PLC 2026, Lecture 10, 12 February 2026

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

In [ ]:
```
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{
    │               ┬┬┬
    │               └──── help: function arguments must have a statically known size, borrowed slices always have a
known size: `&`
    │                │
    │                └──── doesn't have a size known at compile-time
────┘
```

- Trick: Explicitly type with wrong type to reveal type of a value

In [ ]:
```
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());
```

```
5,5
```

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

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

- How do we write a function that operates on variable length arrays?
- 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);
}
```

- `Color` and `Point` are distinct types, though they have the same structure `(i32, i32, i32)`

In [ ]:
```
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;
    let red:Color = origin; // Type mismatch
    println!("Black Red = {}, Origin Z = {}", black_R, origin_z);
}
```

```
[E0308] Error: mismatched types
   ┌─[command_21:1:1]
   │
 9 │     let red:Color = origin; // Type mismatch
   │                ──┬──   ──┬──
   │                  └─────────── expected due to this
   │                         │
   │                         └──── expected `Color`, found `Point`
───┘
```

- The following works without any problem

In [22]:
```
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;
    let red:Color = black; // No problem
    println!("Black Red = {}, Origin Z = {}", black_R, origin_z);
}
```

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

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

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

- Can pass structs to functions

In [24]:
```
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 [ ]:
```
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)
    );
}
```

```
[E0308] Error: mismatched types
    ┌─[command_25:1:1]
    │
 18 │         area(rect1)
    │         ──┬─ ┬┬┬─
    │           └──────── arguments to this function are incorrect
    │              │ │
    │              └─────── help: consider borrowing here: `&`
    │                │
    │                └──── expected `&Rectangle`, found `Rectangle`
────┘
```

In [26]:
```
main()
```

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

Out[26]:
```
()
```

- 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 [27]:
```
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 [28]:
```
main()
```

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

Out[28]:
```
()
```

- 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 [29]:
```
impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

In [30]:
```
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 [31]:
```
main()
```

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

Out[31]: ()

Generics

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

In [32]:
```
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 [33]:
```
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 [34]:
```
main()
```

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

Out[34]:
```
()
```

- 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 [ ]:
```
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_35:1:1]
   │
 1 │ fn largest<T>(list: &[T]) -> &T {
   │                             │
   │                             └─ help: consider restricting type parameter `T` with trait `PartialOrd`: `: std::cmp::PartialOrd`
   │
 5 │         if item > largest {
   │            ──┬─ ┬ ──┬──
   │              └──────┼── &T
   │                 │   │
   │                 └───────── error: binary operation `>` cannot be applied to type `&T`
   │                     │
   │                     └──── &T
───┘
```

- Generic structs and generic methods

In [36]:
```
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 [37]:
```
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());
}
```

In [38]:
```
main()
```

```
p.x = 5
```

Out[38]:
```
()
```

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

In [39]:
```
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 };
}
```

- An example of a generic function with two type variables
- Note that arguments to `mixup()` are `self` and `Point`, not `&self` and `&Point`
- The call `p1.mixup(p2)` transfers ownership of `p1` and `p2` to `mixup()`. Accessing `p1` or `p2` after the call generates an error

In [40]:
```
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);
    // println!("p1.x = {}, p2.y = {}", p1.x, p2.y); // Generates error, because ownership of p1,p2 transferred to
}
```

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

```
p3.x = 5, p3.y = c
```

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

- Naively trying to make `mixup()` take references fails. Need `Copy` trait

In [ ]:
```
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);
    println!("p1.x = {}, p2.y = {}", p1.x, p2.y);
}
```

```
[E0507] Error: cannot move out of `self.x` which is behind a shared reference
   ┌─[command_42:1:1]
   │
 9 │             x: self.x,
   │                ──┬──
   │                  └──── move occurs because `self.x` has type `X1`, which does not implement the `Copy` trait
───┘

[E0507] Error: cannot move out of `other.y` which is behind a shared reference
    ┌─[command_42:1:1]
    │
 10 │             y: other.y,
    │                ───┬───
    │                   └───── move occurs because `other.y` has type `Y2`, which does not implement the `Copy` trai
t
────┘
```

Traits

- Specify common properties across types
  - Via required functions, as headers

- Like Haskell type classes, Java interfaces

In [43]:
```
trait Summary {
    fn summarize(&self) -> String; // No body
}
```

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

In [44]:
```
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 [45]:
```
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 [46]:
```
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 [47]:
```
main()
```

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

Out[47]:
```
()
```

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

In [48]:
```
trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}
```

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

In [49]:
```
struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {}
```

In [50]:
```
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 [51]:
```
main()
```

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

Out[51]:
```
()
```

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

In [52]:
```
fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

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

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

```
[E0405] Error: cannot find trait `Display` in this scope
   ┌─[command_53:1:1]
   │
 1 │ fn notify<T: Summary + Display>(item: &T) {}
   │                        ───┬───
   │                           └─── not found in this scope
   │
   │ Note: help: consider importing this trait: `use std::fmt::Display;
`
───┘
```

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

```
[E0405] Error: cannot find trait `Display` in this scope
   ┌─[command_54:1:1]
   │
 1 │ fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {}
   │                     ───┬───
   │                        └─── not found in this scope
   │
   │ Note: help: consider importing this trait: `use std::fmt::Display;
`
───┘

[E0404] Error: expected trait, found derive macro `Debug`
   ┌─[command_54:1:1]
   │
 1 │ fn some_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 {}
   │                                                 ──┬──
   │                                                   └── not a trait
   │
   │ Note: help: consider importing this trait instead: `use std::fmt::Debug;
`
───┘
```

- Syntactic sugar for type constraints

In [ ]:
```
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_55:1:1]
   │
 3 │     T: Display + Clone,
   │        ───┬───
   │           └─── not found in this scope
   │
   │ Note: help: consider importing this trait: `use std::fmt::Display;
`
───┘
```

```
[E0404] Error: expected trait, found derive macro `Debug`
   ┌─[command_55:1:1]
   │
 4 │     U: Clone + Debug,
   │                ──┬──
   │                  └──── not a trait
   │
   │ Note: help: consider importing this trait instead: `use std::fmt::Debug;
`
   └─
```

- Can use a trait directly as a type

In [56]:
```rust
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 [57]:
```rust
fn largest<T:PartialOrd>(list: &[T]) -> &T {
    let mut largest = &list[0];

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

    largest
}
```

In [58]:
```rust
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 [59]:
```rust
main()
```

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

Out[59]: ()

- And here is the correct version of `mixup()` for points with heterogenous components

In [60]:
```rust
struct Point<X1:Copy, Y1:Copy> {
    x: X1,
    y: Y1,
}

impl<X1:Copy, Y1:Copy> Point<X1, Y1> {
    fn mixup<X2:Copy, Y2:Copy>(&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);
```

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

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

```
p3.x = 5, p3.y = c
p1.x = 5, p2.y = c
```

Out[61]: ()

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

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

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

```
[E0597] Error: `x` does not live long enough
   ┌─[command_62:1:1]
   │
 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 [ ]:
```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;           //  |       |
    }                     // -+       |
                          //          |
    println!("r: {}", r); //          |
}
```

```
[E0597] Error: `x` does not live long enough
   ┌─[command_63:1:1]
   │
 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 [ ]:
```
fn main() {
    let r;                // ---------+-- 'a
                          //          |
    {                     //          |
        let x = 5;        // -+-- 'b  |
        r = &x;                       |
        let y = 7;                    |
        r = &y; //  |       |         |
    }                     // -+       |
                          //          |
    //println!("r: {}", r); //          |
}
```

```
[E0384] Error: cannot assign twice to immutable variable `r`
   ┌─[command_64:1:1]
   │
 2 │     let r;                // ---------+-- 'a
   │         │                            │
   │         └─ help: consider making this binding mutable: `mut `
   │                                      │
 6 │         r = &x;                      │
   │         ──┬──                       │
   │           └─────── first assignment to `r`
   │                                      │
 8 │         r = &y; //  |       |        │
   │         ──┬──                       │
   │           └─────── cannot assign twice to immutable variable
   │
   │ Note: You can change an existing variable to mutable like: `let mut x = x;`
───┘

[unused_variables] Error: variable `r` is assigned to, but never used
   ┌─[command_64:1:1]
   │
 2 │     let r;                // ---------+-- 'a
   │         ┬                            │
   │         └─ warning: variable `r` is assigned to, but never used
───┘

[unused_assignments] Error: value assigned to `r` is never read
   ┌─[command_64:1:1]
   │
 6 │         r = &x;
   │         ──┬──
   │           └─────── warning: value assigned to `r` is never read
───┘

[unused_assignments] Error: value assigned to `r` is never read
   ┌─[command_64:1:1]
   │
 8 │         r = &y; //  |       |
   │         ──┬──
   │           └─────── warning: value assigned to `r` is never read
───┘
```

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

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

```
[E0106] Error: missing lifetime specifier
   ┌─[command_65:1:1]
   │
 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 [66]:
```rust
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 [67]:
```rust
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 [68]:
```
main()
```

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

Out[68]:
```
()
```

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

In [69]:
```
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 [70]:
```
main()
```

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

Out[70]:
```
()
```

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

In [71]:
```
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);
}
```

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

In [72]:
```
fn longest<'a>(x: &'a str, y: &str) -> &'a str {
    x
}
```

- However, no lifetime tagging does not work here either

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

```
[E0106] Error: missing lifetime specifier
   ┌─[command_73:1:1]
   │
 1 │ fn longest(x: &str, y: &str) -> &str {
   │                                 ┬
   │                                 └── expected named lifetime parameter
   │
```