1. The following code sample will fail to compile because the lifetimes of alpha and bravo references cannot be inferred by the compiler. Why can’t they be inferred and what might happen if they were inferred incorrectly? Please give an example.

fn first(alpha: &i32, bravo: &i32) -> &i32 {

if /\*Random number\*/ % 2 == 0 {

alpha

}

bravo

}

The lifetimes cannot be inferred because alpha and bravo might have different lifetimes than each other, and their lifetimes might be different than the returned reference. If the compiler attempted to infer them by simply assuming that all three were the same, it might create a dangling reference by returning a reference with a longer lifetime than the scope of the data it is referencing.

let x = 42;

let mut ref: &i32;

{

let y = 44;

ref = first(x, y);

}

println!(“The Answer: {}”, ref);

1. In the following code snippet and function prototype, does ‘a take the value of the inner or outer scope inside the function call? Why?

fn example<’a>(one: &’a i32, two: &’a i32) -> &’a i32;

let x = 42;

{

let y = 31;

let ref = example(x, y);

}

‘a takes the value of the inner scope. Because both a and b possess the same lifetime of ‘a, ‘a must take the value of the smaller scope to prevent the possibility of a dangling reference if the smaller scoped reference is returned. The lifetime of the outer scope can act as a shorter lifetime if need be, but the converse does not work. This is because anything in the outer scope will always be valid for at least as long as anything in the inner scope.

1. Lifetimes must always be specified in structs, why are they different from functions?

In functions, any reference is guaranteed to be valid for at least the duration of the function itself because the scope of the function is always contained within the scope of a reference. Structures, on the other hand, can be moved and borrowed potentially well outside their original scopes, so they always have the potential to create dangling references unless prevented from doing so.

1. What is the ‘static lifetime? When should and shouldn’t it be used?

The ‘static lifetime is the lifetime of the entire program, so static references never go out of scope. Static references should be used when it is intended that a reference always stay in scope for the entirety of the program. If it is in doubt whether the reference should always be in scope, it should not be a static reference. Static references should be used very sparingly.

1. Why will the following code fail to compile?

struct Example<’a> {

data: f32,

title: String

index: &’a i32

}

impl<’a> Example<’a> {

fn reset(&mut self, alternate: &’a i32) {

index = alternate;

}

}

let index = 0;

let mut test: Example = Example {

data: 1.0,

title: String::from(“Name”),

index

}

for \_ in 0..10 {

let alternate: i32 = some\_function\_that\_returns\_an\_integer ();

test.reset(&alternate);

}

The code fails to compile because the Example instance ‘test’ exists in the outer scope and its reference of lifetime ‘a must be valid for at least as long as it is. Therefore, ‘a takes the lifetime of the outer scope. However, when the loop attempts to reset the index reference to a new reference scoped inside of the loop. This inner scope will cannot act as ‘a because it will not last as long as that outer scope, so the reference to ‘alternate’ isn’t of the correct lifetime to be passed to reset().