Hack `postgres` Source Code: Vol I

Chapter 1 : C & Rust

1.12 Ownership System - Borrowing in Rust. Making C & `postgres` to Sleep for Sometime.

Rust uses ownership concept in its ownership system to prevent problems in memory management for some types such as Strings, Vectors etc.,

When we pass values to functions, rust invokes its ownership mechanism. For example, if we want to print a string and its length at the same time in calling function, we need to move string to length calculating function and return length and string to calling function in which we want to print string and its length.

```rust
fn main() {

    // main-line-1
    let _str: String = String::from("String Object");

    // main-line-2
    let _str_length:usize = calculate_string_length(_str);

    // main-line-3
    println!("{}{}", _str, _length);

}

fn calculate_string_length(_s: String) -> usize {
    let _s_len: usize = _s.len();
    _s_len
}
```

`main-line-1` : Rust creates `_str` pointer in stack memory in RAM. Then, it uses its allocator to find some free memory location in heap memory in RAM to place string data - "String Object". After placing string data into heap memory location, rust retrieves address of that heap memory location and assign it to `_str` pointer in stack memory.

`main-line-2` : Rust transfers `_str` ownership to `_s` . By doing so, it forbids usage of `_str` in main function in further execution as per "ownership" concept. `calculate_string_length` function uses `_s` to find length. At the closing curly bracket of `calculate_string_length` function, rust observes the scope completion of `_s` . So, it clears the heap and stack associated with it. Then, it returns length of `_s` to calling function which is `main`. As length is calculated before the cleanup of `_s`, return value can be transfered to calling function without any issue.

`main-line-3` : Rust throws an error while executing `println!` macro. because, it is aware of `_str_length` in memory. But, it lost `_str` value at the closing bracket of `calculate_string_length` function.

So, to resolve this, when we pass string to called function, we should also return its length and passed string from called function. And also, the calling function should be ready to receive both values at the same time.

```
fn main() {

    // main-line-1
    let _str: String = String::from("String Object");

    // main-line-2 (modified)
    let (_str, _str_length): (String, usize) = calculate_string_length(_str);

    // main-line-3
    println!("{}{}", _str, _length);

}

fn calculate_string_length(_s: String) -> (String, usize) {
    let _s_len: usize = _s.len();
    (_s, _s_len)
}
```

But, the `calculate_string_length` function purpose is to return length only. Rust is forcing us to send string also. We can escape this forcing attitude of rust by using the second concept in "Ownership System" of rust which is "Borrowing".

Instead of transferring the ownership to some other variable or function, rust borrows ownership. When variable - A borrows ownership to variable - B, ownership of variable - A is unchanged and rust doesn't make that variable - A invalid and not forbid its usage further in execution.

Rust uses "references" to implement this borrowing concept.

```
fn main() {

    //main-line-1
    let _str: String = String::from("String Object");

    //main-line-2
    let _str_length: usize = calculate_string_length(&s_str);
    //main-line-3
    println!("{} - {}", _str, _str_length);

}

fn calculate_string_length(_s: &String) -> usize {

    let _s_len: usize = _s.len();

    _s_len

}
```

`main-line-1` : Rust organizes string variable in stack memory and heap memory in RAM.

`main-line-2` : Rust is passing reference of `_str` by passing `&_str` to called function. Called function is also equipped to receive this reference by having `_s: &String` . References hold address of memory locations. In this case, we are passing address of memory location that is held by `_str` in stack memory which leads to string data - "String Object" in heap memory. Function `calculate_string_length` uses this address and goes to heap memory and calculates length of string data. At the end of `calculate_string_length` , there won't be any cleanup of `_s` . Because, `_str` variable's ownership is not transferred to `_s` in `calculate_string_length` function.

`main-line-3` : As `main` has ownership of `_str` , rust can find `_str` . So, it prints string and its length without any issues.

Until we use `mut` in rust, all variable declarations are immutable(As far as I know). We can create multiple immutable references (like `_s: &String`). But, rust allows only one mutable reference at a single point in time.

References have their "life time" between their initialization and their first usage. They don't last until closing curly bracket of a function.

Rust doesn't allow to create mutable references for immutable variables also.

```
fn main() {

    // Creating immutable string
    let _str: String = String::from("String Object");

    // Creating mutable referecne to _str and causes error
   // let _str_mut_ref_1: &mut String = &mut _str;

    // Creating multiple immutable reference to _str
    let _str_ref_1: &String = &_str;
    let _str_ref_2: &String = &_str;
    // first usage of _str_ref_1(_str_ref_1's completion of life time)
    println!("{}", _str_ref_1);

    // Using _str_ref_1 again causes error
    //println!("{}", _str_ref_1);

    // Creating mutable string
    let mut _str_mut: String  = String::from("Mutable String Object");

    //Creating mutable reference
    let _str_mut_ref_1: &mut String = &mut _str_mut;

    //Creating another mutable reference causes error
   // let _str_mut_ref_2: &mut String = &mut _str_mut;

    //End of life time for _str_mut_ref_1
    println!("{}", _str_mut_ref_1);

    //Creating another mutable reference as there is no live mutable reference
    let _str_mut_ref_2:&mut String = &mut _str_mut;

}
```