


Corso: Programmazione di sistema

Quiz: API Programming - Teoria 3 luglio 2025



PORTALE
ESAMI

	Umberto Fontanazza s323916
Iniziato	3 luglio 2025, 15:26
Stato	Completato
Terminato	3 luglio 2025, 16:06
Tempo impiegato	40 min.
Valutazione	6,4 su un massimo di 9,0 (71%)
Riepilogo del tentativo	<div><div>1</div><div>2</div><div>3</div></div>

Domanda 1

Completo

Punteggio
ottenuto 2,2
su 3,0

1. Si descriva il meccanismo di gestione degli errori fornito dal linguaggio RUST, differenziando tra quelli recuperabili e irrecuperabili
2. Si spieghi il funzionamento dell'operatore ?
3. Si mostri un esempio di applicazione dell'operatore ? scrivendo una funzione (avendo cura di scriverla completamente) che deve cercare una stringa (my_string) e restituire il suo offset in un file, se presente.

Per leggere il file si deve usare la funzione

`fn read_file(path: &str) -> Result<String, std::io::Error>` che legge il file di nome path e restituisce l'intero contenuto del file in una String, se non ci sono errori.

Per la ricerca della stringa si faccia uso del seguente metodo

`fn find(&self, pat: P) -> Option<usize>`

where P: Pattern<'a>

che restituisce l'offset della stringa, se trovata

4. Si descrivano i principali vantaggi ed eventuali svantaggi rispetto ad altri linguaggi di programmazione in termini di organizzazione del codice e prestazioni

1. Rust vuole gestione degli errori esplicita ritornando l'enum `Result<T, E>`. Un `unwrap()` o `expect("")` su `Result` torna `T` o `panica` (arresto completo del processo). Quindi il `panic!()` e' irrecuperabile mentre `Result<>` deve essere gestito dal chiamante.

2. ? propaga il caso di errore al chiamante, sarebbe una sorta di `error.if_err()`
`{return error;}`
 Se il tipo dell'errore e' diverso ma il tratto `From` e' implementato per passare da un errore all'altro e' tutto gestito.

3.

```
fn search_str(needle: &str, path: &str) -> Result<usize, std::io::Error> {
    let owned = read_file(path)?;
    match find(Path::)
    Ok()
}
```

4. Il vantaggio e' che e' chiaro a partire dalla signature di una funzione se puo' generare un errore. Inoltre l'allocazione di un enum come tipo di ritorno e' molto piu' semplice e locale rispetto alla gestione di un exception context.
 Lo svantaggio e' il tempo di sviluppo che si allunga, motivo per il quale sono poi introdotti vari `unwrap()`, `expect()` ecc che consentono di non dover sempre gestire gli errori e che panicano

Commento:

1) che semantica hanno? che fa panic?

```
3)fn find_string(path: &str, s: &str) -> Result<Option<usize>, std::io::Error> {
    let content = fs::read_to_string(path)?;
    Ok(content.find(s))
}
```

vantaggi completi: errori da gestire in modo esplicito, non possono essere nascosti, zero costo, non richiede particolare supporto sullo stack

svantaggi completi: +verboso, difficile incapsulare tutti i tipi possibili di errori

Domanda 2

Completo

Punteggio
ottenuto 1,4
su 3,0

A.

Completa questa funzione in modo che compili correttamente, se ritieni che ci sia un errore di compilazione (descrivendolo), altrimenti dichiara il motivo per cui la funzione è scritta correttamente:

```
fn get_prefix(s: &str, len: usize) -> &str {
    &s[..len]
}
```

B.

Data la struct seguente

```
struct Container<'a> {
    data: &'a str,
}

impl<'a> Container<'a> {
```

```

    fn get(&self) -> &str {
        self.data
    }
}

```

Il metodo `get()` necessita di specificare esplicitamente il tempo di vita del valore ritornato? Motivare la risposta affermativa o negativa.

C.

Il codice seguente compila? Se la risposta è negativa, correggilo e spiega il motivo dell'errore di compilazione.

```

fn make_closure (s: & str) -> impl Fn() -> &str {
    move || s
}

```

D.

Considerare la seguente funzione

```

fn concat_parts<'a>(a: &'a str, b: &'a str) -> &'a str {
    let c = String::from(a) + b;
    &c
}

```

Il codice è correttamente compilato? Si giustifichi la risposta.

Nel caso negativo, si fornisca una soluzione corretta, evitando di introdurre copie inutili delle stringhe, ma eventualmente modificando la signature della funzione.

A. la funzione e' errata perche' misura la lunghezza della slice countando i bytes e non i caratteri che possono essere da 1 a 4 byte.

B. Non e' necessario esplicitare nessun tempo di vita in quanto esiste un solo tempo di vita applicabile di default alla struct, al ref sottostante e di conseguenza a qualunque ref ritornabile.

C. Non compila questo perche' ogni closure e' un tipo a se'. Anche due closures che non possiedono variabili e con lo stesso tipo di parametri e tipo di ritorno sono di tipo diverso e di conseguenza `make_closure<T>` ritornerebbe ogni volta un tipo diverso dopo lo step della monomorfizzazione quindi no.

D. Il codice e' errato. Il tempo di vita di `a` non ci interessa piu' dopo la creazione della `String::from` con possesso, tuttavia non posso ritornare un riferimento alla `String` che viene deallocata a fine scope e che sarebbe quindi un dangling pointer.

Non posso da sue slice ritornare una sola slice perche' una slice e' puntatore + lunghezza e cio' presuppone memoria contigua (cosa non esplicitata dai parametri). Senza duplicare le stringhe bisogna creare un tipo a parte con tempo di vita 'a che implementa l'iteratore su caratteri e tutto il resto.

Altrimenti con copie

```

let mut c = String::from(a);
c.push(b);
return c;

```

Commento:

A) la sintassi è corretta, compila. I lifetime sono correttamente elisi

C) Non compila, perché la lifetime del valore di ritorno non può essere inferita (s deve essere viva quando verrà chiamata la funzione).

Devi esplicitare la lifetime nel tipo restituito:

```
fn make_closure<'a>(s: &'a str) -> impl Fn() -> &'a str {  
    move || s  
}
```

D) ok ma manca la signature corretta modificata della funzione

Domanda 3

Completo

Punteggio
ottenuto 2,8
su 3,0

A.

Si consideri il seguente frammento di programma Rust:

```
1 fn main() {  
2     let mut s = String::from("esame");  
3     let r1 = &s;  
  
4     let r2 = &mut s;  
5     r2.push_str(" superato");  
6     println!("r1: {}", r1);  
7     println!("s: {}", s);  
8 }
```

1. Analizza il codice riga per riga. Quale errore segnalerà il compilatore Rust e perché? Indica la riga precisa in cui l'errore viene rilevato.
2. Riscrivi il codice in modo che sia corretto e produca il seguente output, esclusivamente cambiando l'ordine delle istruzioni:

```
r1: esame  
s: esame superato
```

B.

Considera il seguente frammento di codice Rust.

```
1 fn process_data(data: &mut String) {  
2     data.push_str(" processed");  
3 }  
4 fn analyze_data(data: &String) -> usize {  
5     data.len()  
6 }  
7 fn main() {  
8     let mut s = String::from("original data");  
9     let r1 = &s;  
10    println!("Length: {}", analyze_data(r1));  
11    let r2 = &s;  
12    println!("Content (r2): {}", r2);  
13    process_data(&mut s);  
14    println!("Final string: {}", s);  
15    println!("Length from r1 (after processing): {}", analyze_data(r1));  
16 }
```

Domande:

Quale errore specifico di compilazione (o errori) verrà generato dal borrow checker in questo codice? Qual è la riga o le righe incriminate?

Come potresti modificare il codice in modo minimale per farlo compilare correttamente? Si modifichi il codice mantenendo l'intento di processare e poi analizzare i dati con il seguente output

Length: 13

Content (r2): original data

Final string: original data processed

Length from s (after processing): 23

A. Errore alla riga 4 -> non si può creare un ref mut perché esiste già un ref alla stessa variabile.
Regola: al massimo un ref mut o un numero di ref semplici a piacere (a patto che non ci sia un ref mut).
Anticipando la print il compilatore dropa anticipatamente il ref r1 dopo il suo ultimo uso invece che a fine scope.

```
fn main() {  
    let mut s = String::from("esame");  
    let r1 = &s;  
    println!("r1: {}", r1);  
    let r2 = &mut s;  
    r2.push_str(" superato");  
    println!("s: {}", s);  
}
```


=====

B. Ci sono più errori r1 viene usato dopo essere stato dropato (è necessario il drop prima della creazione del ref mut), inoltre non si può stampare la stringa passando il valore con movimento se si vuole ancora usarla dopo.

```
fn process_data(data: &mut String) {  
    data.push_str(" processed");  
}  
  
fn analyze_data(data: &String) -&gt; usize {  
    data.len()  
}  
  
fn main() {  
    let mut s = String::from("original data");  
    let r1 = &s;  
    println!("Length: {}", analyze_data(r1));  
    let r2 = &s;  
    println!("Content (r2): {}", r2);  
    process_data(&mut s);  
    println!("Final string: {}", &s);  
    println!("Length from s (after processing): {}", analyze_data(&s));  
}
```

Commento:

B) non è questione di drop ma di borrow checker

	Umberto Fontanazza s323916
Iniziato	3 luglio 2025, 13:50
Stato	Completato
Terminato	3 luglio 2025, 14:56
Tempo impiegato	1 ora 5 min.
Valutazione	4,0 su un massimo di 6,0 (67%)
Riepilogo del tentativo	<div><div>1</div></div>

Domanda 1

Completo

Punteggio
ottenuto 4,0
su 6,0

Aggregatore di misure



Un sistema di monitoraggio all'interno di uno stabilimento industriale raccoglie misure di temperatura da più sensori. Le misure vengono raccolte in modo asincrono, sono automaticamente etichettate con l'istante temporale in cui sono comunicate e possono essere inviate da più thread contemporaneamente. Compito del sistema è quello di aggregare le misure ricevute, calcolando la temperatura media e il numero di misurazioni ricevute da ciascun sensore, operando un campionamento ad intervalli regolari indicati dal parametro passato alla funzione di costruzione. In tale periodo, un sensore può inviare più misure, che devono essere tutte considerate nel calcolo della media. Un thread interno alla struttura si occupa di calcolare la media delle temperature per ciascun sensore, aggiornandola secondo il periodo di campionamento indicato. All'atto della distruzione della struttura, il thread interno deve essere terminato in modo sicuro. Per implementare tale sistema, si richiede di realizzare la struct `Aggregator` che offre i seguenti metodi thread-safe:

```
use std::time::Instant;

pub struct Aggregator {
    // campi privati
}

pub struct Average {
    pub sensor_id: usize,
    pub reference_time: Instant,          //indica l'istante temporale in cui è stata calcolata la media
    pub average_temperature: f64,
}

impl Aggregator {
    pub fn new(sample_time_millis: u64) -> Self {
        // implementazione del costruttore
        todo!()
    }

    pub fn add_measure(&self, sensor_id: usize, temperature: f64) {
        // aggiunge una misura di temperatura per il sensore con id `sensor_id`
        // e temperatura `temperature`. Le misure sono automaticamente etichettate
        // con l'istante temporale in cui sono comunicate.
        todo!()
    }

    pub fn get_averages(&self) -> Vec<Average> {
        // restituisce un vettore che riporta la temperatura media di ciascun sensore,
        // calcolata durante l'ultimo periodo di campionamento.
        // Sono presenti solo i sensori che hanno inviato almeno una misura.
        todo!()
    }
}
```

*La risposta a questa domanda è stata inserita tramite un'istanza Crownlabs ed è visibile sul Portale della Didattica sulla sezione **Consegna Elaborati** di questo insegnamento.*

Commento:

Non hai implementato la logica richiesta: il thread deve eseguire un campionamento a intervalli regolari, continuamente campionando le temperature medie ricevute. Nel tuo algoritmo, invece, il campionamento avviene su richiesta, a seguito dell'invocazione del metodo `get_measures()`. Inoltre, non ti liberi delle misure consolidate nel loro valore medio, facendo crescere al passare del tempo la struttura dati senza limiti e inquinando le medie successive che riflettono tutta la storia del sensore e non solo il periodo di riferimento.

I test #2, 3, 4 e 5 falliscono: se avessi provato ad eseguirli (o a leggerne la logica) te ne saresti accorto.

Una possibile soluzione dell'esercizio è:


```

        *sum += m.measure;
        *count += 1;
    })
    .or_insert((m.measure, 1));
});
let new_averages: Vec<Average> = averages.iter()
    .map(|(id, (measure, count))| Average {
        sensor_id: *id,
        reference_time: next_wakeup,
        average_temperature: *measure/ *count as f64,
    })
    .collect();
inner_state = state.0.lock().unwrap();
inner_state.recent_averages = new_averages;
    }
});
Self {
    state: inner_state,
    join_handle: Some(join_handle),
}
}

pub fn add_measure(&self, sensor_id: usize, temperature: f64) {
    let now = Instant::now();
    let mut state = self.state.0.lock().unwrap();
    state.measurements.push(
        Measurement {
            id: sensor_id,
            timestamp: now,
            measure: temperature,
        });
}

pub fn get_averages(&self) -> Vec<Average> {
    // restituisce un vettore che riporta la temperatura media calcolata durante l'ultimo periodo
di campionamento
    // da ciascun sensore. Sono presenti solo i sensori che hanno inviato almeno una misura.
    let state = self.state.0.lock().unwrap();
    state.recent_averages.clone()
}

impl Drop for Aggregator {
    fn drop(&mut self) {
        let mut state = self.state.0.lock().unwrap();
        state.running = false;
        drop(state);
        self.state.1.notify_all();
        let join_handle = self.join_handle.take().unwrap();
        join_handle.join().unwrap();
    }
}
}

```