University of Toronto, Department of Electrical and Computer Engineering
ECE 1724-F1 — Performant Software Systems with Rust

Quiz 5 — Multiple Choice Questions

Wednesday, November 25, 2026

Name: _____

Student Number: _____

Instructions: Choose exactly one answer per question. Each complete program is independent. Ignore compiler warnings; assume Rust 2024. These teaching examples use standard-library APIs and patterns; no knowledge of library internals is required.

1. What happens when this complete program is compiled and run? trim removes surrounding whitespace.

```
fn label<'a>(text: &'a str, _note: &str) -> &'a str {
    text.trim()
}

fn main() {
    let text = String::from("  ready  ");
    let result;

    {
        let note = String::from("temporary");
        result = label(&text, &note);
    }

    println!("{}", result);
}
```

Ⓐ It prints `ready`.

Ⓑ It fails to compile because note is dropped before result is used.

Ⓒ It prints `temporary`.

Ⓓ It compiles, then panics.

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

```
struct View<'a> {
    text: &'a str,
}

fn main() {
    let view;

    {
        let message = String::from("saved");
        view = View { text: &message };
    }

    println!("{}", view.text);
}
```

Ⓐ It prints `saved`.

Ⓑ It compiles, then panics.

Ⓒ It fails to compile: view would borrow message after message is dropped.

Ⓓ It fails to compile: structs cannot have lifetime parameters.

Quiz 5 — Multiple Choice Questions (continued)

3. What does this program print? `unwrap_or_else` calls its closure only when the `Option` is None.

```
fn main() {
    let mut calls = 0;

    let first = Some(5).unwrap_or_else(|| {
        calls += 1;
        9
    });

    let second = None::<i32>.unwrap_or_else(|| {
        calls += 1;
        9
    });

    println!("{} {} {}", first, second, calls);
}
```

(A) 5 9 2

(B) 5 9 1

(C) 9 9 2

(D) It fails to compile because closures cannot modify captured variables.

4. What does this complete program print?

```
fn main() {
    let values = vec![2, 4, 6];
    let mut iter = values.iter();
    let first = iter.next().unwrap();
    let total: i32 = iter.map(|value| *value * 2).sum();

    println!("{} {}", first, total);
}
```

(A) 2 24

(B) 4 20

(C) It fails to compile because next consumes the entire iterator.

(D) 2 20

5. Why does `Box<Chain>` allow this recursive enum to have a finite size?

```
enum Chain {
    Node(i32, Box<Chain>),
    End,
}

fn main() {
    let _chain = Chain::Node(3, Box::new(Chain::End));
}
```

(A) Box makes every Chain value static.

(B) Box copies all remaining nodes into the current enum value.

(C) The boxed link has a known size; the next Chain value is stored on the heap.

(D) Box removes the need for an End variant by limiting the chain to one node.