600086 Lab Book

Week 3 – Lab C

Date: 21st Feb 2022

Q1. Multiple Rust files

Question a):

Accessing a secondary rust file using the mod statement and use statements

Solution:

Main

```
mod my_second_file;
use my_second_file::run;
fn main() {

    run();
}
```

Second file

```
pub fn run()
{
    let num_of_threads : i32 = 12;
    let mut list_of_threads = vec!();

    println!("Creating Threads");

    for _id in 0..num_of_threads
    {
        list_of_threads.push(std::thread::spawn(move || perform_task(_id) ));
    }

    println!("Joining Threads");
    for thread in list_of_threads
    {
        thread.join().expect("join failed");
    }
    println!("threads joined");
}

fn perform_task(id:i32)
{
    let result:i32 = id * id;
    println!("Thread: {} Result is {}", id,result);
}
```

Test data:

n/a

Sample output:

```
Creating Threads
Thread: 0 Result is 0
Thread: 2 Result is 4
Thread: 1 Result is 1
Thread: 3 Result is 9
Thread: 4 Result is 16
Joining Threads
Thread: 5 Result is 25
Thread: 6 Result is 36
Thread: 7 Result is 49
Thread: 8 Result is 64
Thread: 9 Result is 81
Thread: 10 Result is 100
Thread: 11 Result is 121
threads joined
```

Reflection:

This is useful for organising code into different files based on area of concern.

Metadata:

“Multiple Rust files”,”mod”,” Ownership”,”use”

Further information:

None

Q2. Ownership

Question:

A) The main function creates a Person struct and passes it to the print_person.
Add a second call to print_person within main, to print out the details a second time. Why does this not compile?

B) Rust has a rule that states you cannot have more than one mutable reference to the same object, neither can you have even a single mutable reference to an object that has one or more immutable references. So why does the code you have created, work?

C) Rust has seen that we are trying to use a mutable reference to an object for which there is still an active immutable reference. How can this be solved?

Solution:

A)

```
fn main() {
    let p1 = Person::new("Jane", 30);

    print_person(&p1);
    print_person(&p1)
}

fn print_person(p: &Person) {
    println!("{} is {} years old", p.name, p.age);
}
```

B)

```
fn main() {
    let mut p1 = Person::new("Jane", 30);

    print_person(&p1);
    print_person(&p1);

    increment_age(&mut p1);
    increment_age(&mut p1);
}

fn print_person(p: & Person) {
    println!("{} is {} years old", p.name, p.age);
}

fn increment_age(p: &mut Person){
    p.age = p.age + 1;
}
```

C)

```
fn main() {
    let mut p1 = Person::new("Jane", 30);

    let r1 = &p1;

    print_person(r1);
```

```
    let r3 = &mut p1;

    increment_age(r3);
    increment_age(r3);

    let r2 = &p1;
    print_person(r2);
}

fn print_person(p: & Person) {
    println!("{} is {} years old", p.name, p.age);
}

fn increment_age(p: &mut Person){
    p.age = p.age + 1;
}
```

Test data:

n/a

Sample output:

A)

```
Jane is 30 years old
Jane is 30 years old
```

B)

```
Jane is 30 years old
Jane is 30 years old
```

C)

```
Jane is 30 years old
Jane is 32 years old
```

Reflection:

A) The code will not compile because the print_person(p1) does not use call by value and so passes ownership of the struct to the initial function call meaning no other functions can interact with it because the struct itself is passed into the function meaning that p1 is now unavailable. This can be rectified by modifying print_person(p1) to utilise a reference which will results in the p1 variable being accessible multiple times

B) The using the mutable reference should not allow the code to work however in the given circumstance the references aren't used in an order that would cause an issue and so the Rust compiler allows it to pass

C) Rust will not allow an immutable reference to be called after a mutable one if it was instantiated prior to the mutable reference as in the example code. In order to solve this instantiating, the immutable reference just prior to use allows the code to now run. This is due to the read-only reference(immutable) could become out of date if called once more after a read/Write reference(mutable) has been used.

Metadata:

"Multiple Rust files","mod"," Ownership","use"

Further information:

None

Q3. Classes

Question:

Implementing a class from a second file to allow data to be shared between threads.

Solution:

```rust
mod shared_data;
use shared_data::SharedData;

fn main()
{
    let mut sd = SharedData::new();


    let mut list_of_threads = vec!();

    println!("Creating Threads");

    list_of_threads.push(std::thread::spawn(move || perform_task(& mut sd) ));

    println!("Joining Threads");
    for thread in list_of_threads
    {
        thread.join().expect("join failed");
    }
    println!("threads joined");
}

fn perform_task(shared : &mut SharedData)
{
    shared.update();
    shared.print();
}
```

Test data:

n/a

Sample output:

```
Creating Threads
Joining Threads
SharedData: value = 1
threads joined
```

Reflection:

This is a useful method for sharing data however I could not get it to work with multiple threads at the same time as attempting to reuse the &mut Shared Data reference causes an error on the move closure in the thread::spawn() function. And attempting to create an additional &mut reference caused a different error.

Metadata:

“Multiple Rust files”,”mod”,” Ownership”,”use”,”Classes”

Further information:

Unsure of how to allow multiple writeable references to be used. Also find the &mut to be extremely confusing.