Hack `postgres` Source Code: Vol I

Chapter 1 : C & Rust

1.27 Rust - Lifeimes

Concept called `Lifetimes` is leveraged in rust to avoid `dangling` references (you can think of them as dangling pointers in C).

For example, consider the following block of code -

```
fn main() {

    let emp_id:i32; // lifetime starting point for `emp_id`

    {
        let assign_id:i32 = 101; // lifetime starting point for `assign_id`
        emp_id = &assign_id;
    }// lifetime ending point for `assign_id`

    println!("Employee id: {}", emp_id);

}// lifetime ending point for `emp_id`
```

when rust has to deal with lifetimes of types, it uses `borrow checker` of its compiler.

In the above mentioned code, rust declares a variable `emp_id` without any value. We get an error if we compile our code with this one line. Because, rust doesn't compile the code with uninialized variables and unused variables without some tweaks. Later, rust opens new block (called inner block with two curly brackets - `{` and `}`). Lifetime of any variable that is created in that block restricted to that block only. They don't have any say in outer block which is owned by `main()`. However, variables created in outer block (in our case `main()` 's block) can own inner block too. After opening new block, rust creates a variable `assing_id` with value `101` of type `i32`. This means.. rust created a variable `assign_id` with value `101` in one of the memory locations in the stack. Then, rust gives memory location address (reference) of `assign_id` to `emp_id` (called as borrowing). After this, `emp_id` has capability of holding memory location address where `101` is stored in the stack. But, at the ending curly bracket of inner block, rust destroys the variable `assign_id` from stack. After this, the variable `emp_id` is like holding memory location which contains no value because of cleanup of the variable `assign_id`. This case technically called as "emp_id pointing to dangling reference (destroyed variable memory location)". Borrow checker in rust compiler is able to find this kind cases in program and throw a compilation error by checking the lifetimes.

This borrow checking happens when we deal with functions, structures etc., in rust.

for example...

```
// main-line-0
fn main() {

        // main-line-1
        let x:i32 = 10;
        // main-line-2
        let y:i32 = 20;

        // main-line-3
        let z:&i32 = largest(&x,&y);

        // main-line-4
        println!("Largest is {}", z);
}

// largest-line-0
fn largest(a:&i32, b:&i32) -> &i32 {

    //largest-line-1
    if a>b {
        a
    } else {
        b
    }
}
```

when we try to compile the above code, rust throws an error with some information related to `lifetimes` .

Let's see how rust executes above code -

`main-line-0` - Starts `main` function with its own block/scope using curly brackets.

`main-line-1` - Immutable variable x with value - 10 of type `i32` is targeted to be stored in one of the memory locations of stack in run time.

`main-line-2` - Immutable variable y with value - 20 of type `i32` is targeted to be stored in one of the memory locations of stack in run time.

`main-line-3` - During runtime, `largest` function is invoked by borrowing `x` and `y` references and returns a reference containing largest value and is going to be sotred in the variable `z` of type `i32` .

`largest-line-0` : rust opens largest function with it's own block/scope using curly brackets.

`largest-line-1` : borrow checker in rust compiler follows cerain rules to deal with life times of references in functions. In this case, as there are no rust's life time annotations(we will see later), borrow checker assigns a life time annotations for each input parameter in function signature. Like, `a` may get life time `l1` and `b` may get life time `l2` . Borrow checker calls them as `input lifetimes` as lifetimes are assigned to input parameters of function. Later, it tries to assign lifetime for returing reference based on code logic. Generally, if there is only `input lifetime` then the borrow checker simply assigns that life time to return type and rust calls that life time as `output lifetime` as it is assigned to return types. But, in our case, we have two input lifetimes and code is defined to return any one of them. Borrow checker gets confused while assigning `output lifetime` as there are two possibilities. It can't just flip a coin to select one lifetime. When rust compiler is confused, it throws an error with some information on resolving that confusion. When we execute that above code, it throws an error with information regarding `named life time parameter` .

`main-line-4` : It doesn't execute.

If we can assign same life time to `a` and `b` in function, rust compiler doesn't get confused as there will be only one input life time. After applying same life time to function -

```
// main-line-0
fn main() {

        // main-line-1
        let x:i32 = 10;
        // main-line-2
        let y:i32 = 20;

        // main-line-3
        let z: &i32 = largest(&x,&y);

        // main-line-4
        println!("Largest is {}", z);
}

// 'a is used to denote lifetime and dubbed as lifetime parameter
fn largest <'a> (a:&'a i32, b:&'a i32) -> &'a i32 {

    //largest-line-1
    if a>b {
        a
    } else {
        b
    }
}
```

named life parameters are same as generic type parameters. But, these are used with general name `a` with apostrophe- `'` such as `'a`

In methods of struct, rust applies life time of `&self` to return type if there are multiple input parameters.