University of Toronto, Department of Electrical and Computer Engineering

ECE 1724-F1 — Performant Software Systems with Rust

Quiz 2 — Sample Questions

Wednesday, September 30, 2026

Note: Each question has exactly one correct answer. Unless a complete program is shown, assume the code is inside `fn main()`.

1. What happens when this complete program is compiled and run?

```
fn label(mut text: String) -> String {
    text.push_str("!");
    text
}

fn main() {
    let original = String::from("ready");
    let tagged = label(original.clone());
    println!("{original} {tagged}");
}
```

Ⓐ It prints `ready! ready!`.

Ⓑ It does not compile because `original` is moved into `label`.

Ⓒ It prints `ready ready!`.

Ⓓ It compiles but panics when `label` returns.

2. What happens when this code is compiled and run?

```
let mut text = String::from("go");
let view = &text;
println!("{view}");
let edit = &mut text;
edit.push_str("!");
println!("{edit}");
```

Ⓐ It prints `go`, then `go!`, on separate lines.

Ⓑ It does not compile because `view` remains in scope.

Ⓒ It prints `go`, then `go`, on separate lines.

Ⓓ It does not compile because the binding `edit` needs `mut`.

Quiz 2 — Sample Questions (continued)

3. Why does this code fail to compile?

```
let mut text = String::from("red");
let first = &mut text;
let second = &mut text;
first.push_str("!");
second.push_str("?");
println!("{text}");
```

(A) A `String` can be modified only once.

(B) `push_str` takes ownership of `text`.

(C) A mutable reference cannot be used to call `push_str`.

(D) The second mutable borrow begins while the first is still needed.

4. What happens when this code is compiled and run?

```
let mut text = String::from("red blue");
let word = &text[..3];
text.clear();
println!("{word}");
```

(A) It prints `red` because a slice owns a copy of the bytes.

(B) It does not compile because `clear()` needs a mutable borrow while the slice is still used later.

(C) It prints an empty line because clearing `text` also empties `word`.

(D) It compiles but panics when `word` is printed.

Quiz 2 — Sample Questions (continued)

5. What does this code print?

```
let mut text = String::from("rain");
let ending = String::from("bow");
text.push_str(&ending);
text.push('!');
println!("{text} {ending}");
```

(A) `rainbow! bow`

(B) `rainbow! bow!`

(C) `rain bow`

(D) It does not compile because `ending` is moved by `push_str`.

6. Which statement correctly describes ownership after these lines?

```
let left = String::from("sun");
let right = String::from("rise");
let joined = left + &right;
let caption = format!("{right}: {joined}");
```

(A) All four variables remain usable because both operations borrow all their inputs.

(B) Both `left` and `right` are moved by `+`.

(C) `left` is moved; `right`, `joined`, and `caption` remain usable.

(D) `left` is moved by `+`, and `right` and `joined` are moved by `format!`.

7. The character é occupies two bytes in UTF-8. What happens when this code is compiled and run?

```
let text = String::from("\u{00E9}lan"); // "elan" with an accented e
println!("{}", text.len());
let start = &text[0..1];
println!("{start}");
```

Ⓐ It prints `4`, then the accented character.

Ⓑ It does not compile because Rust forbids all string slicing.

Ⓒ It prints `5`, then the accented character.

Ⓓ It prints `5`, then panics because byte index `1` is inside a character.

8. What does this code print?

```
struct Sensor {
    name: String,
    active: bool,
}

let name = String::from("room");
let mut sensor = Sensor { active: false, name };
sensor.active = true;
println!("{} {}", sensor.name, sensor.active);
```

Ⓐ It does not compile because the fields are initialized out of order.

Ⓑ `room true`

Ⓒ It does not compile because the field definition needs `mut active: bool`.

Ⓓ `room false`

Quiz 2 — Sample Questions (continued)

9. Which single line can be added at the end without causing a compilation error?

```
struct User {
    name: String,
    active: bool,
}

let first = User {
    name: String::from("Mina"),
    active: true,
};
let second = User { active: false, ..first };
```

Ⓐ `println!("{}", first.name);`

Ⓑ `let third = first;`

Ⓒ `println!("{} {}", first.active, second.name);`

Ⓓ `let third = User { ..first };`

10. Why does this complete program fail to compile?

```
struct Position(i32, i32);
struct Velocity(i32, i32);

fn show(position: Position) {
    println!("{} {}", position.0, position.1);
}

fn main() {
    let speed = Velocity(3, 4);
    show(speed);
}
```

Ⓐ `Position` and `Velocity` are distinct types even though their fields have the same types.

Ⓑ Tuple struct fields cannot be accessed using `.0` and `.1`.

Ⓒ Tuple structs can contain only one field.

Ⓓ A tuple struct cannot be passed to a function by value.