PLC 2024, Lecture 11, 15 February 2024

Tuples

- Similar to other languages
- Can deconstruct by assigning to a tuple of variables
- Index by position using `t.0`, `t.1`, ...

In [2]:
```
fn main() {
    let tup = (500, 6.4, 1);
    let (x, y, z) = tup;
    println!("The value of y is: {y}");
}
```

In [3]:
```
main()
```

```
The value of y is: 6.4
```

Out[3]: ()

In [4]:
```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;
    let six_point_four = x.1;
    let one = x.2;

    println!("x.0 = {}, x.1 = {}, x.2 = {}", five_hundred, six_point_four,one);
}
```

In [5]:
```
main()
```

```
x.0 = 500, x.1 = 6.4, x.2 = 1
```

Out[5]: ()

Arrays

- Notation, indexing are as usual

In [6]:
```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
    println!("first is {}, second is {}", first, second);
}
```

In [7]:
```
main()
```

```
first is 1, second is 2
```

Out[7]: ()

In [8]:
```
let a = [1,2,3,4,5];
```

```
println!("{}",a.len());
```

```
5
```

- Type of an array includes its length!
  - Functions can only work on arrays of a fixed length
- Trick: Explicitly type with wrong type to reveal type of a value

In [9]:
```
fn lastelem(a:[i32]) -> i32{
    let l = a.len();
    a[l-1]
}
```

```
[E0277] Error: the size for values of type `[i32]` cannot be known at compilation time
   ┌─[command_9:1:1]
   │
 1 │ fn lastelem(a:[i32]) -> i32{
   │             ┬ │
   │             └──── doesn't have a size known at compile-time
   │               │
   │               └─ help: function arguments must have a statically known size, borr
owed types always have a known size: `&`
───┘
```

In [10]:
```
let a:() = [1, 2, 3, 4, 5];
```

```
[E0308] Error: mismatched types
   ┌─[command_10:1:1]
   │
 1 │ let a:() = [1, 2, 3, 4, 5];
   │       ┬   ─────────┬─────── 
   │       └────────────┘ expected due to this
   │                    │
   │                    └─────── expected `()`, found `[{integer}; 5]`
───┘
```

In [11]:
```
fn lastelem(a:[i32;5]) -> i32{
    let l = a.len();
    a[l-1]
}
```

In [12]:
```
let b = [1,2,3,4,5];
let n = lastelem(b);
println!("{},{}",n,b.len());
```

In [13]:
```
let b = [1,2,3,4,5,6];
let n = lastelem(b);
println!("{},{}",n,b.len());
```

```
5,5
```

```
[E0308] Error: mismatched types
    ┌─[command_13:1:1]
    │
 2  │ let n = lastelem(b);
    │         ────┬─── ┬
    │             └────┘ arguments to this function are incorrect
    │                  │
    │                  └─ expected an array with a fixed size of 5 elements, found one
with 6 elements
────┘
```

- Instead of passing an array, pass a reference via a slice

In [14]:
```
fn lastelem(a:&[i32]) -> i32{
    let l = a.len();
    a[l-1]
}
```

In [15]:
```
let b = [1,2,3,4,5,7];
let n = lastelem(&b[..]);
println!("{}",n);
```

```
7
```

In [16]:
```
let b = [1,2,3,4,5,7];
let n = lastelem(&b[1..4]);
println!("{}",n);
```

```
4
```

Structs

- Like a tuple with named components
- Instance variables of a class

- Define a class

In [17]:
```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

- Create an instance

In [18]:
```
fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
}
```

- No notion of `private`
- Can always update a component directly

In [19]:
```
fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

- Can have a `struct` with unnamed components
- A tuple is an unnamed `struct` with unnamed components

In [20]:
```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
    let black_R = black.0;
    let origin_z = origin.2;
    println!("Black Red = {}, Origin Z = {}", black_R, origin_z);
}
```

In [21]:
```
main()
```

```
Black Red = 0, Origin Z = 0
```

Out[21]:
```
()
```

- Can pass structs to functions

In [22]:
```
struct Rectangle {
    width: u32,
    height: u32,
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}
```

In [23]:
```
main()
```

```
The area of the rectangle is 1500 square pixels.
```

Out[23]:
```
()
```

- Attach functions to structs --- **methods**
- First parameter of a method is always `&self` (shades of Python)
  - Type of `&self` is fixed by the fact that `impl` refers to the `struct`

In [24]:
```
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()  // Should be (&rect1).area()
    );
}
```

In [25]:
```
main()
```

```
The area of the rectangle is 1500 square pixels.
```

Out[25]:
```
()
```

- Side note: Rust automatically references and dereferences
  - We wrote `rect.area`, which is a short form for `(&rect).area`
  - Inside the function, `self.width` and `self.length` instead of `(*self).width` and `(*self).length`

- Can define methods with parameters other than `self`

In [26]:
```
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

In [27]:
```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
```

```
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

In [28]:
```
main()
```

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

Out[28]: ()

Generics

- Two functions to find the largest element in an array
- Same code, except for the base type of the array

In [29]:
```
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> &char {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

In [30]:
```
fn main() {
    let number_array = [34, 50, 25, 100, 65];
    let number_list = &number_array[..];

    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);

    let char_array = ['y', 'm', 'a', 'q'];
    let char_list = &char_array[..];
```

```
    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}
```

In [31]:
```
main()
```

```
The largest number is 100
The largest char is y
```

Out[31]:
```
()
```

- Create a generic version of the function using a type variable, like Java
- How to specify that the type `T` supports comparison of values?

In [32]:
```
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
[E0369] Error: binary operation `>` cannot be applied to type `&T`
   ┌─[command_32:1:1]
   │
 1 │ fn largest<T>(list: &[T]) -> &T {
   │              │
   │              └─ help: consider restricting type parameter `T`: `: std::cmp::PartialOrd`
   │
 5 │         if item > largest {
   │            ──┬─ ┬ ──┬───
   │              └──────────── &T
   │                 │   │
   │                 └─────── error: binary operation `>` cannot be applied to type `&T`
   │                     │
   │                     └──── &T
───┘
```

- Generic structs and generic methods

In [33]:
```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

In [34]:
```
struct Point<T> {
    x: T,
```

```
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

- As usual, different type variables denote (possibly) different types

In [35]:
```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

In [36]:
```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Traits

- Specify common properties across types
  - Via required functions, as headers
- Like Haskell type classes, Java interfaces

In [37]:
```
trait Summary {
    fn summarize(&self) -> String; // No body
}
```

- Specify that a type implements a trait and provide an implementation of the required function

In [38]:
```
struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}
```

In [39]:
```
struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }

}
```

In [40]:
```
fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

In [41]:
```
main()
```

```
1 new tweet: horse_ebooks: of course, as you probably already know, people
```

Out[41]:
```
()
```

- Can provide a default implementation in a `trait` definition

In [42]:
```
trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
```

```
    }
}
```

- To use default implementation, implement a trait without providing a function definition

In [43]:
```
struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {}
```

In [44]:
```
fn main(){
    let article = NewsArticle {
            headline: String::from("Penguins win the Stanley Cup Championship!"),
            location: String::from("Pittsburgh, PA, USA"),
            author: String::from("Iceburgh"),
            content: String::from(
                "The Pittsburgh Penguins once again are the best \
                 hockey team in the NHL.",
            ),
        };

    println!("New article available! {}", article.summarize());
}
```

In [45]:
```
main()
```

```
New article available! (Read more...)
```

Out[45]:
```
()
```

- Can qualify a type parameter with a trait to limit its scope

In [46]:
```
fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

- Combine multiple trait constraints using `+` (note that `+` stands for *and*)

In [47]:
```
fn notify<T: Summary + Display>(item: &T) {}
```

```
[E0405] Error: cannot find trait `Display` in this scope
   ┌─[command_47:1:1]
   │
 1 │ fn notify<T: Summary + Display>(item: &T) {}
   │                        ───┬───
   │                           └─── not found in this scope
───┘
```

In [48]:
```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {}
```

```
[E0405] Error: cannot find trait `Display` in this scope
   ┌─[command_48:1:1]
   │
 1 │ fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {}
   │                     ──────┬─────
   │                           └─────── not found in this scope
───┘

[E0404] Error: expected trait, found derive macro `Debug`
   ┌─[command_48:1:1]
   │
 1 │ fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {}
   │                                                 ──┬──
   │                                                   └──── not a trait
───┘
```

- Syntactic sugar for type constraints

In [49]:
```
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
{}
```

```
[E0405] Error: cannot find trait `Display` in this scope
   ┌─[command_49:1:1]
   │
 3 │     T: Display + Clone,
   │        ──┬─────
   │          └─────── not found in this scope
───┘

[E0404] Error: expected trait, found derive macro `Debug`
   ┌─[command_49:1:1]
   │
 4 │     U: Clone + Debug,
   │                ──┬──
   │                  └──── not a trait
───┘
```

- Can use a trait directly as a type

In [50]:
```
fn returns_summarizable() -> impl Summary {
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
}
```

- Now we can fix our original example of a generic function for the largest element in an array

In [51]:
```
fn largest<T:PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];

    for item in list {
```

```
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

In [52]:
```
fn main() {
    let number_array = [34, 50, 25, 100, 65];
    let number_list = &number_array[..];

    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_array = ['y', 'm', 'a', 'q'];
    let char_list = &char_array[..];

    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

In [53]:
```
main()
```

```
The largest number is 100
The largest char is y
```

Out[53]:
```
()
```

Lifetimes

- Recall example last time where Rust caught a dangling reference
- In general, a reference cannot outlive the lifetime of the item it refers to

In [54]:
```
fn main() {
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

```
[E0597] Error: `x` does not live long enough
   ┌─[command_54:1:1]
   │
 5 │         let x = 5;
   │             ┬
   │             └── binding `x` declared here
 6 │         r = &x;
   │             ┬┬
   │             └── borrowed value does not live long enough
 7 │     }
   │     ┬
   │     └── `x` dropped here while still borrowed
   │
 9 │     println!("r: {}", r);
   │                       ┬
   │                       └── borrow later used here
───┘
```

In [55]:
```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}
```

```
[E0597] Error: `x` does not live long enough
   ┌─[command_55:1:1]
   │
 5 │         let x = 5;        // -+-- 'b  |
   │             ┬
   │             └── binding `x` declared here
 6 │         r = &x;           //  |       |
   │             ┬┬
   │             └── borrowed value does not live long enough
 7 │     }                     // -+       |
   │     ┬
   │     └── `x` dropped here while still borrowed
   │
 9 │     println!("r: {}", r); //          |
   │                       ┬
   │                       └── borrow later used here
───┘
```

- In the following example, what does the return value refer to?
- From the code, it either tracks `x` or `y`, but which one?

In [56]:
```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
[E0106] Error: missing lifetime specifier
   ┌─[command_56:1:1]
   │
 1 │ fn longest(x: &str, y: &str) -> &str {
   │                                 ┬
   │                                 └── expected named lifetime parameter
───┘
```

- Can use variables to tag lifetimes
- Like type variables, but preceded by a `'`
- Convention is to use lowercase for lifetime variables, uppercase for type variables
- This does not change the lifetime, merely tells Rust what to check
  - In the example below, the return value must share a lifetime with both `x` and `y`

In [57]:
```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

- Now the following code works
  - `string1.as_str()` is same as `&string1[..]`

In [58]:
```
fn main() {
    let string1 = String::from("long string is long");
    let string2 = String::from("xyz");
    let result = longest(string1.as_str(), string2.as_str());
    println!("The longest string is {}", result);
}
```

In [59]:
```
main()
```

```
The longest string is long string is long
```

Out[59]:
```
()
```

- This also works, though the lifetime of `string2` is less than `string1`
- When the function runs, both arguments are valid

In [60]:
```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(), string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

In [61]:
```
main()
```

```
The longest string is long string is long
```

Out[61]:
```
()
```

- However, if we refer to `result` outside the lifetime of `string2`, the compiler complains

In [62]:
```
fn main() {
    let string1 = String::from("long string is long");
    let result;

    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }

    println!("The longest string is {}", result);
}
```

```
[E0597] Error: `string2` does not live long enough
    ┌─[command_62:1:1]
    │
  6 │         let string2 = String::from("xyz");
    │             ───┬───
    │                └─────── binding `string2` declared here
  7 │         result = longest(string1.as_str(), string2.as_str());
    │                                             ───┬───
    │                                                └─────── borrowed value does not liv
e long enough
  8 │     }
    │     ┬
    │     └── `string2` dropped here while still borrowed
    │
 10 │     println!("The longest string is {}", result);
    │                                          ──┬──
    │                                            └──── borrow later used here
────┘
```

- If we know that only one argument matters, we need not tag the other with a lifetime

In [63]:
```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

- However, no lifetime tagging does not work here either

In [64]:
```
fn longest(x: &str, y: &str) -> &str {
    x
}
```

```
[E0106] Error: missing lifetime specifier
   ┌─[command_64:1:1]
   │
 1 │ fn longest(x: &str, y: &str) -> &str {
   │                                 ┬
   │                                 └── expected named lifetime parameter
───┘
```