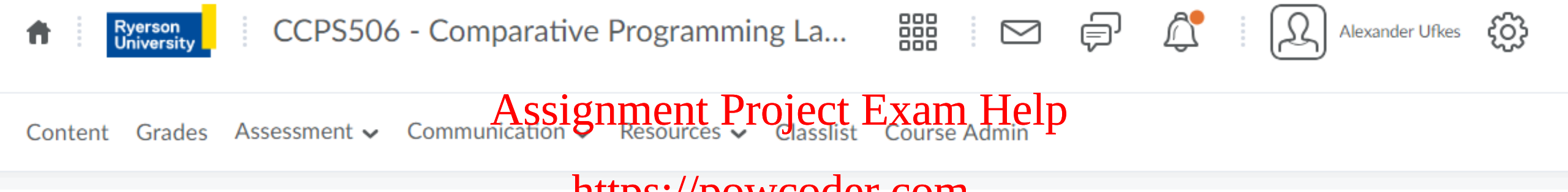
Comparative Programming Languages

Prof. Alex Ufkes

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, f64`

Boolean – `bool` (true, false)

Character – Unicode: `'Z', 'a', '&', '\u{00C5}'`, etc

2 Compound types:

Tuple – heterogeneous

Arrays – homogeneous

struct

Structures: Similar to C/C++

Can contain heterogeneous data, just like tuples:

```rust
struct Person {
    name: String,
    age: u8,
}

fn main()
{
    let user1 = Person {
        age: 85,
        name: String::from("Bill"),
    };
}
```

- Struct fields separated by commas. Even the last item ends in a comma.
- Must indicate type

- 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:

```rust
struct Person {
    name: String,
    age: u8,
}

fn main()
{
    let user1 = Person {
        age: 85,
        //name: String::from
    };
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0063]: missing field `name` in initializer of `Person`
 --> main.rs:8:17
  |
8 |     let user1 = Person {
  |                 ^^^^^^ missing `name`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0063`.

C:\_RustCode>
```

Structures: Accessing Fields

```rust
struct Person {
    name: String,
    age: u8,
}

fn main()
{
    let mut user1 = Person {
        age: 85,
        name: String::from("Bill"),
    };

    user1.age = 78;
    user1.name = String::from("Ted");
}
```

- If we want to change struct values, must declare using `mut`
- Entire struct must be mutable
- Use the dot operator to access

Structures: As Return Values

```rust
struct Person {
    name: String,
    age: u8,
}

fn main()
{
    let user1 = build_person(String::from("Tim"), 99);
    println!("Name: {}\nAge: {}", user1.name, user1.age);
}

fn build_person (name: String, age: u8) -> Person
{
    Person { age: age, name: name, }
}
```


Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
Name: Tim
Age: 99

C:\_RustCode>_
```

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

Structures: Parameter Shorthand

```rust
struct Person {
    name: String,
    age: u8,
}

fn main()
{
    let user1 = build_person(String::from("Tim"), 99);
    println!("Name: {}\nAge: {}", user1.name, user1.age);
}

fn build_person (name: String, age: u8) -> Person
{
    Person { age, name, }
}
```

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

Tuple Structures

```rust
struct Color(u8, u8, u8);
struct Pt2d(f64, f64);

fn main()
{
    let _red = Color(255, 0, 0);
    let pt = Pt2d(8.0, -4.0);

    println!("Point = <{}, {}>", pt.0, pt.1);
}
```


Notice:

- Fields are not named.
- We access them using *numeric* index, just like a normal tuple.

Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
Point = <8, -4>

C:\_RustCode>_
```

Structures, References, Lifetimes

```rust
struct Person {
    name: &str,
    age: u8,
}

fn main()
{
    let user1 = Person {
        name: "Tim",
        age: 55,
    };
}
```

Notice:

- **name** is declared as a string slice (**&str**)
- We should be able to initialize it with a literal

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0106]: missing lifetime specifier
 --> main.rs:2:11
  |
2 |     name: &str,
  |           ^ expected lifetime parameter
```

Structures, References, Lifetimes

```rust
struct Person {
    name: &str,
    age: u8,
}

fn main()
{
    let user1 = Person {
        name: "Tim",
        age: 55,
    };
}
```

Danger:

- This reference could point to data owned by something else
- Similar to the dangling pointer problem
- Struct might still be in scope, while the data referenced has gone out of scope.
- We can annotate lifetimes here

Structures, References, Lifetimes

```rust
struct Person<'a> {
    name: &'a str,
    age: u8,
}

fn main()
{
    let user1 = Person {
        name: "Tim",
        age: 55,
    };

    println!("{}, {}", user1.name, user1.age);
}
```

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

In other words, **`user1`** cannot outlive **"`Tim`"**

Methods VS Functions

Like functions, methods can accept parameters, return a value, and contain code that is run when called.

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

```rust
struct Rect {
    width: f64,
    height: f64,
}

impl Rect {
    fn area (&self) -> f64 {
        self.width * self.height
    }
}

fn main()
{
    let r1 = Rect { width: 5.0, height: 3.0 };
    println!("Area = {}", r1.area());
}
```

Rect struct containing two fields, width and height

- Method **area** *implemented* for **Rect** structs. Use `impl` block.
- `&self` refers to the calling struct
- `self.width` and `self.height` reference struct fields.
- Similar to `this` in Java.

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

```rust
struct Rect {
    width: f64,
    height: f64,
}

impl Rect {
    fn area (&self) -> f64 {
        self.width * self.height
    }
}

fn main()
{
    let r1 = Rect { width: 5.0, height: 3.0 };
    println!("Area = {}", r1.area());
}
```

Why Methods?

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

Function VS Method:

- Strictly speaking, we can have associated *functions* as well
- The difference syntactically is that a function doesn't use the **&**`self` parameter, while a method does.

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

Enums

```rust
enum IpType {
    V4,
    V6,
}

struct IpAddr {
    kind: IpType,
    address: String,
}

fn main()
{
    let ip1 = IpAddr {
        kind: IpType::V4,
        address: String::from("127.0.0.1"),
    };
}
```

- **`enum`** is used to define a custom type and the range of values it can take.
- In this case, **`IpType`** can take the values of **`V4`** and **`V6`**.
- In our **`IpAddr`** struct, we have a field whose type is **`IpType`**.
- Use :: to access **`enum`** values

More Concisely

```rust
enum IpType {
    V4(u8, u8, u8, u8),
    V6(String),
}

fn main()
{
    let home = IpType::V4(127, 0, 0, 1);
}
```

- Instead of using a struct…
- We can attach data directly to each enum variant.
- Enum values are tuple structs

Pattern Matching

(with *Enums!*)

Pattern Matching with `match`

```
enum IpType {
    V4(u8, u8, u8, u8),
    V6(String),
}

fn main()
{
    let v4 = IpType::V4(127, 0, 0, 1);
    let v6 = IpType::V6(String::from("2001:0DBB:AC10:FE01"));


}
```

IpType enum from before

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

Pattern Matching with `match`

```
enum IpType {
    V4(u8, u8, u8, u8),
    V6(String),
}

fn main()
{
    let v4 = IpType::V4(127, 0, 0, 1);
    let v6 = IpType::V6(String::from("2001:0DBB:AC10:FE01"));

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


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

Pattern Matching with `match`

```
fn print_ip (ip: IpType)
{
    match ip {
        https://powcoder.com
        Add WeChat powcoder
    }
}
```

IpType arg, no return value

Pattern match input arg


```
enum IpType {
    V4(u8, u8, u8, u8),
    V6(String),
}

fn main()
{
    let v4 = IpType::V4(127, 0, 0, 1);
    let v6 = IpType::V6(String::from("2001:0DBB:AC10:FE01"));

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

Pattern Matching with `match`

Each `match` branch can be an entire block

```
fn print_ip (ip: IpType)
{
    match ip {
        IpType::V4(a, b, c, d) => {
            println!("{}:{}:{}:{}", a, b, c, d);  },
        IpType::V6(a) => {
            println!("{}", a);  },
    }
}
```

V4 case. Notice four values for members of V4

V6 case. Single value for String


```
enum IpType {
    V4(u8, u8, u8, u8),
    V6(String),
}

fn main()
{
    let v4 = IpType::V4(127, 0, 0, 1);
    let v6 = IpType::V6(String::from("2001:0DBB:AC10:FE01"));

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

Command Prompt

```
C:\_RustCode>main
127:0:0:1
2001:0DBB:AC10:FE01

C:\_RustCode>
```

Rules: `match`

Like Haskell, match structures must be exhaustive:

```rust
fn print_ip (ip: IpType)
{
    match ip {
        IpType::V4(a, b, c
            println!("{}:{
        //IpType::V6(a) =>
        //    println!("{}"
    }
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0004]: non-exhaustive patterns: `V6(_)` not cover
ed
  --> main.rs:17:11
   |
17 |     match ip {
   |           ^^ pattern `V6(_)` not covered

error: aborting due to previous error
For more information about this error, try `rustc --expl
ain E0004`.

C:\_RustCode>_
```

Rules: `match`

Also like Haskell, underscores are wild:

```rust
fn print_ip (ip: IpType)
{
    match ip {
        IpType::V4(a, b, c, d) => {
            println!("{}:{}:{}:{}", a,
        _ => {
            println!("Oops!");  },
    }
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
127:0:0:1
Oops!

C:\_RustCode>
```

Nested Clauses

```rust
fn check_red (vals: (u8, u8, u8)) -> String
{
    match vals {
        (a, 0, 0) => {
            if a > 200 { String::from("Bright red") }
            else if a > 100 { String::from("Medium red") }
            else { String::from("Dark red") }
        }
        (_, _, _) => String::from("Not pure red"),
    }
}
```

- 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!

<u>What's wrong with **`NULL`**?</u>

- It's easy to accidentally use NULL as a non-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 **Maybe**, Rust 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

```rust
fn main()
{
    let v1 = Some(5);
    let v2 = Some("Hello");

    let v3: Option<i32> = None;


}
```


Notice:

- When using **None**, we specify a type.
- Rust can't infer a type from **None**.

```rust
fn main()
{
    let v1 = Some(5);
    let v2 = Some("Hello");
    let v3: Option<i32> = None;

    let v4 = v1 + 5;
    let v5 = v3 + 5;
}
```

Command Prompt

```
error[E0369]: binary operation `+` cannot be applied
type `std::option::Option<{integer}>`
  --> main.rs:18:14
   |
18 |     let v4 = v1 + 5;
   |              ^^^^^^
   |
   = note: an implementation of `std::ops::Add` might
 missing for `std::option::Option<{integer}>`
```


- This highlights the difference between using **`NULL`** vs **`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)`

```rust
fn main()
{
    let v1 = Some(5);

    let v1_val =
        match v1 {
            Some(x) => x,
            None => 0,
        };

    println!("{}", v1_val + 5);
}
```


- Very similar to Haskell
- Extract value x from Some(x)
- Decide on some default value for None
- In this case, we say 0. Sensible because we're just 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

Command Prompt

```
C:\_RustCode>main
10
```

Generic Types

Generic Types

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

```
fn max_val (arr: &[i32]) -> i32
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```


- Rust is strongly, statically typed.
- This function will only work for arrays of type i32.
- **What about:** i8, i16, i64, isize, u8, u16, u32, 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!

```rust
fn max_val (arr: &[i32]) -> i32
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```

```rust
fn max_val<T> (arr: &[T]) -> T
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
```



```
C:\_RustCode>rustc main.rs
error[E0369]: binary operation `>` cannot be applied to
type `T`
  --> main.rs:13:12
   |
13 |         if n > largest { largest = n; }
   |            ^^^^^^^^^^^
   |
   = note: `T` might need a bound for `std::cmp::PartialOrd`
```

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

```
Command Prompt
C:\_RustCode>rustc main.rs
error[E0369]: binary operation `>` cannot be applied to type `T`
  --> main.rs:13:12
   |
13 |         if n > largest { largest = n; }
   |            ^^^^^^^^^^^
   |
   = note: `T` might need a bound for `std::cmp::PartialOrd`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0369`.

C:\_RustCode>_
```


- 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

```rust
fn max_val (arr: &[i32]) -> i32
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```

```rust
fn max_val<T> (arr: &[T]) -> T
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```


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

Generic Types: Structs

```rust
struct Point<T> {
    x: T,
    y: T,
}

fn main()
{
    let pt1 = Point { x: 1, y: 2 };
    let pt2 = Point { x: 1.0, y: 2.0 };
    let pt3 = Point { x: "1.0", y: "2.0" };

    println!("{}, {}", pt3.x, pt3.y);
}
```

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

Points declared using int, float, and even Strings

Command Prompt

```
C:\_RustCode>main
1.0, 2.0

C:\_RustCode>
```

However…

```rust
struct Point<T> {
    x: T,
    y: T,
}

fn main()
{
    let pt1 = Point { x: 1, y: 2.0 };

    println!("{}, {}", pt1.x, pt1.y);
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0308]: mismatched types
 --> main.rs:8:32
  |
8 |     let pt1 = Point { x: 1, y: 2.0 };
  |                                ^^^ expected integral variable, found floating-point variable
  |
  = note: expected type `{integer}`
             found type `{float}`

error: aborting due to previous error
```


Type of x and y must match

Generic Types: Struct Methods

```rust
impl<T> Point<T> {
    fn swap_coords(self) -> Point<T>
    {
        Point { x: self.y, y: self.x, }
    }
}

fn main()
{
    let pt1 = Point {x: 1.0, y: 2.0};
    let pt1 = pt1.swap_coords();

    println!("< {}, {} >", pt1.x, pt1.y);
}
```

- Here, we're moving ownership (no &)
- That's OK, we're sending a new Point back to main.
- It doesn't matter what type T is, we're allowed to create a new Point

Command Prompt

```
C:\_RustCode>main
< 2, 1 >

C:\_RustCode>
```

Generic Types: Struct Methods

```rust
struct Point<T> {
    x: T,
    y: T,
}

impl Point<f64> {
    fn vec_len(&self) -> f64 {
        (self.x*self.x + self.y*self.y).sqrt()
    }
}
```

- 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.
- **sqrt()** and * aren't defined for all possible T
- Speaking of **sqrt()**, that's weird…

- We're calling it as a method, rather than a function.
- **sqrt()** is implemented over type f64

```rust
struct Point<T> { x: T, y: T, }

impl Point<f64> {
    fn vec_len(&self) -> f64 {
        (self.x*self.x + self.y*self.y).sqrt()
    }
}

impl<T> Point<T> {
    fn swap_coords(self) -> Point<T> {
        Point { x: self.y, y: self.x, }
    }
}

fn main()
{
    let pt1 = Point {x: 1.0, y: 2.0};
    let pt1 = pt1.swap_coords();

    println!("< {}, {} >", pt1.x, pt1.y);
}
```

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 any type T. Old point is destroyed, new point with swapped 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

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```


- Here's an example with two generic types.
- **`Ok`** can have a different type from **`Err`**
- Simply add more type variables in the definition

- **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

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}

fn main()
{
    let r1 = Result::Ok(5);
    let r2 = Result::Err("Bad!");

}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0282]: type annotations needed
 --> main.rs:8:14
  |
8 |     let r1 = Result::Ok(5);
  |              ^^^^^^^^^^^^^ cannot infer type for `E`
  |
  |          consider giving `r1` a type

error: aborting due to previous error
```


- 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

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}

fn main()
{
    let _r1: Result<i32, String> = Result::Ok(5);
    let _r2: Result<f64, &str> = Result::Err("Bad!");
}
```

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

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

mut

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}

fn main()
{
    let mut _r1: Result<i32, String> = Result::Ok(5);

    _r1 = Result::Err(String::from("Test"));
}
```

- If `_r1` is mutable, it can take the value of `Ok(T)` ***or*** `Err(E)`
- Thus, Rust must know the type of each at compile time.

Traits

Traits

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

Let's revisit our max_val function


```rust
fn max_val (arr: &[i32]) -> i32
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```

```rust
fn max_val<T> (arr: &[T]) -> T
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```

```rust
fn max_val<T> (arr: &[T]) -> T
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```

- 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 with the **`PartialOrd`** trait.
- Just like Haskell's Ord type class

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> main.rs:5:12
  |
5 |         if n > largest { largest = n; }
  |            ^^^^^^^^^^^
  |
  = note: `T` might need a bound for `std::cmp::PartialOrd`
```

```rust
fn max_val<T> (arr: &[T]) -> T
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```

```rust
fn max_val<T: PartialOrd> (arr: &[T]) -> T
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```


- Tell Rust we want max_val to accept any type T that implements the PartialOrd trait.
- We're almost there…

Command Prompt

```
C:\_RustCode>rustc main.rs
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
  |                       help: consider using a reference instea
```

```
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 are stored on the stack
- There is still potential here for T being a type stored 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
```

```rust
fn main()
{
    let s = String::from("weird");

    stringPass(s);

    println!("{}", s);
}

fn stringPass (word: String)
{
    println!("{}", word);
}
```

- Ownership moved from **s** to **word**!
- **s** is now invalid!
- This is very different from any other language we're used to.
- This doesn't happen 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!

```rust
fn max_val<T: PartialOrd> (arr: &[T]) -> T
{
}
```



```rust
fn max_val<T: PartialOrd + Copy> (arr: &[T]) -> T
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}
```


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

```rust
fn max_val<T: PartialOrd + Copy> (arr: &[T]) -> T
{
    let mut largest = arr[0];
    for &n in arr.iter() {
        if n > largest { largest = n; }
    }
    largest
}

fn main ()
{
    let list1 = [1, 9, 2, 5, 8, 6, 4];
    let list2 = [1.7, 9.2, 12.0, 5.0, 0.8];
    let list3 = ['z', 'Q', '?', 'A', 'Z'];

    println!("{}", max_val(&list1));
    println!("{}", max_val(&list2));
    println!("{}", max_val(&list3));
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
9
12
Z

C:\_RustCode>
```

```rust
fn main ()
{
    let list1 = [1, 9, 2, 5, 8, 6, 4];
    let list2 = [1.7, 9.2, 12.0, 5.0, 0.8];
    let list3 = ['z', 'Q', '?', 'A', 'Z'];
    let list4 = ["z", "Q", "?", "A", "Z"];

    println!("{}", max_val(&list1));
    println!("{}", max_val(&list2));
    println!("{}", max_val(&list3));
    println!("{}", max_val(&list4));
}
```

?

Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
9
12
Z
Z

C:\_RustCode>
```


- 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**

```rust
fn main ()
{
    let list1 = [1, 9, 2, 5, 8, 6, 4];
    let list2 = [1.7, 9.2, 12.0, 5.0, 0.8];
    let list3 = ['z', 'Q', '?', 'A', 'Z'];
    let list4 = [String::from("A"), String::from("Q")];

    println!("{}", max_v
    println!("{}", max_v
    println!("{}", max_v
    println!("{}", max_v
}
```


Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0277]: the trait bound std::string::String: std::marker::
Copy` is not satisfied
  --> main.rs:20:20
   |
20 |     println!("{}", max_val(&list4));
   |                    ^^^^^^^ the trait `std::marker::Copy` is
not implemented for `std::string::String`
   |
note: required by `max_val`
  --> main.rs:1:1
   |
1  | fn max_val<T: PartialOrd + Copy> (arr: &[T]) -> T
```

Traits and Methods

```rust
struct Point<T> { x: T, y: T, }

impl<T> Point<T> {
    fn swap_coords(self) -> Point<T> {
        Point { x: self.y, y: self.x }
    }
}

fn main()
{
    let pt1 = Point {x: 1, y: 2};
    println!("< {}, {} >", pt1.x, pt1.y);
    let pt1 = pt1.swap_coords();
    println!("< {}, {} >", pt1.x, pt1.y);
}
```

Clunky. Let's create a method we can call that will print our Points.

```rust
struct Point<T> { x: T, y: T, }

impl<T> Point<T> {
    fn swap_coords(self) -> Point<T> {
        Point { x: self.y, y: self.x, }
    }
}
```

```rust
impl<T> Point<T> {
    fn print(&self) {
        println!("< {}, {} >", self.x, self.y);
    }
}
```

```rust
fn main()
{
    let pt1 = Point {x: 1, y: 2
    pt1.print();
    let pt1 = pt1.swap_coords()
    pt1.print();
}
```

- Makes sense right? Except…
- We're assuming that Rust knows how to print every type T.
- Turns out this isn't the case.

```
Command Prompt
error[E0277]: `T` doesn't implement `std::fmt::Display`
  --> main.rs:13:
   |
13 |         prin

matted with the default formatter; try using `:?` instea
d if you are using a format string
   |
   = help: the trait `std::fmt::Display` is not implemen
ted for `T`
   = help: consider adding a `where T: std::fmt::Display
```

Not to worry! **`Display`** is a trait. We know how to use traits.

```rust
use std::fmt::Display;

struct Point<T> { x: T, y: T, }

impl<T> Point<T> {
    fn swap_coords(self) -> Point<T> {
        Point { x: self.y, y: self.x, }
    }
}

impl<T: Display> Point<T> {
    fn print(&self) {
        println!("< {}, {} >", self.x, self.y);
    }
}

fn main()
{
    let pt1 = Point {x: 1, y: 2};
    pt1.print();
    let pt1 = pt1.swap_coords();
    pt1.print();
}
```

First time seeing:

- Similar to a "using namespace" statement in C++

- Include the Display trait in the `impl` definition.
- Now, the print method will work for any type T that implements **`Display`**

Command Prompt

```
C:\_RustCode>main
< 1, 2 >
< 2, 1 >
```

Implement Traits For Custom Types?

Everything we've seen regarding traits so far has been about ***restricting*** functions or methods to types with specific traits.

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

Custom Trait: `Pt3D`

```
struct Pt3D { x: f64, y: f64, z: f64 }

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

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

- Create a new trait (using **`trait`** keyword) called **PointOps**.
- This trait will contain operations on our Pt3D data type
- Initially, we'll start with a simple method for addition.
- This 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

```
struct Pt3D { x: f64, y: f64, z: f64 }

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

impl PointOps for Pt3D {
    fn plus(&self, pt: &Pt3D) -> Pt3D {
        Pt3D { x: self.x + pt.x,
               y: self.y + pt.y,
               z: self.z + pt.z }
    }
}
```

- We're implementing the PointOps trait for our Pt3D type
- Inside the `impl` we find our method definition(s)
- We create and return a new Pt3D whose elements are the sum of the Pt3D 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

```rust
struct Pt3D { x: f64, y: f64, z: f64 }

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

impl PointOps for Pt3D {
  /*fn plus(&self, pt: &Pt3D) -> Pt3D {
      Pt3D { x: self.x + pt.x,
             y: self.y + pt.y,
             z: self.z + pt.z  }
  }*/
}
```

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

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0046]: not all trait items implemented, missing: `plus`
 --> main.rs:7:1
  |
4 |     fn plus (&self, pt: &Pt3D) -> Pt3D;
  |     ---------------------------------- `plus` from trait
...
7 | impl PointOps for Pt3D {
  | ^^^^^^^^^^^^^^^^^^^^^^ missing `plus` in implementation
```

Custom Trait: Pt3D

```rust
struct Pt3D { x: f64, y: f64, z: f64 }

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

impl PointOps for Pt3D {
    fn plus(&self, pt: &Pt3D) -> Pt3D {
        Pt3D { x: self.x + pt.x,
               y: self.y + pt.y,
               z: self.z + pt.z }
    }
}

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);
}
```


- Let's test our **`plus`** method
- Declare two points, call the plus method on p1, pass in p2 as argument.
- Store the result in a new point, p3.
- Print p3.

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

Custom Trait: Pt3D

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

impl PointOps for Pt3D {
    fn plus(&self, pt: &Pt3D) -> Pt3D {
        Pt3D { x: self.x + pt.x,
               y: self.y + pt.y,
               z: self.z + pt.z }
    }
    fn print(&self) {
        println!("< {}, {}, {} >",
            self.x, self.y, self.z);
    }
}
```

```rust
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);
    p3.print();
}
```


Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
< 5.300000000000001, 3.2, 3.3 >

C:\_RustCode>
```

Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
< 5.300000000000001, 3.2, 3.3 >

C:\_RustCode>
```


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 implement it for another type? How about Pt2D?

Custom Trait: Pt2D

```
struct Pt2D { x: f64, y: f64 }
struct Pt3D { x: f64, y: f64, z: f64 }

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

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


```
impl PointOps for Pt3D {
    fn plus(&self, pt: &Pt3D) -> Pt3D {
        Pt3D { x: self.x + pt.x,
               y: self.y + pt.y,
               z: self.z + pt.z  }
    }
    fn pnint(&self) {
```

Custom Trait: Pt2D

```rust
struct Pt2D { x: f64, y: f64 }
struct Pt3D { x: f64, y: f64, z: f64 }

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

impl PointOps for Pt2D {
    fn plus(&self, pt: &Pt2D) -> Pt2D {
        Pt2D { x: self.x + pt.x, y: self.y + pt.y }
    }
    fn print(&self) {
        println!("< {}, {} >", self.x, self.y);
    }
}

impl PointOps for Pt3D {
    fn plus(&self, pt: &Pt3D) -> Pt3D {
        Pt3D { x: self.x + pt.x,
               y: self.y + pt.y,
               z: self.z + pt.z  }
    }
    fn print(&self) {
```

- This time, **impl** PointOps for Pt2D
- Functionality is similar
- We're dealing with a 2D point instead of 3D.

Custom Trait: Pt2D

```rust
impl PointOps for Pt2D {
    fn plus(&self, pt: &Pt2D) -> Pt2D {
        Pt2D { x: self.x + pt.x, y: self.y + pt.y }
    }
    fn print(&self) {
        println!("< {}, {} >", self.x, self.y);
    }
}

fn main()
{
    let p1 = Pt2D {x: 1.1, y: 2.2};
    let p2 = Pt2D {x: 4.2, y: 1.0};
    let p3 = p1.plus(&p2);
    p3.print();
}
```


Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
< 5.300000000000001, 3.2 >

C:\_RustCode>_
```

Implement *Existing* Trait For Custom Types?

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

We can create ***new*** 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:

```
struct Pt2D { x: f64, y: f64 }

fn main()
{
    let p1 = Pt2D {x: 1.0, y: 2.0};
    let p2 = p1;

    println!("< {}, {} >", p1.x, p1.y);
}
```

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

```rust
struct Pt2D { x: f64, y: f64 }

fn main()
{
    let p1 = Pt2D {x: 1.0, y: 2.0};
    let p2 = p1;

    println!("< {}, {} >", p1.x, p1.y);
}
```

- p1 is ***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.

```rust
struct Pt2D { x: f64, y: f64 }

fn main()
{
    let p1 = Pt2D {x: 1.0, y: 2.0};
    let p2 = p1;

    println!("< {}, {} >", p1.x, p1.y);
}
```

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

Command Prompt

```
error[E0382]: use of moved value: `p1.x`
 --> main.rs:9:28
  |
7 |     let p2 = p1;
  |         -- value moved here
8 |
9 |     println!("< {}, {} >", p1.x, p1.y);
  |                            ^^^^ value used here after move
  |
  = note: move occurs because `p1` has type `Pt2D`, which does not
implement the `Copy` trait
```


- Move occurs because Pt2D doesn't implement Copy.
- Variables 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.


```rust
struct Pt2D { x: f64, y: f64 }

fn main()
{
    let p1 = Pt2D {x: 1.0, y: 2.0};
    let p2 = p1;

    println!("< {}, {} >", p1.x, p1.y);
}
```


- The elements of Pt2D are just f64.
- They certainly implement copy.
- We should be OK

#derive

```rust
#[derive(Copy)]
struct Pt2D { x: f64, y: f64 }

fn main()
{
    let p1 = Pt2D {x: 1.0, y: 2.0};
    let p2 = p1;

    println!("< {}, {} >", p1.x, p1.y);
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0277]: the trait bound `Pt2D: std::clone::Clone` is not satisfied
 --> main.rs:2:10
  |
2 | #[derive(Copy)]
  |          ^^^^ the trait `std::clone::Clone` is not implemented for `Pt2D`

error: aborting due to previous error
```


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

#derive

```rust
#[derive(Copy, Clone)]
struct Pt2D { x: f64, y: f64 }

fn main()
{
    let p1 = Pt2D {x: 1.0, y: 2.0};
    let p2 = p1;

    println!("< {}, {} >", p1.x, p1.y);
    println!("< {}, {} >", p2.x, p2.y);
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
< 1, 2 >
< 1, 2 >
C:\_RustCode>
```

- 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.

```rust
#[derive(Copy, Clone)]
struct Pt2D { x: String, y: String }

fn main()
{
    let p1 = Pt2D {
        x: String::from("1.0"),
        y: String::from("2.0")  };

    let p2 = p1;

    println!("< {}, {} >", p1.x, p1.y);
    println!("< {}, {} >", p2.x, p2.y);
}
```

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

Command Prompt

```
C:\ RustCode>rustc main.rs
error[E0204]: the trait `Copy` may not be implemented for this type
 --> main.rs:2:10
  |
2 | #[derive(Copy, Clone)]
  |          ^^^^
3 | struct Pt2D { x: String, y: String }
  |               --------- this field does not implement `Copy`
```

#derive Display?

```rust
use std::fmt::Display;

#[derive(Display)]
struct Pt2D { x: f64, y: f64 }

fn main()
{
    let p1 = Pt2D { x: 1.0, y: 2.0 };

    println!("{}", p1);
}
```

Command Prompt

```
C:\ RustCode>rustc main.rs
error: cannot find derive macro `Display` in t
his scope
 --> main.rs:3:10
  |
3 | #[derive(Display)]
  |          ^^^^^^^

error: aborting due to previous error
```


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:


```
impl fmt::Display for Pt2D {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result
    {
        write!(f, "< {}, {} >", self.x, self.y)
    }
}
```


Format your output as desired here

```rust
use std::fmt;

struct Pt2D { x: f64, y: f64 }

impl fmt::Display for Pt2D {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result
    {
        write!(f, "< {}, {} >", self.x, self.y)
    }
}

fn main()
{
    let p1 = Pt2D { x: 1.0, y: 2.0 };

    println!("{}", p1);
}
```



```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
< 1, 2 >

C:\_RustCode>_
```

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

Rust traits

- Must implement methods described in trait definition
- Can derive existing traits
- Can implement existing traits

Fantastic Rust Reference:

https://doc.rust-lang.org/book/second-edition/