


# Corso: Programmazione di sistema

## Quiz: API Programming - Teoria 17 giugno 2025



PORTALE  
ESAMI

	Umberto Fontanazza <b>s323916</b>
<b>Iniziato</b>	17 giugno 2025, 17:46
<b>Stato</b>	Completato
<b>Terminato</b>	17 giugno 2025, 18:26
<b>Tempo impiegato</b>	40 min.
<b>Valutazione</b>	<b>6,7</b> su un massimo di 9,0 ( <b>74%</b> )
<b>Riepilogo del tentativo</b>	<div><div>1</div><div>2</div><div>3</div></div>

### Domanda 1

Completo

Punteggio  
ottenuto 1,4  
su 3,0

Si considerino le seguenti strutture dati e rispettive porzioni di codice. Per ciascuna di esse si indichi la dimensione di memoria allocata nello stack e nello heap, ipotizzando un'architettura a 64 bit.

```
let boxed: Box<[u32]> = vec![10, 20, 30, 40, 50].into_boxed_slice();  
let slice_ref: &[u32] = &boxed[1..4];
```

#### Domande:

1. Qual è la dimensione della variabile `boxed` nello stack?
2. Quanta memoria è allocata nello heap dopo l'esecuzione del codice?
3. Qual è lo spazio di memoria occupato da `slice_ref` nello stack?
4. La slice `slice_ref` punta alla stessa memoria heap della `boxed`?

Si considerino le seguenti strutture dati e porzioni di codice. Per ciascuna di esse si indichi la dimensione di memoria allocata nello stack e nello heap, ipotizzando un'architettura a 64 bit.

```
use std::rc::Rc;  
use std::cell::RefCell;  
let shared_vec: Rc<RefCell<Vec<u8>>> = Rc::new(RefCell::new(vec![1, 2, 3]));  
let shared_clone = Rc::clone(&shared_vec);  
shared_clone.borrow_mut().push(4);
```

#### Domande:

1. Quanta memoria è usata nello stack da `shared_vec` e `shared_clone`?
2. Quanto spazio è allocato nello heap in totale (considerando `Rc`, `RefCell`, e `Vec`)?
3. Dopo l'esecuzione del codice, quanti riferimenti forti e deboli esistono?
4. È possibile accedere contemporaneamente in lettura da `shared_vec` e in scrittura da `shared_clone`?

1A: boxed nello stack e' un puntatore (8 byte).

2A: dopo la prima riga nello heap ci sono 20 byte (4 \* 5) di valori + la struttura dello slice (puntatore, lunghezza per altri 16 byte in totale) = 36 byte.

3A: uno slice e' un fat pointer (puntatore 8 byte + lunghezza 8 byte) = 16 byte sullo stack.

4A: Non esattamente, dal box al dato effettivo bisogna attraversare due puntatori mentre la slice va direttamente al dato. I 20 byte di dati esistono tuttavia in memoria una sola volta.

1B: Ogni Rc (i due sono della stessa dimensione sullo stack) e' composto da: puntatore al dato e puntatore alla struttura di controllo con i contatori quindi ogni Rc e' 16 byte.

2B: Dall'interno verso l'esterno 3 byte di dati + struttura del Vec (puntatore, len e size) 24 byte, struttura RefCell (contatore prestiti lettura e contatore in scrittura = 16 byte), L'Rc ha solo la struttura con i contatori sullo heap per 16 byte

3B: 2 forti(gli rc) e

4B: no mai, runtime check

Commento:

A.1(0,4p): fat pointer (la slice non è sized):  $8 + 8 = 16$  byte

A.2(0,4p):  $5 \times 4 = 20$  byte

A.3(0,4p): lunghezza slice + puntatore:  $8 + 8 = 16$  byte

A.4(0,3p): sì, dal secondo elemento

B.1(0,5p): il puntatore, 8 byte

B.2(0,5p):  $[Rc[RefCell[Vec[dati]]]] = 16(Rc) + 8(RefCell) + 24(Vec) + 4(4 \text{ u8}) = 52$

B.3(0,3p): 2 strong 0 weak

B.4(0,2p): no non si può avere borrow\_mut con un borrow

## Domanda 2

Completo

Punteggio  
ottenuto 3,0  
su 3,0

Nel contesto della programmazione generica, si spieghi che cosa si intende per **monomorfizzazione**.

Si scriva una porzione di codice che mostri un esempio pratico in cui la monomorfizzazione viene applicata, spiegando perché tale esempio è rilevante.

Si indichino i principali punti di forza e i punti di debolezza di tale proprietà, facendo un raffronto rispetto agli approcci alla programmazione generica offerti da altri linguaggi di programmazione.

```
Monomorfizzazione vuol dire che per esempio la funzione viene compilata per ogni tipo per cui e' chiamata. L'approccio alternativo per garantire polimorfismo e' quello di usare un puntatore in piu' che punti alla struct dove i dati dell'oggetto sono raccolti e un secondo puntatore alla Vtable con i metodi a esso associati.
```

```
I vantaggi della monomorfizzazione sono la velocita' di esecuzione e lo svantaggio principale e' una maggiore dimensione dei binari compilati.
```

```
Nel caso un tipo generico venga utilizzato per un solo tipo concreto quindi conviene sempre la monomorfizzazione.
```

```
struct Duck();
struct

impl Into<target = String> for Duck {
    ...etc
}

fn greet<T>(label: T) where T: Into<String> {
    let str_label: String = label.into();
    println!("Hello, {str_label}");
}
```

Commento:

ok

## Domanda 3

Completo

Punteggio  
ottenuto 2,3  
su 3,0

Si consideri il seguente programma Rust che usa due thread per gestire una coda condivisa. Le righe sono numerate.

```

1 use std::sync::{Arc, Mutex, Condvar};
2 use std::thread;
3 use std::collections::VecDeque;
4
5 fn main() {
6     let queue = Arc::new((Mutex::new(VecDeque::new()), Condvar::new()));
7
8     let queue_producer = Arc::clone(&queue);
9     let queue_cons = Arc::clone(&queue);
10
11     let producer = thread::spawn(move || {
12         let (lock, cvar) = &*queue_producer;
13         for i in 1..=3 {
14             thread::sleep(std::time::Duration::from_millis(500));
15             let mut queue = lock.lock().unwrap();
16             queue.push_back(i);
17             cvar.notify_one();
18         }
19     });
20
21     let consumer = thread::spawn(move || {
22         let (lock, cvar) = &*queue_cons;
23         loop {
24             let mut queue = lock.lock().unwrap();
25             while queue.is_empty() {
26                 queue = cvar.wait(queue).unwrap();
27             }
28             if let Some(val) = queue.pop_front() {
29                 println!("Consumer 1: got {}", val);
30             } else {
31                 println!("(A)");
32             }
33         }
34     });
35
36     producer.join().unwrap();
37     consumer.join().unwrap();
38 }

```

#### Domande:

1. Descrivere il comportamento complessivo del programma e spiegare cosa fanno i 2 thread.
2. È possibile che venga stampato (A) alla riga 31? Se sì, in quali condizioni?
3. Il programma termina? Se no, spiegare perché e proporre una modifica per permettere una terminazione pulita dei thread consumatori. Nella soluzione proposta occorre indicare le (eventuali) istruzioni da cancellare e le (eventuali) istruzioni da aggiungere, specificando tra quali righe deve avvenire l'inserimento.
4. Ci sono potenziali race condition o deadlock? Giustificare la risposta.

Il primo thread scrive 1, 2, 3 nella coda e termina.

Il secondo thread è sempre in ascolto per elementi aggiunti alla coda, non stampa mai "A" perché se non ci fossero elementi in coda sarebbe bloccato.

Il programma non termina perché il thread consumer non termina mai e il main lo deve aspettare alla join().

Si può usare o un wait\_timeout sulla condvar di modo che dopo un po' che non si ricevono valori il thread termini l'esecuzione (stampando A stavolta perché la coda sarebbe vuota) oppure mettendo un counter che arrivato a 3 fa un break dal loop a seconda del comportamento desiderato.

Se si vuole un comportamento migliore non basato sui timeout si potrebbero avvolgere i valori aggiunti alla coda in un enum dai valori Continue(value), Stop con la variante stop che segna l'intenzione di uscire dal loop.

Commento:

1. come si sincronizzano?

3) ok il wrap, il resto non è generale


4.?

# Corso: Programmazione di sistema

## Quiz: API Programming - Rust 17 giugno 2025



PORTALE  
ESAMI

	Umberto Fontanazza <b>s323916</b>
<b>Iniziato</b>	17 giugno 2025, 16:37
<b>Stato</b>	Completato
<b>Terminato</b>	17 giugno 2025, 17:37
<b>Tempo impiegato</b>	1 ora
<b>Valutazione</b>	<b>5,0</b> su un massimo di 6,0 ( <b>83%</b> )
<b>Riepilogo del tentativo</b>	<div><div>1</div></div>

### Domanda 1

Completo

Punteggio  
ottenuto 5,0  
su 6,0

### TokenManager



Un applicativo software multithread fa accesso ai servizi di un server remoto, attraverso richieste di tipo HTTP.

Tali richieste devono includere un token di sicurezza che identifica l'applicativo stesso e ne autorizza l'accesso.

Per motivi di sicurezza, il token ha una validità limitata nel tempo (qualche minuto) e deve essere rinnovato alla sua scadenza.

Il token viene ottenuto attraverso una funzione (fornita esternamente e conforme al tipo `TokenAcquirer`) che restituisce alternativamente un token e la sua data di scadenza o un messaggio di errore se non è possibile fornirlo.

Poiché la emissione richiede un tempo apprezzabile (da alcune centinaia di millisecondi ad alcuni secondi), si vuole centralizzare la gestione del token, per evitare che più thread ne facciano richiesta in contemporanea. A tale scopo deve essere implementata la `struct TokenManager` che si occupa di gestire il rilascio, il rinnovo e la messa a disposizione del token a chi ne abbia bisogno, secondo la logica di seguito indicata.

La `struct TokenManager` offre i seguenti metodi:

```
type TokenAcquirer = dyn Fn() -> Result<(String, Instant), String> + Sync

pub fn new(acquire_token: Box<TokenAcquirer> ) -> Self
pub fn get_token(&self) -> Result<String, String>s
pub fn try_get_token(&self) -> Option<string>
```

Al proprio interno, la `struct TokenManager` mantiene 3 possibili stati:

1. `Empty` - indica che non è ancora stato richiesto alcun token
2. `Pending` - indica che è in corso una richiesta di acquisizione del token
3. `Valid` - indica che è disponibile un token in corso di validità

Il metodo `new(...)` riceve il puntatore alla funzione in grado di acquisire il token. Essa opera in modalità pigra e si limita a creare un'istanza della struttura con le necessarie informazioni per gestire il suo successivo comportamento.

Il metodo `get_token(...)` implementa il seguente comportamento:

- Se lo stato è `Empty`, passa allo stato `Pending` e invoca la funzione per acquisire il token; se questa ritorna un risultato valido, memorizza il token e la sua scadenza, porta lo stato a `Valid` e restituisce copia del token stesso; se, invece, questa restituisce un errore, pone lo stato a `Empty` e restituisce l'errore ricevuto.
- Se lo stato è `Pending`, attende senza consumare cicli di CPU che questo passi ad un altro valore, dopodiché si comporta di conseguenza.
- Se lo stato è `Valid` e il token non risulta ancora scaduto, ne restituisce una copia; altrimenti pone lo stato ad `Pending` e inizia una richiesta di acquisizione, come indicato sopra.

Il metodo `try_get_token(...)` implementa il seguente comportamento:

- Se lo stato è `Valid` e il token non è scaduto, restituisce una copia del token opportunamente incapsulata in un oggetto di tipo `Option`.

- In tutti gli altri casi restituisce **None**.

Si implementi tale struttura nel linguaggio Rust.

A supporto della validazione del codice realizzato si considerino i seguenti test (due dei quali sono forniti con la relativa implementazione, i restanti sono solo indicati e devono essere opportunamente completati):

```
[test]
fn a_new_manager_contains_no_token() {
    let a: Box<tokenacquirer> = Box::new(|| Err("failure".to_string()));
    let manager = TokenManager::new(a);
    assert!(manager.try_get_token().is_none());
}

[test]
fn a_failing_acquirer_always_returns_an_error() {
    let a: Box<tokenacquirer> = Box::new(|| Err("failure".to_string()));
    let manager = TokenManager::new(a);
    assert_eq!(manager.get_token(), Err("failure".to_string()));
    assert_eq!(manager.get_token(), Err("failure".to_string()));
}

[test]
fn a_successful_acquirer_always_returns_success() {
    //...to be implemented
}

[test]
fn a_slow_acquirer_causes_other_threads_to_wait() {
    //...to be implemented
}
```

*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:

La funzione di acquisizione del token è immutabile e non deve stare all'interno del Mutex: infatti, mentre si acquisisce il token, occorre rilasciare il mutex, altrimenti lo stato Pending non è osservabile.

La notifica va inviata quando lo stato diventa Empty o Valid a seguito dell'invocazione della funzione di acquisizione.

Piuttosto che costruire una funzione ricorsiva, di cui non puoi controllare la profondità, è opportuno trasformarla in una funzione basata su un ciclo, così da poter gestire l'evoluzione dello stato.

Una soluzione basata sul tuo codice 'è:

```

use std::time::{Duration, Instant};
use std::sync::{Mutex, Condvar, Arc};
use std::thread::sleep;

type TokenAcquirer = dyn Fn() -> Result<(String, Instant), String> + Sync+Send;

#[derive(Clone, PartialEq)]
enum State {
    Empty,
    Pending, // with this implementation Pending is basically useless
    Valid((String, Instant))
}

pub struct TokenManager {
    mutex: Mutex<State>,
    condvar: Condvar,
    acquire_token: Box<TokenAcquirer>,
}

impl TokenManager {
    pub fn new(acquire_token: Box<TokenAcquirer>) -> Self {
        Self {
            mutex: Mutex::new(State::Empty),
            condvar: Condvar::new(),
            acquire_token: acquire_token,
        }
    }

    pub fn get_token(&self) -> Result<String, String> {
        let mut lock = self.mutex.lock().unwrap();
        loop {
            match &*lock {
                State::Empty => {
                    *lock = State::Pending;
                    drop(lock);
                    let res = (self.acquire_token)();
                    lock = self.mutex.lock().unwrap();
                    match res {
                        Ok((token, expiration)) => {
                            *lock = State::Valid((token.clone(), expiration));
                            drop(lock);
                            self.condvar.notify_all();
                            return Ok(token)
                        },
                        Err(e) => {
                            *lock = State::Empty;
                            drop(lock);
                            self.condvar.notify_all();
                            return Err(e)
                        },
                    }
                },
                State::Pending => {
                    lock = self.condvar.wait_while(lock, |l| *l == State::Pending).unwrap();
                },
                State::Valid((token, expiration)) => {
                    if *expiration > Instant::now() {
                        return Ok(token.clone())
                    } else {
                        *lock = State::Empty
                    }
                }
            }
        }
    }

    pub fn try_get_token(&self) -> Option<String> {

```

```
        let lock = self.mutex.lock().unwrap();
        if let State::Valid((token, expiration)) = &*lock {
            if *expiration > Instant::now() {
                return Some(token.clone());
            }
        }
        None
    }
}
```