Hack `postgres` Source Code: Vol I

Chapter 1 : C & Rust

1.15 Pointers in C and References in Rust.

So, C is too powerful language that it can do good as well as bad. There is such provision in C to accomodate unsafe environment in computer.

If we eliminate C's power of doing bad things, we get Rust. Ofcourse, we can make rust to do bad things but it is not as easy as C.

So, it is the concept of `pointers` in C that makes too much powerful. Someone can have access to computer's memory (and more) using C's `pointers`.

We can say `pointers` kind of `references` in rust but not exactly.

`pointers` in C are used to store address of other types. `references` in rust are also used to store address of other types. Eventhouth both are doing the same thing, they have their own purpose.

```c
#include<stdio.h>

int
main () {

    // c-main-line-1
    int number = 10;

    // c-main-line-2
    int *ptr_to_number = &number;
    //Also valid - `int* ptr_to_number = &number;` But, prefere above one.

    // c-main-line-3
    printf("C placed %d in a memory location with address %p", number, ptr_to_number);

    return 0;

}
```

`c-main-line-1` : C stores value 10 with identifier `number` at some memory location in RAM memory. Unlike rust, there is no specific storage mechanism in C to store either in stack memory or heap memory in RAM memory as per `C reference`. I think, it depends on compiler. However, one thing doesn't change. It stores variable values in RAM memory somewhere.

c-main-line-2 : In C, pointer variable names prefixed with * (most of the cases). Pointers are used to store address of other variable and pointers type should match to that other variable type. This brings another notation for pointer that the * can be suffixed to type also. C goes to memory location of number variable and fetches the address of that memory location and assign it to ptr_to_number variable.

c-main-line-3 : pointers use format specifier - %p to print memory address. If we use *ptr_to_number in printf statment, C goes to address of the memory location stored in ptr_to_number and gets value at that memory location. This is called dereferencing in C. As the value that we would get after deferencing is int , we need to use %d as format specifier in printf .

```rust
fn main() {

    // rust-main-line-1
    let number: i32 = 10;

    // rust-main-line-2
    let refernce_to_number: &i32 = &number;

    // rust-main-line-3
    println!("Rust placed {} in memory location at address {:p}", number, refernce_to_

    }
```

rust-main-line-1 : Unlike C, rust uses specific storage mechanism. It stores primitives on stack memory and organizes complex types(Vectors, Strings, et.,) in stack memory and heap memory. So, rust stores variable number value at some memory location in stack memory. *Note: Java also stores primitives on stack and objects created out classes on heap. But, stores those object references on stack.*

rust-main-line-2 : Like C, reference-storing-variable-type in rust is same as variable from which reference is retrieved. Rust goes to memory location of number variable and gets memory location address and assign it to refernce_to_number . Unlike C, there is no * around reference/memory address storing variable. However, we need to use & as prefix to data type of refernce_to_number variable. As per rust's ownership system rules, we are not just storing address of number variable, we are actually borrowing ownership from number variable.

rust-main-line-3 : Rust follows automatic dereferencing . If we use {} in pritln! macro for refernce_to_number , rust goes to memory location address stored in refernce_to_number and gets value and print it. But, if we use {:p} in println! macro, rust just prints reference/memory location address.