Operating System (H)

Southern University of Science and Technology
Mengxuan Wu
12212006

Assignment 1

Mengxuan Wu

The commented lines are the original code.

Test 1

```
fn test1() {
    let s1 = String::new();
    let s2 = s1;

    // answer_here!(println!("s1:{:?}", s1););
    answer_here!(println!("s1:{:?}", s2););

    assert!(s2.is_empty());
}
```

The original code fails to compile because before the `println!` macro, the variable `s1` is moved to `s2`. Thus, `s1` is no longer valid.
The modified code uses `s2`, which is still valid after the move.

Test 2

```
fn test2() {
    let s1 = String::new();
    // answer_here!(let s2 = s1;);
    answer_here!(let s2 = &s1;);

    assert!(s1.is_empty());
    assert!(s2.is_empty());
}
```

The original code fails to compile because before the `assert!` macro, the variable `s1` is moved to `s2`. Thus, `s1` is no longer valid.
The modified code changes the type of `s2` to a reference, which does not take ownership of the value. Thus, `s1` is still valid after the move.

Test 3

```rust
fn test3() {
    let mut value = 0;
    let ref_value = &value;

    value = 1;

    // answer_here!();
    answer_here!(let ref_value = &value;);

    assert_eq!(*ref_value, 1);
}
```

The original code fails to compile because the variable `value` is changed during the lifetime of the immutable reference `ref_value`.
The modified code creates a new immutable reference `ref_value` after the change of `value`. This new ref_value shadows the old one and becomes the one used in the `assert_eq!` macro. During the lifetime of the new `ref_value`, `value` is not changed. Thus, the code compiles.

Test 4

```rust
fn test4() {
    let default_str = String::default();
    let str1 = String::new();
    let result;
    {
        let str2 = String::new();

        // answer_here!();
        answer_here!(let str2 = &default_str;);

        result = longest(&str1, &str2);
    }
    assert!(result.is_empty());
}
```

The original code fails to compile because the function `longest` may return a reference to `str2`, which is not valid after the inner scope. If so, when `result` is used in the `assert!` macro, it will cause a dangling reference.
The modified code makes `str2` a reference to `default_str`, which is still valid after the inner scope. Thus, the code compiles.

Test 5

```rust
fn test5() {
    let ref_cell = RefCell::new(0);
```

```
        let change1 = &ref_cell;
        // answer_here!();
        answer_here!(*change1.borrow_mut() = 1;);

        assert!(ref_cell.into_inner() == 1);
    }
```

The original code can compile but will fail the test. This is because the inner value of a `RefCell` is 0, which fails the assertion.
The modified code borrows the `RefCell` mutably and changes the inner value to 1. Thus, the code passes the test.