Soluzioni agli esercizi di:

Rust

Guida ***completa*** per sviluppatori e principianti

di Tony Chan

Indice:

Capitolo 1

1) Scrivi un programma che prenda in input due numeri e stampi la loro somma.

```
use std::io;

fn main() {
    println!("Inserisci il primo numero:");
    let mut input1 = String::new();
    io::stdin().read_line(&mut input1).expect("Errore nella lettura dell'input");
    let numero1: i32 = input1.trim().parse().expect("Inserisci un numero valido");

    println!("Inserisci il secondo numero:");
    let mut input2 = String::new();
    io::stdin().read_line(&mut input2).expect("Errore nella lettura dell'input");
    let numero2: i32 = input2.trim().parse().expect("Inserisci un numero valido");

    let somma = numero1 + numero2;
    println!("La somma è: {}", somma);
}
```

2) Scrivi un programma che determini se un numero è pari o dispari.

```
use std::io;

fn main() {
    println!("Inserisci un numero:");
    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");
```

```
    let numero: i32 = input.trim().parse().expect("Inserisci un numero valido");

    if numero % 2 == 0 {
        println!("Il numero è pari");
    } else {
        println!("Il numero è dispari");
    }
}
```

3) Scrivi un programma che stampi i primi 10 numeri naturali.

```
fn main() {
    for i in 1..=10 {
        println!("{}", i);
    }
}
```

4) Scrivi un programma che calcoli la media di 5 numeri inseriti dall'utente.

```
use std::io;

fn main() {
    let mut somma = 0.0;

    for i in 1..=5 {
        println!("Inserisci il numero {}:", i);
        let mut input = String::new();
        io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");
        let numero: f64 = input.trim().parse().expect("Inserisci un numero valido");
        somma += numero;
    }
```

```
    let media = somma / 5.0;
    println!("La media è: {}", media);
}
```

5) Scrivi un programma che prenda una stringa in input e la stampi al contrario.

```
use std::io;

fn main() {
    println!("Inserisci una stringa:");
    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");

    let inversa: String = input.trim().chars().rev().collect();
    println!("La stringa al contrario è: {}", inversa);
}
```

6) Scrivi un programma che calcoli il fattoriale di un numero.

```
use std::io;

fn main() {
    println!("Inserisci un numero:");
    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");
    let numero: u32 = input.trim().parse().expect("Inserisci un numero valido");

    let mut fattoriale = 1;
    for i in 1..=numero {
        fattoriale *= i;
```

```
    }

    println!("Il fattoriale di {} è: {}", numero, fattoriale);
}
```

7) Scrivi un programma che verifichi se una stringa è palindroma.

```
use std::io;

fn main() {
    println!("Inserisci una stringa:");
    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");

    let stringa = input.trim();
    let inversa: String = stringa.chars().rev().collect();

    if stringa == inversa {
        println!("La stringa è palindroma");
    } else {
        println!("La stringa non è palindroma");
    }
}
```

8) Scrivi un programma che stampi tutti i numeri primi da 1 a 100.

```
fn e_primo(numero: u32) -> bool {
    if numero < 2 {
        return false;
    }
    for i in 2..numero {
```

```
        if numero % i == 0 {
            return false;
        }
    }
    true
}

fn main() {
    println!("Numeri primi da 1 a 100:");
    for i in 1..=100 {
        if e_primo(i) {
            println!("{}", i);
        }
    }
}
```

9) Scrivi un programma che calcoli la potenza di un numero base e un esponente forniti dall'utente.

```
use std::io;

fn main() {
    println!("Inserisci la base:");
    let mut base_input = String::new();
    io::stdin().read_line(&mut base_input).expect("Errore nella lettura dell'input");
    let base: i32 = base_input.trim().parse().expect("Inserisci un numero valido");

    println!("Inserisci l'esponente:");
    let mut esponente_input = String::new();
    io::stdin().read_line(&mut esponente_input).expect("Errore nella lettura
dell'input");
    let esponente: i32 = esponente_input.trim().parse().expect("Inserisci un numero
valido");
```

```
    let risultato = i32::pow(base, esponente as u32);
    println!("{} elevato a {} è: {}", base, esponente, risultato);
}
```

10) Scrivi un programma che stampi i primi 10 numeri della serie di Fibonacci.

```
fn main() {
    let mut a = 0;
    let mut b = 1;

    println!("I primi 10 numeri della serie di Fibonacci:");
    for _ in 0..10 {
        println!("{}", a);
        let temp = a;
        a = b;
        b = temp + b;
    }
}
```

11) Scrivi un programma che determini se un anno è bisestile.

```
use std::io;

fn main() {
    println!("Inserisci un anno:");
    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");
    let anno: i32 = input.trim().parse().expect("Inserisci un anno valido");

    if (anno % 4 == 0 && anno % 100 != 0) || anno % 400 == 0 {
```

```
        println!("{} è un anno bisestile", anno);
    } else {
        println!("{} non è un anno bisestile", anno);
    }
}
```

12) Scrivi un programma che calcoli l'area di un cerchio dato il raggio.

```
use std::io;

fn main() {
    const PI: f64 = 3.141592653589793;

    println!("Inserisci il raggio del cerchio:");
    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");
    let raggio: f64 = input.trim().parse().expect("Inserisci un numero valido");

    let area = PI * raggio * raggio;
    println!("L'area del cerchio è: {}", area);
}
```

13) Scrivi un programma che stampi la tabellina del 5 fino a 10.

```
fn main() {
    for i in 1..=10 {
        println!("5 x {} = {}", i, 5 * i);
    }
}
```

14) Scrivi un programma che conti il numero di vocali in una stringa.

```
use std::io;

fn main() {
    println!("Inserisci una stringa:");
    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");

    let stringa = input.trim().to_lowercase();
    let mut conteggio_vocali = 0;

    for carattere in stringa.chars() {
        if "aeiou".contains(carattere) {
            conteggio_vocali += 1;
        }
    }

    println!("Il numero di vocali è: {}", conteggio_vocali);
}
```

15) Scrivi un programma che calcoli la somma di tutti i numeri pari da 1 a 100.

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

    for i in 1..=100 {
        if i % 2 == 0 {
            somma += i;
        }
    }
```

```
    println!("La somma di tutti i numeri pari da 1 a 100 è: {}", somma);
}
```

16) Scrivi un programma che trovi il numero massimo in un array di numeri interi.

```
fn main() {
    let numeri = [3, 7, 2, 9, 5, 11, 6];
    let mut massimo = numeri[0];

    for &numero in &numeri {
        if numero > massimo {
            massimo = numero;
        }
    }

    println!("Il numero massimo è: {}", massimo);
}
```

17) Scrivi un programma che inverta l'ordine degli elementi in un array di interi.

```
fn main() {
    let mut numeri = [1, 2, 3, 4, 5];

    numeri.reverse();

    println!("L'array invertito è: {:?}", numeri);
}
```

18) Scrivi un programma che conti il numero di elementi positivi, negativi e zeri in un array.

```
fn main() {
```

```
    let numeri = [3, -1, 0, 7, -5, 0, 2];
    let mut positivi = 0;
    let mut negativi = 0;
    let mut zeri = 0;

    for &numero in &numeri {
        if numero > 0 {
            positivi += 1;
        } else if numero < 0 {
            negativi += 1;
        } else {
            zeri += 1;
        }
    }

    println!("Positivi: {}, Negativi: {}, Zeri: {}", positivi, negativi, zeri);
}
```

19) Scrivi un programma che calcoli il perimetro e l'area di un rettangolo dati la base e l'altezza.

```
use std::io;

fn main() {
    println!("Inserisci la base del rettangolo:");
    let mut base_input = String::new();
    io::stdin().read_line(&mut base_input).expect("Errore nella lettura dell'input");
    let base: f64 = base_input.trim().parse().expect("Inserisci un numero valido");

    println!("Inserisci l'altezza del rettangolo:");
    let mut altezza_input = String::new();
```

```
    io::stdin().read_line(&mut altezza_input).expect("Errore nella lettura
dell'input");

    let altezza: f64 = altezza_input.trim().parse().expect("Inserisci un numero
valido");


    let perimetro = 2.0 * (base + altezza);

    let area = base * altezza;


    println!("Il perimetro è: {}", perimetro);

    println!("L'area è: {}", area);

}
```

20) Scrivi un programma che trovi il secondo numero più grande in un insieme di numeri interi.

```
fn main() {

    let numeri = [5, 3, 9, 1, 8, 6];


    let mut massimo = i32::MIN;

    let mut secondo_massimo = i32::MIN;


    for &numero in &numeri {

        if numero > massimo {

            secondo_massimo = massimo;

            massimo = numero;

        } else if numero > secondo_massimo && numero != massimo {

            secondo_massimo = numero;

        }

    }


    println!("Il secondo numero più grande è: {}", secondo_massimo);

}
```

21) Qual è la differenza tra una costante e una variabile?

Una costante è un valore che non può mai essere cambiato durante l'esecuzione del programma, mentre una variabile può essere modificata dopo la sua inizializzazione. Le costanti vengono dichiarate con const e devono avere un tipo esplicito. Le variabili sono dichiarate con let, e possono essere mutabili se si aggiunge mut.

Esempio:

```
fn main() {
    const PI: f64 = 3.1415; // Costante, non può essere modificata
    let mut x = 10; // Variabile mutabile
    x = 20; // Modifica del valore di una variabile

    println!("PI: {}, x: {}", PI, x);
}
```

22) Qual è la sintassi per dichiarare una variabile?

Le variabili in Rust si dichiarano con la parola chiave let. Se la variabile deve essere modificabile, si usa anche la parola chiave mut.

Esempio:

```
fn main() {
    let x = 5; // Variabile non mutabile
    let mut y = 10; // Variabile mutabile
    println!("x: {}, y: {}", x, y);
}
```

23) Qual è la sintassi per dichiarare una costante?

Le costanti in Rust si dichiarano con la parola chiave const e devono avere un tipo esplicito.

Esempio:

```
fn main() {
    const MAX_UTENTI: u32 = 100; // Costante con tipo u32
    println!("Il numero massimo di utenti è: {}", MAX_UTENTI);
}
```

24) Qual è il tipo di dato che rappresenta un insieme di valori univoci e immutabili?

Il tipo di dato che rappresenta un insieme di valori univoci e immutabili è l'enumerazione, dichiarata con enum.

Esempio:

```
enum Giorno {
    Lunedi,
    Martedi,
    Mercoledi,
    Giovedi,
    Venerdi,
    Sabato,
    Domenica,
}

fn main() {
    let oggi = Giorno::Lunedi;
```

```
    println!("Oggi è: {:?}", oggi); // Stampa il valore dell'enum
}
```

25) Qual è il tipo di dato che rappresenta un insieme di valori modificabili e indicizzati tramite chiavi?

Il tipo di dato che rappresenta un insieme di valori modificabili e indicizzati tramite chiavi è la HashMap.

Esempio:

```
use std::collections::HashMap;

fn main() {
    let mut mappa = HashMap::new();
    mappa.insert("chiave1", 10);
    mappa.insert("chiave2", 20);

    println!("Valore associato a 'chiave1': {:?}", mappa.get("chiave1"));
}
```

26) Qual è il tipo di dato che rappresenta un valore booleano?

Il tipo di dato che rappresenta un valore booleano è bool, e può assumere i valori true o false.

Esempio:

```
fn main() {
    let vero: bool = true;
    let falso: bool = false;
```

```
    println!("vero: {}, falso: {}", vero, falso);
}
```

27) Quale costrutto viene utilizzato per iterare su una sequenza di valori?

Il costrutto più comune per iterare su una sequenza di valori è il ciclo for.

Esempio:

```
fn main() {
    let numeri = [1, 2, 3, 4, 5];

    for numero in numeri.iter() {
        println!("Numero: {}", numero);
    }
}
```

28) Come si può interrompere un ciclo?

Un ciclo può essere interrotto usando la parola chiave break.

Esempio:

```
fn main() {
    for i in 1..=10 {
        if i == 5 {
            break; // Interrompe il ciclo quando i è uguale a 5
        }
        println!("i: {}", i);
    }
```

```
}
```

29) Qual è l'operatore di assegnazione in Rust?

L'operatore di assegnazione in Rust è =.

Esempio:

```
fn main() {
    let x = 10; // Assegna il valore 10 alla variabile x
    println!("Il valore di x è: {}", x);
}
```

30) Cosa fa la funzione String?

La funzione String::new() crea una nuova stringa vuota, mentre String::from() crea una stringa da un valore statico.

Esempio:

```
fn main() {
    let mut stringa_vuota = String::new(); // Crea una stringa vuota
    let stringa_piena = String::from("Ciao"); // Crea una stringa con contenuto

    stringa_vuota.push_str("Aggiungo del testo");
    println!("Stringa piena: {}, Stringa vuota modificata: {}", stringa_piena,
stringa_vuota);
}
```

Capitolo 2

1) Scrivere un'espressione che calcola il valore del prodotto tra due numeri interi.

```
fn main() {
    let a = 5;
    let b = 3;
    let prodotto = a * b; // Calcola il prodotto
    println!("Il prodotto tra {} e {} è {}", a, b, prodotto);
}
```

2) Scrivere un'espressione che calcola il valore dell'operazione di esponenziazione tra due numeri interi a e b.

```
fn main() {
    let a = 2;
    let b = 3;
    let esponenziale = i32::pow(a, b); // Calcola a^b
    println!("{} elevato alla {} è {}", a, b, esponenziale);
}
```

3) Scrivere un'espressione che concatena due stringhe s1 e s2.

```
fn main() {
    let s1 = String::from("Ciao, ");
    let s2 = String::from("mondo!");
    let concatenazione = s1 + &s2; // Concatena le due stringhe
    println!("{}", concatenazione);
```

```
}
```

4) Scrivere un'espressione che verifica se un numero intero x è pari o dispari.

```
fn main() {
    let x = 4;
    let pari = x % 2 == 0; // Verifica se il numero è pari
    println!("{} è pari? {}", x, pari);
}
```

5) Scrivere un'espressione che calcola il valore dell'area di un cerchio di raggio r.

```
fn main() {
    let r = 3.0;
    let area = std::f64::consts::PI * r * r; // Area del cerchio
    println!("L'area del cerchio di raggio {} è {}", r, area);
}
```

6) Scrivere un'espressione che calcola la somma dei quadrati di due numeri interi a e b.

```
fn main() {
    let a = 3;
    let b = 4;
    let somma_quadrati = a * a + b * b; // Somma dei quadrati
    println!("La somma dei quadrati di {} e {} è {}", a, b, somma_quadrati);
}
```

7) Scrivere un'espressione che calcola il valore della media aritmetica di una lista di numeri n.

```
fn main() {
```

```
    let numeri = vec![1, 2, 3, 4, 5];
    let somma: i32 = numeri.iter().sum();
    let media = somma as f32 / numeri.len() as f32; // Calcola la media
    println!("La media è {}", media);
}
```

8) Scrivere un'espressione che restituisce il valore massimo di un vettore di numeri.

```
fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    let massimo = numeri.iter().max().unwrap(); // Trova il massimo
    println!("Il numero massimo è {}", massimo);
}
```

9) Scrivere un identificatore valido e uno non valido.

Identificatore valido: deve iniziare con una lettera e può contenere lettere, numeri e underscore.

```
let variabile_valida = 10; // Identificatore valido
```

Identificatore non valido: non può iniziare con un numero o contenere caratteri speciali come spazi o simboli.

```
// let 1variabile = 10;  // Identificatore non valido
```

10) Qual è la convenzione di naming per le variabili e per le costanti?

Variabili: in Rust si usa la snake_case per i nomi delle variabili. Esempio: `let numero_totale = 5;`.

Costanti: si usa la SCREAMING_SNAKE_CASE. Esempio: `const NUMERO_MASSIMO: i32 = 100;`.

11) Qual è l'identificatore riservato per indicare una funzione lambda?

In Rust, le funzioni lambda (o chiusure) si definiscono utilizzando la sintassi |parametri| { corpo }.

Esempio:

```
fn main() {
    let quadrato = |x: i32| x * x; // Lambda che calcola il quadrato di un numero
    println!("Il quadrato di 4 è {}", quadrato(4));
}
```

12) Scrivere un programma che richiede all'utente d'inserire un numero intero e stampa il suo quadrato. In caso d'inserimento di un valore non valido, il programma deve gestire l'eccezione e stampare un messaggio di errore.

```
use std::io;

fn main() {
    println!("Inserisci un numero intero:");

    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");

    match input.trim().parse::<i32>() {
        Ok(num) => println!("Il quadrato di {} è {}", num, num * num),
        Err(_) => println!("Errore: non hai inserito un numero valido."),
    }
}
```

13) Scrivere un programma che apre un file di testo e stampa il suo contenuto riga per riga. In caso di errore nell'apertura del file, il programma deve gestire l'eccezione e stampare un messaggio di errore.

```
use std::fs::File;
use std::io::{self, BufRead};
use std::path::Path;

fn main() {
    let path = "file.txt";

    if let Ok(file) = File::open(path) {
        let reader = io::BufReader::new(file);
        for line in reader.lines() {
            match line {
                Ok(content) => println!("{}", content),
                Err(e) => println!("Errore nella lettura della riga: {}", e),
            }
        }
    } else {
        println!("Errore nell'apertura del file {}", path);
    }
}
```

14) Scrivere una funzione che divide due numeri e gestisce l'eccezione di divisione per zero.

```
fn divide(a: i32, b: i32) -> Result<i32, &'static str> {
    if b == 0 {
        Err("Errore: divisione per zero!")
    } else {
        Ok(a / b)
    }
}
```

```
fn main() {
    let a = 10;
    let b = 0;

    match divide(a, b) {
        Ok(result) => println!("Il risultato della divisione è {}", result),
        Err(e) => println!("{}", e),
    }
}
```

15) Scrivere un programma che richiede all'utente d'inserire una lista di numeri interi e ne stampa il valore massimo. In caso d'inserimento di un valore non valido, il programma deve gestire l'eccezione e stampare un messaggio di errore.

```
use std::io;

fn main() {
    println!("Inserisci una lista di numeri interi separati da spazi:");

    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");

    let numeri: Result<Vec<i32>, _> = input.trim().split_whitespace().map(|s|
s.parse()).collect();

    match numeri {
        Ok(nums) => {
            if let Some(max) = nums.iter().max() {
                println!("Il valore massimo è {}", max);
            } else {
                println!("La lista è vuota.");
            }
```

```
        }
        Err(_) => println!("Errore: hai inserito un valore non valido."),
    }
}
```

16) Scrivere una funzione che calcoli il reciproco di un numero e che gestisca l'eccezione di divisione per zero.

```
fn reciproco(x: f64) -> Result<f64, &'static str> {
    if x == 0.0 {
        Err("Errore: divisione per zero!")
    } else {
        Ok(1.0 / x)
    }
}

fn main() {
    let numero = 0.0;

    match reciproco(numero) {
        Ok(r) => println!("Il reciproco di {} è {}", numero, r),
        Err(e) => println!("{}", e),
    }
}
```

17) Scrivere un programma che crea un nuovo file di testo e che scriva una serie di righe in esso.

```
use std::fs::File;
use std::io::{self, Write};

fn main() {
    let path = "nuovo_file.txt";
```

```
    let mut file = File::create(path).expect("Errore nella creazione del file");

    let righe = vec!["Prima riga", "Seconda riga", "Terza riga"];

    for riga in righe {
        writeln!(file, "{}", riga).expect("Errore nella scrittura del file");
    }

    println!("File creato e righe scritte con successo.");
}
```

18) Scrivere un programma che apre un file di testo esistente, legga il suo contenuto e ne stampi ogni riga.

```
use std::fs::File;
use std::io::{self, BufRead};

fn main() {
    let path = "file.txt";

    if let Ok(file) = File::open(path) {
        let reader = io::BufReader::new(file);

        for line in reader.lines() {
            match line {
                Ok(content) => println!("{}", content),
                Err(e) => println!("Errore nella lettura della riga: {}", e),
            }
        }
    } else {
        println!("Errore nell'apertura del file {}", path);
```

```
    }
}
```

19) Scrivere un programma che copia il contenuto di un file di testo in un altro file.

```
use std::fs::{self, File};
use std::io::Write;

fn main() {
    let file_origine = "file_origine.txt";
    let file_destinazione = "file_copia.txt";

    match fs::read_to_string(file_origine) {
        Ok(contenuto) => {
            let mut file = File::create(file_destinazione).expect("Errore nella
creazione del file di destinazione");
            file.write_all(contenuto.as_bytes()).expect("Errore nella scrittura del
file di destinazione");
            println!("File copiato con successo.");
        }
        Err(e) => println!("Errore nell'apertura del file di origine: {}", e),
    }
}
```

20) Scrivere un programma che chiede all'utente d'inserire la propria data di nascita (nel formato GG-MM-ANNO) e calcoli l'età in giorni, mesi e anni. Il programma deve anche calcolare il giorno della settimana in cui l'utente è nato.

```
use chrono::{NaiveDate, Datelike, Local};

fn main() {
    println!("Inserisci la tua data di nascita (formato GG-MM-ANNO):");
```

```
    let mut input = String::new();
    std::io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");

    let data_nascita = NaiveDate::parse_from_str(input.trim(), "%d-%m-%Y");

    match data_nascita {
        Ok(data) => {
            let oggi = Local::today().naive_local();
            let durata = oggi.signed_duration_since(data);

            let anni = durata.num_days() / 365;
            let mesi = (durata.num_days() % 365) / 30;
            let giorni = durata.num_days() % 30;

            println!("Hai {} anni, {} mesi e {} giorni.", anni, mesi, giorni);
            println!("Sei nato di {}", data.weekday());
        }
        Err(_) => println!("Errore: formato data non valido."),
    }
}
```

21) Qual è la funzione utilizzata per aprire un file?

In Rust, la funzione utilizzata per aprire un file è File::open(). Essa restituisce un Result che rappresenta il file se l'operazione ha successo, o un errore se fallisce.

Esempio:

```
use std::fs::File;
```

```
fn main() {
    match File::open("file.txt") {
        Ok(_) => println!("File aperto con successo!"),
        Err(e) => println!("Errore nell'apertura del file: {}", e),
    }
}
```

22) Qual è il metodo utilizzato per creare una nuova directory?

Per creare una nuova directory in Rust, si utilizza il metodo fs::create_dir(), che prende in input il percorso della nuova cartella.

Esempio:

```
use std::fs;

fn main() {
    match fs::create_dir("nuova_cartella") {
        Ok(_) => println!("Cartella creata con successo!"),
        Err(e) => println!("Errore nella creazione della cartella: {}", e),
    }
}
```

23) Come si legge un file di testo riga per riga?

Per leggere un file riga per riga, si usa BufReader e si itera sulle righe utilizzando .lines().

Esempio:

```
use std::fs::File;
use std::io::{self, BufRead};

fn main() {
    if let Ok(file) = File::open("file.txt") {
        let reader = io::BufReader::new(file);
        for line in reader.lines() {
            match line {
                Ok(content) => println!("{}", content),
                Err(e) => println!("Errore nella lettura della riga: {}", e),
            }
        }
    } else {
        println!("Errore nell'apertura del file.");
    }
}
```

24) Qual è il metodo utilizzato per ottenere il percorso assoluto di un file?

Il metodo utilizzato per ottenere il percorso assoluto di un file è fs::canonicalize(). Questo metodo restituisce un Result con il percorso assoluto del file.

Esempio:

```
use std::fs;

fn main() {
    match fs::canonicalize("file.txt") {
        Ok(percorso) => println!("Il percorso assoluto è: {:?}", percorso),
        Err(e) => println!("Errore nel trovare il percorso assoluto: {}", e),
```

```
    }
}
```

25) Qual è il metodo utilizzato per scrivere su un file di testo?

Per scrivere su un file di testo in Rust si utilizza il metodo write_all() della struttura File dopo aver creato o aperto un file con File::create() o File::open().

Esempio:

```
use std::fs::File;
use std::io::Write;

fn main() {
    let mut file = File::create("testo.txt").expect("Errore nella creazione del file");
    file.write_all(b"Questa è una nuova riga di testo.").expect("Errore nella scrittura del file");
    println!("Testo scritto con successo!");
}
```

26) Come si calcola la differenza tra due date utilizzando chrono?

Per calcolare la differenza tra due date in Rust, si utilizza la libreria chrono. Si crea una data con NaiveDate e si utilizza il metodo signed_duration_since() per ottenere la differenza.

Esempio:

```
use chrono::NaiveDate;

fn main() {
```

```
    let data1 = NaiveDate::from_ymd(2020, 5, 15);
    let data2 = NaiveDate::from_ymd(2023, 9, 25);
    let differenza = data2.signed_duration_since(data1);

    println!("La differenza è di {} giorni.", differenza.num_days());
}
```

27) Qual è il metodo utilizzato per ottenere il timestamp corrente?

Per ottenere il timestamp corrente in Rust, si utilizza la funzione Local::now() della libreria chrono, che restituisce la data e l'ora corrente.

Esempio:

```
use chrono::Local;

fn main() {
    let timestamp = Local::now();
    println!("Il timestamp corrente è: {}", timestamp);
}
```

28) Scrivere un listato che richieda all'utente d'inserire un numero intero, e di stampare su schermo "Il numero inserito è pari" se il numero è pari, "Il numero inserito è dispari" se il numero è dispari.

```
use std::io;

fn main() {
    println!("Inserisci un numero intero:");

    let mut input = String::new();
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");
```

```
    match input.trim().parse::<i32>() {
        Ok(numero) => {
            if numero % 2 == 0 {
                println!("Il numero inserito è pari.");
            } else {
                println!("Il numero inserito è dispari.");
            }
        }
        Err(_) => println!("Errore: hai inserito un valore non valido."),
    }
}
```

Capitolo 3

1) Scrivere una funzione che prenda come input il raggio di un cerchio e restituisca l'area del cerchio. Utilizzare il valore di π = 3.14.

```
fn area_cerchio(raggio: f64) -> f64 {
    let pi = 3.14;
    pi * raggio * raggio
}

fn main() {
    let raggio = 5.0;
    println!("L'area del cerchio è: {}", area_cerchio(raggio));
}
```

2) Scrivere una funzione che prenda come input una lista di numeri e restituisca la somma dei suoi elementi.

```
fn somma_lista(numeri: &[i32]) -> i32 {
    numeri.iter().sum()
}

fn main() {
    let lista = vec![1, 2, 3, 4, 5];
    println!("La somma degli elementi è: {}", somma_lista(&lista));
}
```

3) Scrivere una funzione che prenda due argomenti e restituisca la loro somma.

```
fn somma(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    let risultato = somma(10, 20);
    println!("La somma è: {}", risultato);
}
```

4) Definire una funzione che prenda un numero come parametro e restituisca il suo quadrato.

```
fn quadrato(num: i32) -> i32 {
    num * num
}

fn main() {
    let numero = 4;
    println!("Il quadrato di {} è: {}", numero, quadrato(numero));
}
```

5) Scrivere una funzione che prenda in input una stringa e ne restituisca la lunghezza.

```
fn lunghezza_stringa(s: &str) -> usize {
    s.len()
}

fn main() {
    let stringa = "ciao mondo";
    println!("La lunghezza della stringa è: {}", lunghezza_stringa(stringa));
```

```
}
```

6) Scrivere una funzione che prenda in input una lista di parole e ne restituisca una lista di queste in ordine alfabetico.

```
fn ordina_lista_parole(parole: &mut Vec<String>) {
    parole.sort();
}

fn main() {
    let mut lista_parole = vec![
        String::from("banana"),
        String::from("mela"),
        String::from("arancia"),
    ];
    ordina_lista_parole(&mut lista_parole);
    println!("Lista ordinata: {:?}", lista_parole);
}
```

7) Scrivere una funzione che prenda in input una lista di numeri e ne restituisca una di numeri pari.

```
fn numeri_pari(numeri: &[i32]) -> Vec<i32> {
    numeri.iter().filter(|&x| x % 2 == 0).cloned().collect()
}

fn main() {
    let lista = vec![1, 2, 3, 4, 5, 6];
    println!("Numeri pari: {:?}", numeri_pari(&lista));
}
```

8) Scrivere una chiusura che prenda in input un intero e ne restituisca il suo doppio.

```
fn main() {
    let raddoppia = |x: i32| x * 2;
    let numero = 5;
    println!("Il doppio di {} è: {}", numero, raddoppia(numero));
}
```

9) Scrivere una chiusura che prenda in input una stringa e restituisca la stessa in maiuscolo.

```
fn main() {
    let maiuscolo = |s: &str| s.to_uppercase();
    let stringa = "ciao";
    println!("Stringa in maiuscolo: {}", maiuscolo(stringa));
}
```

10) Scrivere una chiusura che prenda in input due numeri e restituisca il loro prodotto.

```
fn main() {
    let prodotto = |a: i32, b: i32| a * b;
    let a = 3;
    let b = 4;
    println!("Il prodotto di {} e {} è: {}", a, b, prodotto(a, b));
}
```

11) Scrivere una funzione che calcoli il fattoriale di un numero intero e applichi un’altra funzione per controllare che il numero in input sia maggiore o uguale a zero.

In Rust non esistono i "decoratori" come in altri linguaggi (Python, ad esempio), ma possiamo implementare un controllo all'interno della funzione stessa per garantire che il numero sia maggiore o uguale a zero.

```
fn fattoriale(n: i32) -> i32 {
    if n < 0 {
        panic!("Il numero deve essere maggiore o uguale a zero.");
    }
    (1..=n).product()
}

fn main() {
    let numero = 5;
    println!("Il fattoriale di {} è: {}", numero, fattoriale(numero));
}
```

12) Scrivere una funzione che restituisca la somma di due numeri e li converta in float.

```
fn somma_float(a: i32, b: i32) -> f64 {
    (a as f64) + (b as f64)
}

fn main() {
    let a = 10;
    let b = 20;
    println!("La somma di {} e {} in float è: {}", a, b, somma_float(a, b));
}
```

13) Scrivere una funzione che restituisca una stringa applicandogli un prefisso.

```
fn aggiungi_prefisso(s: &str, prefisso: &str) -> String {
    format!("{}{}", prefisso, s)
}

fn main() {
```

```
    let parola = "mondo";
    let prefisso = "ciao, ";
    println!("{}", aggiungi_prefisso(parola, prefisso));
}
```

14) Scrivere una funzione che applichi un callback ad ogni elemento di una lista.

```
fn applica_callback<T, F>(lista: &[T], callback: F)
where
    F: Fn(&T),
{
    for elemento in lista {
        callback(elemento);
    }
}

fn main() {
    let numeri = vec![1, 2, 3, 4];
    applica_callback(&numeri, |x| println!("Elemento: {}", x));
}
```

15) Scrivere una funzione che sommi gli elementi di una lista e applichi una funzione di callback al risultato.

```
fn somma_e_callback<F>(numeri: &[i32], callback: F)
where
    F: Fn(i32),
{
    let somma: i32 = numeri.iter().sum();
    callback(somma);
}
```

```
fn main() {
    let numeri = vec![1, 2, 3, 4];
    somma_e_callback(&numeri, |somma| println!("La somma è: {}", somma));
}
```

16) Scrivere una funzione che applichi un callback ad ogni elemento di una lista e restituisca solo gli elementi per cui il callback restituisce True.

```
fn filtra<T, F>(lista: &[T], callback: F) -> Vec<&T>
where
    F: Fn(&T) -> bool,
{
    lista.iter().filter(|&x| callback(x)).collect()
}

fn main() {
    let numeri = vec![1, 2, 3, 4, 5, 6];
    let numeri_pari = filtra(&numeri, |&x| x % 2 == 0);
    println!("Numeri pari: {:?}", numeri_pari);
}
```

17) Scrivere un programma che effettui richieste HTTP a più URL in contemporanea.

Per effettuare richieste HTTP in parallelo, possiamo usare la libreria reqwest insieme a tokio per la concorrenza. Ecco un esempio:

```
# Aggiungi queste dipendenze al tuo Cargo.toml
[dependencies]
reqwest = { version = "0.11", features = ["json"] }
tokio = { version = "1", features = ["full"] }
```

Codice:

```
use reqwest::get;
use tokio::task;

#[tokio::main]
async fn main() {
    let urls = vec![
        "https://www.rust-lang.org",
        "https://www.google.com",
        "https://www.github.com",
    ];

    let handles: Vec<_> = urls.into_iter()
        .map(|url| task::spawn(async move {
            match get(url).await {
                Ok(resp) => println!("Risposta da {}: {}", url, resp.status()),
                Err(e) => println!("Errore nella richiesta a {}: {}", url, e),
            }
        }))
        .collect();

    for handle in handles {
        handle.await.unwrap();
    }
}
```

18) Scrivere un programma che effettui la lettura di file in parallelo.

```
use tokio::fs::File;
use tokio::io::{self, AsyncBufReadExt, BufReader};
```

```
#[tokio::main]
async fn main() -> io::Result<()> {
    let file_paths = vec!["file1.txt", "file2.txt", "file3.txt"];

    let handles: Vec<_> = file_paths.into_iter()
        .map(|path| tokio::spawn(async move {
            let file = File::open(path).await.expect("Errore nell'apertura del
file");
            let mut reader = BufReader::new(file).lines();
            while let Some(line) = reader.next_line().await.expect("Errore nella
lettura") {
                println!("{}", line);
            }
        }))
        .collect();

    for handle in handles {
        handle.await.expect("Errore nell'esecuzione della lettura");
    }

    Ok(())
}
```

19) Scrivere un programma che effettui una richiesta HTTP e che attenda un tempo massimo di cinque secondi per ricevere la risposta.

```
use reqwest::get;
use tokio::time::{timeout, Duration};

#[tokio::main]
async fn main() {
```

```
    let url = "https://www.rust-lang.org";
    let durata = Duration::from_secs(5);

    match timeout(durata, get(url)).await {
        Ok(Ok(resp)) => println!("Risposta ricevuta: {}", resp.status()),
        Ok(Err(e)) => println!("Errore nella richiesta: {}", e),
        Err(_) => println!("Timeout superato!"),
    }
}
```

20) Scrivere un programma che effettui l'elaborazione di una lista di elementi in parallelo.

Per elaborare una lista in parallelo, possiamo usare il crate rayon, che facilita l'elaborazione parallela.

```
# Aggiungi questa dipendenza al tuo Cargo.toml
[dependencies]
rayon = "1.5"
```

Codice:

```
use rayon::prelude::*;

fn main() {
    let numeri = vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    // Elaborazione parallela per raddoppiare ogni numero
    let risultati: Vec<i32> = numeri.par_iter().map(|&x| x * 2).collect();

    println!("Risultati: {:?}", risultati);
}
```

Capitolo 4

1) Definire una struct Libro con titolo, autore e pagine, e un trait Leggibile con un metodo leggi che descrive il libro.

```
trait Leggibile {
    fn leggi(&self) -> String;
}

struct Libro {
    titolo: String,
    autore: String,
    pagine: u32,
}

impl Leggibile for Libro {
    fn leggi(&self) -> String {
        format!("Leggendo {} di {} ({} pagine).", self.titolo, self.autore, self.pagine)
    }
}

fn main() {
    let libro = Libro {
        titolo: String::from("Il Signore degli Anelli"),
        autore: String::from("J.R.R. Tolkien"),
        pagine: 1200,
    };
    println!("{}", libro.leggi());
}
```

2) Definire una struct Veicolo e un trait Motorizzato con i metodi accendi e spegni che funzionano diversamente a seconda del tipo di veicolo (Auto o Moto).

```
trait Motorizzato {
    fn accendi(&self);
    fn spegni(&self);
}

struct Auto {
    modello: String,
}

struct Moto {
    marca: String,
}

impl Motorizzato for Auto {
    fn accendi(&self) {
        println!("Accendendo l'auto: {}", self.modello);
    }

    fn spegni(&self) {
        println!("Spegnendo l'auto: {}", self.modello);
    }
}

impl Motorizzato for Moto {
    fn accendi(&self) {
        println!("Accendendo la moto: {}", self.marca);
    }
```

```
    fn spegni(&self) {
        println!("Spegnendo la moto: {}", self.marca);
    }
}

fn main() {
    let auto = Auto {
        modello: String::from("Fiat 500"),
    };
    let moto = Moto {
        marca: String::from("Ducati"),
    };

    auto.accendi();
    moto.accendi();
}
```

3) Definire una struct Punto3D che estende Punto2D tramite un trait Posizionabile con un metodo sposta che cambia le coordinate di un punto.

```
struct Punto2D {
    x: f64,
    y: f64,
}

struct Punto3D {
    x: f64,
    y: f64,
    z: f64,
}
```

```
trait Posizionabile {
    fn sposta(&mut self, dx: f64, dy: f64);
}

impl Posizionabile for Punto2D {
    fn sposta(&mut self, dx: f64, dy: f64) {
        self.x += dx;
        self.y += dy;
    }
}

impl Posizionabile for Punto3D {
    fn sposta(&mut self, dx: f64, dy: f64) {
        self.x += dx;
        self.y += dy;
        self.z += dx + dy;  // Regola speciale per il 3D
    }
}

fn main() {
    let mut punto2d = Punto2D { x: 1.0, y: 2.0 };
    let mut punto3d = Punto3D { x: 3.0, y: 4.0, z: 5.0 };

    punto2d.sposta(1.0, 1.0);
    punto3d.sposta(1.0, 1.0);

    println!("Punto2D: ({}, {})", punto2d.x, punto2d.y);
    println!("Punto3D: ({}, {}, {})", punto3d.x, punto3d.y, punto3d.z);
}
```

4) Definire un trait Disegnabile per disegnare forme diverse (Cerchio, Rettangolo) e mostrare l'area e il perimetro.

```
trait Disegnabile {
    fn area(&self) -> f64;
    fn perimetro(&self) -> f64;
}

struct Cerchio {
    raggio: f64,
}

struct Rettangolo {
    larghezza: f64,
    altezza: f64,
}

impl Disegnabile for Cerchio {
    fn area(&self) -> f64 {
        3.14 * self.raggio * self.raggio
    }

    fn perimetro(&self) -> f64 {
        2.0 * 3.14 * self.raggio
    }
}

impl Disegnabile for Rettangolo {
    fn area(&self) -> f64 {
        self.larghezza * self.altezza
    }
```

```
    fn perimetro(&self) -> f64 {
        2.0 * (self.larghezza + self.altezza)
    }
}

fn main() {
    let cerchio = Cerchio { raggio: 5.0 };
    let rettangolo = Rettangolo {
        larghezza: 4.0,
        altezza: 6.0,
    };

    println!("Area cerchio: {}", cerchio.area());
    println!("Perimetro cerchio: {}", cerchio.perimetro());

    println!("Area rettangolo: {}", rettangolo.area());
    println!("Perimetro rettangolo: {}", rettangolo.perimetro());
}
```

5) Creare una struct ContoCorrente che implementa un trait Rendiconto con un metodo mostra_saldo.

```
trait Rendiconto {
    fn mostra_saldo(&self) -> String;
}

struct ContoCorrente {
    saldo: f64,
}

impl Rendiconto for ContoCorrente {
```

```
    fn mostra_saldo(&self) -> String {
        format!("Il saldo del conto è: {:.2} euro.", self.saldo)
    }
}

fn main() {
    let conto = ContoCorrente { saldo: 500.0 };
    println!("{}", conto.mostra_saldo());
}
```

6) Definire una struct Dipendente con nome e salario e un trait Lavoratore con un metodo paga che calcola il salario annuale.

```
trait Lavoratore {
    fn paga_annuale(&self) -> f64;
}

struct Dipendente {
    nome: String,
    salario_mensile: f64,
}

impl Lavoratore for Dipendente {
    fn paga_annuale(&self) -> f64 {
        self.salario_mensile * 12.0
    }
}

fn main() {
    let dipendente = Dipendente {
        nome: String::from("Mario Rossi"),
```

```
        salario_mensile: 2000.0,
    };

    println!(
        "Il salario annuale di {} è: {:.2} euro.",
        dipendente.nome,
        dipendente.paga_annuale()
    );
}
```

7) Implementare un trait Misurabile per una struct Cilindro che calcola il volume e l'area della superficie.

```
trait Misurabile {
    fn volume(&self) -> f64;
    fn area_superficie(&self) -> f64;
}

struct Cilindro {
    raggio: f64,
    altezza: f64,
}

impl Misurabile for Cilindro {
    fn volume(&self) -> f64 {
        3.14 * self.raggio.powi(2) * self.altezza
    }

    fn area_superficie(&self) -> f64 {
        2.0 * 3.14 * self.raggio * (self.raggio + self.altezza)
    }
```

```
}

fn main() {
    let cilindro = Cilindro {
        raggio: 3.0,
        altezza: 5.0,
    };

    println!("Volume del cilindro: {}", cilindro.volume());
    println!("Area della superficie del cilindro: {}", cilindro.area_superficie());
}
```

8) Definire una struct Studente e un trait Esame con il metodo passa per verificare se lo studente ha superato un esame.

```
trait Esame {
    fn passa(&self, punteggio: u32) -> bool;
}

struct Studente {
    nome: String,
    materia: String,
}

impl Esame for Studente {
    fn passa(&self, punteggio: u32) -> bool {
        punteggio >= 60
    }
}

fn main() {
```

```
    let studente = Studente {
        nome: String::from("Paolo"),
        materia: String::from("Matematica"),
    };

    let punteggio = 75;
    println!(
        "{} ha passato l'esame di {}? {}",
        studente.nome,
        studente.materia,
        studente.passa(punteggio)
    );
}
```

9) Creare una struct Archivio e implementare il trait Memorizzabile con i metodi aggiungi e rimuovi elementi da un array.

```
trait Memorizzabile<T> {
    fn aggiungi(&mut self, elemento: T);
    fn rimuovi(&mut self);
}

struct Archivio<T> {
    elementi: Vec<T>,
}

impl<T> Memorizzabile<T> for Archivio<T> {
    fn aggiungi(&mut self, elemento: T) {
        self.elementi.push(elemento);
    }
```

```
    fn rimuovi(&mut self) {
        self.elementi.pop();
    }
}

fn main() {
    let mut archivio = Archivio { elementi: vec![1, 2, 3] };

    archivio.aggiungi(4);
    archivio.rimuovi();

    println!("Elementi nell'archivio: {:?}", archivio.elementi);
}
```

10) Definire una struct Animale e un trait Movimento con un metodo muovi che mostra come un animale si muove.

```
trait Movimento {
    fn muovi(&self);
}

struct Animale {
    nome: String,
    tipo: String,
}

impl Movimento for Animale {
    fn muovi(&self) {
        println!("Il {} chiamato {} si sta muovendo!", self.tipo, self.nome);
    }
}
```

```
fn main() {
    let cane = Animale {
        nome: String::from("Fido"),
        tipo: String::from("cane"),
    };

    cane.muovi();
}
```

11) Definire un trait Stampa con un metodo mostra e una funzione generica stampa_elemento che accetti qualsiasi tipo che implementa Stampa.

```
trait Stampa {
    fn mostra(&self) -> String;
}

struct Persona {
    nome: String,
    eta: u32,
}

impl Stampa for Persona {
    fn mostra(&self) -> String {
        format!("Nome: {}, Età: {}", self.nome, self.eta)
    }
}

fn stampa_elemento<T: Stampa>(elemento: T) {
    println!("{}", elemento.mostra());
}
```

```
fn main() {
    let persona = Persona {
        nome: String::from("Giovanni"),
        eta: 30,
    };

    stampa_elemento(persona);
}
```

12) Utilizzare un trait object per consentire a una lista di contenere più tipi che implementano un trait Animale, e stampare il suono di ogni animale.

```
trait Animale {
    fn suono(&self) -> &str;
}

struct Cane;
struct Gatto;

impl Animale for Cane {
    fn suono(&self) -> &str {
        "Bau!"
    }
}

impl Animale for Gatto {
    fn suono(&self) -> &str {
        "Miao!"
    }
}
```

```
fn main() {
    let animali: Vec<Box<dyn Animale>> = vec![Box::new(Cane), Box::new(Gatto)];

    for animale in animali {
        println!("Suono dell'animale: {}", animale.suono());
    }
}
```

13) Creare una funzione generica che accetta un riferimento a una lista di qualsiasi tipo di dati e restituisce l'elemento più grande.

```
fn trova_massimo<T: PartialOrd>(lista: &[T]) -> &T {
    let mut massimo = &lista[0];
    for elemento in lista.iter() {
        if elemento > massimo {
            massimo = elemento;
        }
    }
    massimo
}

fn main() {
    let numeri = vec![10, 20, 5, 30, 25];
    println!("Il numero massimo è: {}", trova_massimo(&numeri));

    let parole = vec!["casa", "gatto", "auto"];
    println!("La parola più grande è: {}", trova_massimo(&parole));
}
```

14) Creare una funzione generica che prenda due riferimenti e restituisca il più grande. Utilizzare lifetime per gestire i riferimenti.

```
fn il_piu_grande<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x > y {
        x
    } else {
        y
    }
}

fn main() {
    let parola1 = "albero";
    let parola2 = "zebra";
    let grande = il_piu_grande(parola1, parola2);

    println!("La parola più grande è: {}", grande);
}
```

15) Scrivere un trait Calcolabile per calcolare la somma di una lista di numeri e implementarlo per generics.

```
trait Calcolabile {
    fn somma(&self) -> i32;
}

impl<T: Into<i32> + Copy> Calcolabile for Vec<T> {
    fn somma(&self) -> i32 {
        self.iter().map(|&x| x.into()).sum()
    }
}
```

```
fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    println!("La somma è: {}", numeri.somma());
}
```

16) Utilizzare un trait object per implementare un logger generico che può gestire diversi tipi di output (ConsoleLogger, FileLogger).

```
trait Logger {
    fn log(&self, messaggio: &str);
}

struct ConsoleLogger;
struct FileLogger;

impl Logger for ConsoleLogger {
    fn log(&self, messaggio: &str) {
        println!("Log su console: {}", messaggio);
    }
}

impl Logger for FileLogger {
    fn log(&self, messaggio: &str) {
        println!("Log su file: {}", messaggio);
    }
}

fn esegui_logging(logger: &dyn Logger, messaggio: &str) {
    logger.log(messaggio);
}
```

```
fn main() {
    let console_logger = ConsoleLogger;
    let file_logger = FileLogger;

    esegui_logging(&console_logger, "Messaggio per la console.");
    esegui_logging(&file_logger, "Messaggio per il file.");
}
```

17) Definire una funzione generica che accetti un Vec di qualsiasi tipo e lo restituisca ordinato. Utilizzare generics e trait bounds.

```
fn ordina<T: Ord>(mut lista: Vec<T>) -> Vec<T> {
    lista.sort();
    lista
}

fn main() {
    let numeri = vec![5, 1, 4, 3, 2];
    let numeri_ordinati = ordina(numeri);
    println!("Numeri ordinati: {:?}", numeri_ordinati);

    let parole = vec!["banana", "mela", "pera"];
    let parole_ordinate = ordina(parole);
    println!("Parole ordinate: {:?}", parole_ordinate);
}
```

18) Utilizzare un lifetime per scrivere una funzione che accetta una stringa e restituisce la sottostringa più lunga trovata tra due stringhe.

```
fn sottostringa_lunga<'a>(s1: &'a str, s2: &'a str) -> &'a str {
```

```
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}

fn main() {
    let stringa1 = "Rust";
    let stringa2 = "Programmazione";

    let lunga = sottostringa_lunga(stringa1, stringa2);
    println!("La stringa più lunga è: {}", lunga);
}
```

19) Scrivere una struct generica Contenitore che può contenere qualsiasi tipo di valore e fornisca un metodo per restituire il valore.

```
struct Contenitore<T> {
    valore: T,
}

impl<T> Contenitore<T> {
    fn nuovo(valore: T) -> Self {
        Contenitore { valore }
    }

    fn ottieni_valore(&self) -> &T {
        &self.valore
    }
}
```

```
fn main() {
    let c1 = Contenitore::nuovo(42);
    let c2 = Contenitore::nuovo("Rust");

    println!("Valore nel contenitore 1: {}", c1.ottieni_valore());
    println!("Valore nel contenitore 2: {}", c2.ottieni_valore());
}
```

20) Creare una struct generica Pila che utilizzi un Vec come contenitore sottostante e fornisca i metodi push e pop.

```
struct Pila<T> {
    elementi: Vec<T>,
}

impl<T> Pila<T> {
    fn nuova() -> Self {
        Pila {
            elementi: Vec::new(),
        }
    }

    fn push(&mut self, elemento: T) {
        self.elementi.push(elemento);
    }

    fn pop(&mut self) -> Option<T> {
        self.elementi.pop()
    }
}
```

```
fn main() {
    let mut pila = Pila::nuova();

    pila.push(10);
    pila.push(20);
    pila.push(30);

    println!("Pop dalla pila: {:?}", pila.pop());
    println!("Pop dalla pila: {:?}", pila.pop());
}
```

21) Scrivere una funzione che prenda in input due stringhe con lifetime e restituisca quella più lunga.

```
fn stringa_piu_lunga<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}

fn main() {
    let stringa1 = "ciao";
    let stringa2 = "mondo!";
    let lunga = stringa_piu_lunga(stringa1, stringa2);

    println!("La stringa più lunga è: {}", lunga);
}
```

22) Creare una struct Libro con due riferimenti a stringhe e utilizzare lifetime per evitare errori di borrowing.

```
struct Libro<'a> {
    titolo: &'a str,
    autore: &'a str,
}

impl<'a> Libro<'a> {
    fn descrizione(&self) -> String {
        format!("{} scritto da {}", self.titolo, self.autore)
    }
}

fn main() {
    let titolo = String::from("Rust Programming");
    let autore = String::from("John Doe");

    let libro = Libro {
        titolo: &titolo,
        autore: &autore,
    };

    println!("{}", libro.descrizione());
}
```

23) Implementare un trait Somma che permette la somma di due struct Punto che contengono coordinate x e y.

```
struct Punto {
    x: i32,
```

```
    y: i32,
}

trait Somma {
    fn somma(&self, altro: &Self) -> Self;
}

impl Somma for Punto {
    fn somma(&self, altro: &Self) -> Self {
        Punto {
            x: self.x + altro.x,
            y: self.y + altro.y,
        }
    }
}

fn main() {
    let punto1 = Punto { x: 1, y: 2 };
    let punto2 = Punto { x: 3, y: 4 };

    let risultato = punto1.somma(&punto2);
    println!("Risultato: ({}, {})", risultato.x, risultato.y);
}
```

24) Scrivere una funzione generica che accetta un riferimento a un Vec<T> e restituisce il primo elemento. Usare lifetime per gestire il borrowing.

```
fn primo_elemento<'a, T>(lista: &'a Vec<T>) -> Option<&'a T> {
    lista.first()
}
```

```
fn main() {
    let numeri = vec![10, 20, 30, 40];
    if let Some(primo) = primo_elemento(&numeri) {
        println!("Il primo numero è: {}", primo);
    }
}
```

25) Utilizzare un trait Display per implementare la stampa personalizzata di una struct Persona che include nome ed età.

```
use std::fmt;

struct Persona {
    nome: String,
    eta: u32,
}

impl fmt::Display for Persona {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "Nome: {}, Età: {}", self.nome, self.eta)
    }
}

fn main() {
    let persona = Persona {
        nome: String::from("Giulia"),
        eta: 25,
    };

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

26) Implementare un pattern che utilizza il trait Iterator per sommare gli elementi di un iteratore, evitando anti-pattern di duplicazione del codice.

```
fn somma_elementi<I>(iter: I) -> i32
where
    I: Iterator<Item = i32>,
{
    iter.sum()
}

fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    let somma = somma_elementi(numeri.into_iter());
    println!("La somma è: {}", somma);
}
```

In Rust, l'istruzione ***where*** non è utilizzata solo nei pattern, ma principalmente per migliorare la leggibilità e flessibilità quando si specificano i vincoli sui tipi generici.

Nella programmazione con generics, spesso hai bisogno di vincolare i tipi a certi trait per poterli usare con determinate operazioni. Normalmente, questi vincoli vengono specificati direttamente nella firma della funzione o del metodo. Tuttavia, se ci sono molti vincoli o la firma diventa lunga, l'istruzione where permette di spostare questi vincoli alla fine della dichiarazione, rendendo il codice più leggibile e ordinato.

Vediamo un esempio con e senza l'uso di where:

Senza where:

```
fn somma<T: std::ops::Add<Output = T> + Copy>(a: T, b: T) -> T {
    a + b
}
```

Con where:

```
fn somma<T>(a: T, b: T) -> T
where
    T: std::ops::Add<Output = T> + Copy,
{
    a + b
}
```

In entrambi i casi, la funzione richiede che il tipo generico T implementi il trait Add (che consente l'uso dell'operatore +) e che possa essere copiato (con Copy).

L'istruzione where può essere usata anche in dichiarazioni di tipi, implementazioni di trait, struct, e altri contesti, non solo nei pattern matching. Questo la rende uno strumento molto versatile per la gestione dei vincoli sui generics in Rust.

27) Creare una funzione asincrona che effettua una richiesta HTTP e gestisce il timeout utilizzando il crate tokio e trait asincroni.

```
use tokio::time::{timeout, Duration};

async fn richiesta_http() -> &'static str {
    // Simulazione di una richiesta HTTP
    tokio::time::sleep(Duration::from_secs(2)).await;
    "Risposta ricevuta"
}

#[tokio::main]
async fn main() {
    match timeout(Duration::from_secs(3), richiesta_http()).await {
        Ok(result) => println!("Risultato: {}", result),
```

```
        Err(_) => println!("Timeout scaduto"),
    }
}
```

28) Definire un anti-pattern in Rust dove viene creata una struct con riferimenti non necessari e mostrare la sua ottimizzazione.

```
struct Libro<'a> {
    titolo: &'a str,
    autore: &'a str,
}

fn main() {
    let libro = Libro {
        titolo: "Il Rustaceo",
        autore: "Autore Anonimo",
    };
    // La struct non necessita di riferimenti qui
}
```

Pattern ottimizzato:

```
struct Libro {
    titolo: String,
    autore: String,
}

fn main() {
    let libro = Libro {
        titolo: String::from("Il Rustaceo"),
        autore: String::from("Autore Anonimo"),
```

```
    };
}
```

29) Utilizzare un trait per rappresentare un comportamento generico di un database asincrono. Definire un metodo inserisci che può essere chiamato su diverse implementazioni di database.

```
use async_trait::async_trait;

#[async_trait]
trait Database {
    async fn inserisci(&self, dati: &str);
}

struct MySQL;
struct PostgreSQL;

#[async_trait]
impl Database for MySQL {
    async fn inserisci(&self, dati: &str) {
        println!("Inserimento su MySQL: {}", dati);
    }
}

#[async_trait]
impl Database for PostgreSQL {
    async fn inserisci(&self, dati: &str) {
        println!("Inserimento su PostgreSQL: {}", dati);
    }
}

#[tokio::main]
```

```
async fn main() {
    let mysql = MySQL;
    let postgres = PostgreSQL;
    mysql.inserisci("Dati per MySQL").await;
    postgres.inserisci("Dati per PostgreSQL").await;
}
```

30) Utilizzare lifetimes in combinazione con generics e trait per implementare un wrapper che mantenga un riferimento a un valore di tipo generico e fornisca un metodo per restituire quel valore.

```
struct Wrapper<'a, T> {
    valore: &'a T,
}

impl<'a, T> Wrapper<'a, T> {
    fn nuovo(valore: &'a T) -> Self {
        Wrapper { valore }
    }

    fn ottieni_valore(&self) -> &T {
        self.valore
    }
}

fn main() {
    let numero = 42;
    let wrapper = Wrapper::nuovo(&numero);

    println!("Valore nel wrapper: {}", wrapper.ottieni_valore());
}
```

Capitolo 5

1) Scrivere una funzione che prenda una stringa in prestito (borrow) e restituisca la sua lunghezza, ma senza trasferire l'ownership.

```
fn lunghezza(s: &String) -> usize {
    s.len()
}

fn main() {
    let stringa = String::from("Ciao");
    let len = lunghezza(&stringa);
    println!("La lunghezza è: {}", len);
}
```

2) Scrivere una funzione che prenda un riferimento mutable a un vettore e ne aggiunga un elemento, dimostrando il mutabile borrowing.

```
fn aggiungi_elemento(v: &mut Vec<i32>, elemento: i32) {
    v.push(elemento);
}

fn main() {
    let mut numeri = vec![1, 2, 3];
    aggiungi_elemento(&mut numeri, 4);
    println!("{:?}", numeri);
```

```
}
```

3) Creare una funzione che prenda due riferimenti a stringhe e restituisca la stringa più lunga, dimostrando l'uso dei lifetimes.

```
fn stringa_piu_lunga<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}

fn main() {
    let s1 = "Ciao";
    let s2 = "Mondo";
    let risultato = stringa_piu_lunga(s1, s2);
    println!("La stringa più lunga è: {}", risultato);
}
```

4) Scrivere una funzione che prenda un riferimento a una stringa e restituisca un nuovo riferimento mutabile ad essa, ma gestisca correttamente i lifetimes.

```
fn muta_stringa<'a>(s: &'a mut String) -> &'a mut String {
    s.push_str(" Rust!");
    s
}

fn main() {
    let mut saluto = String::from("Ciao");
    let mutato = muta_stringa(&mut saluto);
```

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

5) Scrivere una funzione che prenda una struttura con un campo mutabile e lo modifichi tramite un riferimento mutabile, dimostrando l'ownership in una struct.

```
struct ContoBancario {
    saldo: i32,
}

fn deposita(conto: &mut ContoBancario, importo: i32) {
    conto.saldo += importo;
}

fn main() {
    let mut mio_conto = ContoBancario { saldo: 100 };
    deposita(&mut mio_conto, 50);
    println!("Saldo aggiornato: {}", mio_conto.saldo);
}
```

6) Scrivere una funzione che prenda due riferimenti mutabili a variabili diverse e ne modifichi i valori, dimostrando che Rust non consente due mutabili borrows simultanei sulla stessa variabile.

```
fn scambia(a: &mut i32, b: &mut i32) {
    let temp = *a;
    *a = *b;
    *b = temp;
}

fn main() {
    let mut x = 5;
```

```
    let mut y = 10;
    scambia(&mut x, &mut y);
    println!("x: {}, y: {}", x, y);
}
```

7) Implementare una funzione che accetti un riferimento immutabile e uno mutabile nella stessa funzione e dimostrare come Rust gestisce i borrow.

```
fn processa_valori(val1: &i32, val2: &mut i32) {
    *val2 += *val1;
}

fn main() {
    let val1 = 10;
    let mut val2 = 20;
    processa_valori(&val1, &mut val2);
    println!("Valore aggiornato: {}", val2);
}
```

8) Creare una funzione che accetti un oggetto che implementa il trait Drop e dimostrare l'ownership alla fine del ciclo di vita dell'oggetto.

```
struct Risorsa;

impl Drop for Risorsa {
    fn drop(&mut self) {
        println!("Risorsa liberata!");
    }
}

fn usa_risorsa() {
```

```
    let _risorsa = Risorsa;
    println!("Sto usando la risorsa");
}

fn main() {
    usa_risorsa();
}
```

9) Scrivere una funzione che accetti una chiusura che prende in prestito una variabile immutabile e dimostrare l'uso del borrowing in una chiusura.

```
fn esegui<F>(f: F)
where
    F: Fn() -> i32,
{
    println!("Risultato: {}", f());
}

fn main() {
    let x = 5;
    esegui(|| x + 1);
}
```

10) Creare una funzione che accetti una chiusura che prende in prestito mutabilmente una variabile e modifichi il suo valore, dimostrando il borrowing mutabile in una chiusura.

```
fn esegui_mut<F>(mut f: F)
where
    F: FnMut(),
{
    f();
```

```
}

fn main() {
    let mut x = 5;
    esegui_mut(|| x += 1);
    println!("Valore di x: {}", x);
}
```

11) Scrivere una funzione che accetti una variabile e la trasferisca (move) in un'altra variabile, dimostrando il comportamento implicito del move.

```
fn trasferisci_ownership(s: String) {
    println!("Stringa trasferita: {}", s);
}

fn main() {
    let s = String::from("Ciao, Rust!");
    trasferisci_ownership(s);  // Ownership trasferita
    // println!("{}", s);  // Errore: l'ownership di `s` è stata trasferita
}
```

12) Scrivere una funzione che prenda in prestito (borrow) una variabile immutabile e dimostrare come evitare il move implicito.

```
fn prendi_in_prestito(s: &String) {
    println!("Stringa in prestito: {}", s);
}

fn main() {
    let s = String::from("Ciao, Rust!");
    prendi_in_prestito(&s);  // Solo prestito, ownership non trasferita
```

```
    println!("Stringa originale: {}", s);  // Accessibile dopo il prestito
}
```

13) Scrivere una funzione che accetti un riferimento mutabile e lo modifichi, dimostrando come un mutabile borrow non trasferisce ownership ma permette modifiche.

```
fn modifica_valore(v: &mut i32) {
    *v += 10;
}

fn main() {
    let mut numero = 5;
    modifica_valore(&mut numero);
    println!("Valore modificato: {}", numero);
}
```

14) Scrivere una funzione che prenda un riferimento immutabile e uno mutabile allo stesso tempo e gestisca correttamente i prestiti multipli.

```
fn leggi_e_modifica(valore: &i32, mutabile: &mut i32) {
    println!("Valore immutabile: {}", valore);
    *mutabile += *valore;
}

fn main() {
    let immutabile = 10;
    let mut mutabile = 5;
    leggi_e_modifica(&immutabile, &mut mutabile);
    println!("Valore mutabile modificato: {}", mutabile);
}
```

15) Scrivere una funzione che utilizzi il concetto di move esplicito, restituendo una variabile da una funzione per trasferire l'ownership indietro.

```
fn trasferisci_e_restituisci(s: String) -> String {
    println!("Ownership trasferita a funzione: {}", s);
    s  // Restituisce l'ownership
}

fn main() {
    let s = String::from("Ciao, Rust!");
    let s = trasferisci_e_restituisci(s);  // Ownership ritornata
    println!("Ownership ritornata al chiamante: {}", s);
}
```

16) Scrivere una funzione che accetti un trait bound con un riferimento immutabile, dimostrando l'interazione tra trait bounds e borrowing.

```
fn stampa<T: AsRef<str>>(s: &T) {
    println!("{}", s.as_ref());
}

fn main() {
    let s = String::from("Ciao, Rust!");
    stampa(&s);
}
```

17) Scrivere una funzione che accetti un trait bound con un riferimento mutabile e modifichi la struttura sottostante, dimostrando il trait bound con prestito mutabile.

```
fn aggiungi_a_stringa<T: AsMut<String>>(s: &mut T) {
    s.as_mut().push_str(" World!");
```

```
}

fn main() {
    let mut s = String::from("Ciao,");
    aggiungi_a_stringa(&mut s);
    println!("{}", s);
}
```

18) Scrivere una funzione che sposti (move) un valore in una struttura e dimostri come i move funzionano con le struct.

```
struct Contenitore {
    valore: String,
}

fn sposta_in_struct(s: String) -> Contenitore {
    Contenitore { valore: s }
}

fn main() {
    let s = String::from("Ciao, Rust!");
    let contenitore = sposta_in_struct(s);  // Ownership trasferita alla struct
    println!("Contenuto nella struct: {}", contenitore.valore);
}
```

19) Scrivere una funzione generica che accetti un trait bound con riferimento e che implementi un'operazione di borrowing su una struct generica.

```
fn mostra_contenuto<T: AsRef<str>>(contenitore: &T) {
    println!("Contenuto: {}", contenitore.as_ref());
}
```

```
struct Contenitore {
    valore: String,
}

impl AsRef<str> for Contenitore {
    fn as_ref(&self) -> &str {
        &self.valore
    }
}

fn main() {
    let c = Contenitore {
        valore: String::from("Rust è potente!"),
    };
    mostra_contenuto(&c);
}
```

20) Scrivere una funzione che utilizzi i lifetimes per restituire un riferimento a una parte di una struttura, dimostrando l'uso di lifetimes in relazione ai trait e borrowing.

```
struct Contenitore<'a> {
    valore: &'a str,
}

impl<'a> Contenitore<'a> {
    fn ottieni_valore(&self) -> &'a str {
        self.valore
    }
}
```

```
fn main() {
    let testo = String::from("Ciao, Rust!");
    let contenitore = Contenitore { valore: &testo };
    println!("Valore dal contenitore: {}", contenitore.ottieni_valore());
}
```

21) Creare un puntatore intelligente Box per gestire un valore su heap e dimostrare come l'ownership è trasferita.

```
fn main() {
    let x = Box::new(5);  // Puntatore intelligente su heap
    println!("Valore in Box: {}", x);  // Box de-referenziato automaticamente
}
```

22) Utilizzare Rc per gestire la condivisione di dati in più parti del programma, dimostrando come il conteggio di riferimenti avviene con Rc.

```
use std::rc::Rc;

fn main() {
    let valore = Rc::new(String::from("Ciao, Rust!"));
    let a = Rc::clone(&valore);  // Clone incrementa il conteggio di riferimenti
    let b = Rc::clone(&valore);  // Incrementa di nuovo
    println!("Conteggio di riferimenti: {}", Rc::strong_count(&valore));  // 3
    println!("Valore condiviso: {}", a);
}
```

23) Utilizzare Arc per gestire la condivisione di dati tra thread e dimostrare come Arc si differenzia da Rc per la gestione concorrente.

```
use std::sync::Arc;
```

```
use std::thread;

fn main() {
    let valore = Arc::new(String::from("Ciao, mondo!"));
    let valore1 = Arc::clone(&valore);
    let handle = thread::spawn(move || {
        println!("Thread 1: {}", valore1);
    });

    println!("Main thread: {}", valore);
    handle.join().unwrap();
}
```

24) Utilizzare RefCell per permettere mutabilità interna in una struttura, anche quando è referenziata in modo immutabile.

```
use std::cell::RefCell;

fn main() {
    let valore = RefCell::new(5);

    *valore.borrow_mut() += 10;  // Mutabilità interna
    println!("Valore modificato: {}", valore.borrow());
}
```

25) Dimostrare l'uso di Cell per la mutabilità interna quando si trattano tipi primitivi copiabili.

```
use std::cell::Cell;

fn main() {
    let valore = Cell::new(5);
```

```
    valore.set(10);  // Modifica attraverso mutabilità interna
    println!("Nuovo valore: {}", valore.get());
}
```

26) Utilizzare Mutex per garantire accesso esclusivo a una variabile condivisa tra thread, prevenendo condizioni di gara.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let valore = Arc::new(Mutex::new(0));
    let valore1 = Arc::clone(&valore);

    let handle = thread::spawn(move || {
        let mut num = valore1.lock().unwrap();
        *num += 1;  // Accesso esclusivo
    });

    handle.join().unwrap();
    println!("Valore dopo thread: {}", *valore.lock().unwrap());
}
```

27) Utilizzare Arc e Mutex insieme per condividere in sicurezza una variabile tra thread multipli e prevenire condizioni di gara

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
```

```
    let valore = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let valore = Arc::clone(&valore);
        let handle = thread::spawn(move || {
            let mut num = valore.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Valore finale: {}", *valore.lock().unwrap());
}
```

28) Utilizzare Rc e RefCell insieme per gestire mutabilità interna in una struttura condivisa senza usare multi-threading.

```
use std::rc::Rc;
use std::cell::RefCell;

struct Contenitore {
    valore: Rc<RefCell<i32>>,
}

fn main() {
    let valore = Rc::new(RefCell::new(10));
```

```
    let contenitore1 = Contenitore { valore: Rc::clone(&valore) };
    let contenitore2 = Contenitore { valore: Rc::clone(&valore) };

    *contenitore1.valore.borrow_mut() += 5;  // Modifica da contenitore1
    println!("Valore in contenitore 1: {}", contenitore1.valore.borrow());
    println!("Valore in contenitore 2: {}", contenitore2.valore.borrow());

    *contenitore2.valore.borrow_mut() += 10;  // Modifica da contenitore2
    println!("Valore aggiornato in contenitore 1: {}", contenitore1.valore.borrow());
    println!("Valore aggiornato in contenitore 2: {}", contenitore2.valore.borrow());
}
```

In questo esempio, usiamo Rc per condividere la proprietà del valore e RefCell per mutare il valore internamente, anche se il valore è immutabile dal punto di vista esterno.

29) Utilizzare Arc, Mutex e una funzione asincrona per gestire la concorrenza con un contatore condiviso che viene modificato da più task.

```
use std::sync::{Arc, Mutex};
use tokio::task;

#[tokio::main]
async fn main() {
    let contatore = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..5 {
        let contatore = Arc::clone(&contatore);
        let handle = task::spawn(async move {
            let mut num = contatore.lock().unwrap();
            *num += 1;
```

```
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.await.unwrap();
    }

    println!("Valore finale del contatore: {}", *contatore.lock().unwrap());
}
```

Qui, usiamo Arc e Mutex per gestire la mutabilità condivisa tra task asincroni e prevenire condizioni di gara durante l'accesso al contatore.

30) Utilizzare Cell per mutare un valore copiabile in una funzione che richiede l'accesso immutabile alla struttura, dimostrando la mutabilità interna.

```
use std::cell::Cell;

struct Punto {
    x: i32,
    y: Cell<i32>,  // Cell permette di mutare y anche se Punto è immutabile
}

fn main() {
    let punto = Punto { x: 5, y: Cell::new(10) };

    // Mutare y anche se punto è immutabile
    punto.y.set(20);

    println!("Punto: x = {}, y = {}", punto.x, punto.y.get());
```

```
}
```

Usiamo Cell per permettere la modifica del campo y all'interno di una struttura che potrebbe essere referenziata in modo immutabile, sfruttando la mutabilità interna per evitare borseggio immutabile.

Capitolo 6

1) Scrivere un programma che verifichi se una stringa contiene solo lettere maiuscole utilizzando un'espressione regolare.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"^[A-Z]+$").unwrap();
    let test_string = "HELLO";

    if regex.is_match(test_string) {
        println!("La stringa contiene solo lettere maiuscole.");
    } else {
        println!("La stringa contiene altri caratteri.");
    }
}
```

2) Scrivere un programma che controlli se una stringa è un indirizzo email valido.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"^[\w\.-]+@[\w\.-]+\.[a-z]{2,}$").unwrap();
    let email = "test@example.com";
```

```
    if regex.is_match(email) {
        println!("Indirizzo email valido.");
    } else {
        println!("Indirizzo email non valido.");
    }
}
```

3) Scrivere un programma che sostituisca tutte le cifre in una stringa con il carattere #.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"\d").unwrap();
    let input = "Il mio numero di telefono è 12345";
    let result = regex.replace_all(input, "#");

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

4) Scrivere un programma che estragga tutte le parole che iniziano con la lettera "a" da una stringa.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"\b[aA]\w*\b").unwrap();
    let testo = "Avocado è un frutto, mentre arancia è un agrume.";

    for cattura in regex.captures_iter(testo) {
        println!("Parola trovata: {}", &cattura[0]);
    }
}
```

5) Scrivere un programma che verifichi se una stringa rappresenta un numero decimale valido.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"^\d+(\.\d+)?$").unwrap();
    let numero = "123.45";

    if regex.is_match(numero) {
        println!("La stringa è un numero decimale valido.");
    } else {
        println!("La stringa non è un numero decimale valido.");
    }
}
```

6) Scrivere un programma che trovi tutte le date nel formato gg-mm-aaaa in una stringa.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"\b\d{2}-\d{2}-\d{4}\b").unwrap();
    let testo = "Le date importanti sono 12-05-2022 e 23-11-2023.";

    for cattura in regex.captures_iter(testo) {
        println!("Data trovata: {}", &cattura[0]);
    }
}
```

7) Scrivere un programma che verifichi se una password è valida (deve contenere almeno una lettera maiuscola, una minuscola, un numero e un simbolo speciale).

```
use regex::Regex;


fn main() {
    let regex = Regex::new(r"^(?=.*[a-z])(?=.*[A-
Z])(?=.*\d)(?=.*[\W_]).{8,}$").unwrap();
    let password = "Password123!";


    if regex.is_match(password) {
        println!("La password è valida.");
    } else {
        println!("La password non è valida.");
    }
}
```

8) Scrivere un programma che verifichi se un numero di telefono nel formato +XX-XXXXXXXXXX è valido.

```
use regex::Regex;


fn main() {
    let regex = Regex::new(r"^\+\d{2}-\d{10}$").unwrap();
    let numero_telefono = "+39-1234567890";


    if regex.is_match(numero_telefono) {
        println!("Numero di telefono valido.");
    } else {
        println!("Numero di telefono non valido.");
    }
}
```

9) Scrivere un programma che rimuova tutti i caratteri non alfabetici da una stringa.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"[^a-zA-Z]").unwrap();
    let testo = "Ciao! Come va? 123";
    let risultato = regex.replace_all(testo, "");

    println!("Stringa pulita: {}", risultato);
}
```

10) Scrivere un programma che estragga tutti i link URL da un testo.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"https?://[^\s]+").unwrap();
    let testo = "Visita https://www.example.com e https://rust-lang.org per maggiori informazioni.";

    for cattura in regex.captures_iter(testo) {
        println!("Link trovato: {}", &cattura[0]);
    }
}
```

11) Scrivere un programma che verifichi se una stringa contiene caratteri alfanumerici utilizzando sequenze di escape predefinite.

```
use regex::Regex;
```

```
fn main() {
    let regex = Regex::new(r"^\w+$").unwrap();  // \w corrisponde a [a-zA-Z0-9_]
    let testo = "Rust2024";

    if regex.is_match(testo) {
        println!("La stringa contiene solo caratteri alfanumerici.");
    } else {
        println!("La stringa contiene caratteri non alfanumerici.");
    }
}
```

12) Scrivere un programma che verifichi se una stringa rappresenta un numero intero (positivo o negativo).

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"^-?\d+$").unwrap();  // \d corrisponde a [0-9], il segno ? indica che il "-" è facoltativo
    let numero = "-123";

    if regex.is_match(numero) {
        println!("La stringa è un numero intero valido.");
    } else {
        println!("La stringa non è un numero intero valido.");
    }
}
```

13) Scrivere un programma che verifichi se una stringa contiene spazi bianchi (spazio, tab, newline) utilizzando \s.

```
use regex::Regex;
```

```
fn main() {
    let regex = Regex::new(r"\s").unwrap();  // \s corrisponde a qualsiasi spazio bianco
    let testo = "Ciao mondo!";

    if regex.is_match(testo) {
        println!("La stringa contiene spazi bianchi.");
    } else {
        println!("La stringa non contiene spazi bianchi.");
    }
}
```

14) Scrivere un programma che verifichi se una stringa inizia con una cifra e termina con un punto.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"^\d.*\.$").unwrap();  // ^ indica inizio, \d una cifra, .* qualsiasi carattere, \. un punto, $ fine
    let testo = "5 Questo è un esempio.";

    if regex.is_match(testo) {
        println!("La stringa inizia con una cifra e termina con un punto.");
    } else {
        println!("La stringa non rispetta il formato.");
    }
}
```

15) Scrivere un programma che verifichi se una stringa contiene un indirizzo IP IPv4 valido.

```
use regex::Regex;
```

```
fn main() {
    let regex = Regex::new(r"^(\d{1,3}\.){3}\d{1,3}$").unwrap();  // \d{1,3} rappresenta un numero di 1-3 cifre, \. è il punto
    let ip = "192.168.1.1";

    if regex.is_match(ip) {
        println!("L'indirizzo IP è valido.");
    } else {
        println!("L'indirizzo IP non è valido.");
    }
}
```

16) Scrivere un programma che trovi tutti i caratteri che non siano lettere o numeri in una stringa.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"[^\w]").unwrap();  // [^\w] corrisponde a tutto ciò che non è alfanumerico
    let testo = "Esempio: Rust 2024!";

    for match_obj in regex.find_iter(testo) {
        println!("Carattere non alfanumerico trovato: {}", match_obj.as_str());
    }
}
```

17) Scrivere un programma che verifichi se una stringa contiene solo caratteri di controllo, come newline o tab.

```
use regex::Regex;
```

```
fn main() {
    let regex = Regex::new(r"^[\x00-\x1F]+$").unwrap();  // \x00-\x1F corrisponde ai caratteri di controllo ASCII
    let testo = "\n\t";

    if regex.is_match(testo) {
        println!("La stringa contiene solo caratteri di controllo.");
    } else {
        println!("La stringa contiene altri caratteri.");
    }
}
```

18) Scrivere un programma che verifichi se una stringa contiene una sequenza di parole separate da virgole, ma senza spazi extra.

```
use regex::Regex;

fn main() {
    let regex = Regex::new(r"^(\w+,)+\w+$").unwrap();  // (\w+,)+\w+ corrisponde a parole separate da virgole senza spazi
    let testo = "ciao,mondo,2024";

    if regex.is_match(testo) {
        println!("La stringa è una sequenza valida di parole separate da virgole.");
    } else {
        println!("La stringa non è valida.");
    }
}
```

19) Scrivere un programma che estragga tutte le parole che contengono solo lettere minuscole da una stringa.

```
use regex::Regex;


fn main() {
    let regex = Regex::new(r"\b[a-z]+\b").unwrap();  // \b indica un confine di
parola, [a-z]+ solo lettere minuscole
    let testo = "Il gatto corre veloce, ma il cane no.";


    for match_obj in regex.find_iter(testo) {
        println!("Parola trovata: {}", match_obj.as_str());
    }
}
```

20) Scrivere un programma che verifichi se una stringa contiene una data nel formato MM/GG/AAAA utilizzando gruppi di cattura.

```
use regex::Regex;


fn main() {
    let regex =
Regex::new(r"(?P<month>\d{2})/(?P<day>\d{2})/(?P<year>\d{4})").unwrap();  // gruppi
di cattura per mese, giorno e anno
    let data = "12/31/2023";


    if let Some(captures) = regex.captures(data) {
        println!("Mese: {}, Giorno: {}, Anno: {}",
                 &captures["month"], &captures["day"], &captures["year"]);
    } else {
        println!("Formato data non valido.");
    }
}
```

21) Scrivere un programma che verifichi se una stringa rappresenta un indirizzo MAC (Media Access Control) valido.

```
use regex::Regex;


fn main() {
    let regex = Regex::new(r"^([0-9A-Fa-f]{2}:){5}[0-9A-Fa-f]{2}$").unwrap();  // verifica formato MAC
    let mac = "01:23:45:67:89:AB";


    if regex.is_match(mac) {
        println!("L'indirizzo MAC è valido.");
    } else {
        println!("L'indirizzo MAC non è valido.");
    }
}
```

([0-9A-Fa-f]{2}:){5} verifica sei coppie di caratteri esadecimali separate da :.

[0-9A-Fa-f]{2} rappresenta l'ultima coppia esadecimale.

22) Scrivere un programma che verifichi se una stringa rappresenta un numero di porta TCP/UDP valido (tra 1 e 65535)

```
use regex::Regex;


fn main() {
    let regex = Regex::new(r"^([1-9]\d{0,3}|[1-5]\d{4}|6[0-4]\d{3}|65[0-4]\d{2}|655[0-2]\d|6553[0-5])$").unwrap();  // verifica porta
    let porta = "8080";


    if regex.is_match(porta) {
        println!("La porta è valida.");
    } else {
```

```
        println!("La porta non è valida.");
    }
}
```

La regex copre i numeri tra 1 e 65535 suddividendo l'intervallo in gruppi con regole specifiche per ogni range.

23) Scrivere un programma che analizzi un log di server web Apache e verifichi la correttezza del formato di ogni riga, includendo l'indirizzo IP del client, la data, il metodo HTTP e lo stato di risposta.

```
use regex::Regex;

fn main() {
    let log_entry = r#"192.168.1.1 - - [27/Sep/2024:10:00:00 +0000] "GET /index.html HTTP/1.1" 200 2326"#;
    let log_pattern = r#"^(\d{1,3}\.){3}\d{1,3} - - \[\d{2}/[A-Za-z]{3}/\d{4}:\d{2}:\d{2}:\d{2} \+\d{4}\] "(GET|POST|PUT|DELETE|PATCH) .+ HTTP/1.1" \d{3} \d+$"#;

    let regex = Regex::new(log_pattern).unwrap();

    if regex.is_match(log_entry) {
        println!("La riga del log è valida.");
    } else {
        println!("La riga del log non è valida.");
    }
}
```

(\d{1,3}\.){3}\d{1,3} rappresenta un indirizzo IP IPv4.

\[...\] gestisce la data nel formato [giorno/mese/anno:ora:minuto:secondo timezone].

Il metodo HTTP può essere GET, POST, PUT, DELETE o PATCH.

La parte finale include il codice di stato e il numero di byte restituiti.

24) Scrivere un programma che estragga tutti gli indirizzi IP da un file di log del server, compresi gli indirizzi IPv4 e IPv6, e che li memorizzi in un HashSet senza duplicati.

```
use regex::Regex;
use std::collections::HashSet;

fn main() {
    let logs = vec![
        "Client 192.168.0.1 connected.",
        "Client 2001:0db8:85a3:0000:0000:8a2e:0370:7334 connected.",
        "Client 192.168.0.1 connected again.",
    ];

    let ip_pattern = r#"(\b\d{1,3}\.){3}\d{1,3}\b|([a-fA-F0-9]{1,4}:){7}[a-fA-F0-9]{1,4}"#;
    let regex = Regex::new(ip_pattern).unwrap();
    let mut unique_ips = HashSet::new();

    for log in logs {
        for cap in regex.captures_iter(log) {
            unique_ips.insert(cap[0].to_string());
        }
    }

    for ip in unique_ips {
        println!("Indirizzo IP trovato: {}", ip);
    }
}
```

La regex permette di catturare sia indirizzi IPv4 che IPv6.

(\b\d{1,3}\.){3}\d{1,3} gestisce gli indirizzi IPv4.

([a-fA-F0-9]{1,4}:){7}[a-fA-F0-9]{1,4} gestisce gli indirizzi IPv6.

Gli IP trovati vengono memorizzati in un HashSet per eliminare duplicati.

25) Scrivere un programma che controlli un file di configurazione di un server DNS e verifichi la correttezza dei record A, CNAME e MX, estrapolando i domini e indirizzi IP associati.

```
use regex::Regex;


fn main() {
    let config_lines = vec![
        "example.com. 3600 IN A 192.168.0.1",
        "mail.example.com. 3600 IN MX 10 mailserver.example.com.",
        "www.example.com. 3600 IN CNAME example.com.",
    ];


    let record_pattern =
r#"(?P<domain>\S+)\s+\d+\s+IN\s+(?P<type>A|CNAME|MX)\s+(?P<value>\S+)"#;
    let regex = Regex::new(record_pattern).unwrap();


    for line in config_lines {
        if let Some(caps) = regex.captures(line) {
            let record_type = &caps["type"];
            let domain = &caps["domain"];
            let value = &caps["value"];
            println!("Tipo di record: {}, Dominio: {}, Valore: {}", record_type,
domain, value);
        }
    }
}
```

La regex cerca i record DNS di tipo A, CNAME e MX.

(?P<domain>\S+) cattura il dominio.

(?P<type>A|CNAME|MX) cattura il tipo di record.

(?P<value>\S+) cattura il valore del record, che può essere un indirizzo IP o un dominio.

26) Scrivere un programma che verifichi se una stringa rappresenta un indirizzo URL valido con una porta specificata, estraendo l'host e la porta per eventuali controlli di sicurezza.

```
use regex::Regex;


fn main() {
    let url = "https://www.example.com:8080/path";
    let url_pattern = r#"^https?://(?P<host>[a-zA-Z\d\.-]+):(?P<port>\d+)(/.*)?$"#;


    let regex = Regex::new(url_pattern).unwrap();


    if let Some(caps) = regex.captures(url) {
        let host = &caps["host"];
        let port = &caps["port"];
        println!("Host: {}, Porta: {}", host, port);
    } else {
        println!("L'URL non è valido.");
    }
}
```

(?P<host>[a-zA-Z\d\.-]+) cattura l'host.

(?P<port>\d+) cattura la porta.

La regex gestisce sia http che https e prevede una porta esplicita.

Capitolo 7

1) Scrivere un programma che organizza il codice in due moduli: matematica e stringhe. Il primo deve contenere una funzione per sommare due numeri, mentre il modulo stringhe deve contenere una funzione che concatena due stringhe. Entrambe devono essere pubbliche (pub).

```
mod matematica {
    pub fn somma(a: i32, b: i32) -> i32 {
        a + b
    }
}

mod stringhe {
    pub fn concatena(s1: &str, s2: &str) -> String {
        format!("{}{}", s1, s2)
    }
}

fn main() {
    let risultato = matematica::somma(5, 3);
    println!("Somma: {}", risultato);

    let stringa = stringhe::concatena("Ciao, ", "mondo!");
    println!("Stringa concatenata: {}", stringa);
}
```

Il codice è organizzato in due moduli, ciascuno contenente funzioni pubbliche (pub).

Le funzioni sono accessibili dall'esterno del modulo tramite nome_modulo::funzione.

2) Scrivere un programma che utilizza sotto-moduli per gestire operazioni matematiche avanzate. Creare un modulo matematica, con due sotto-moduli: algebra per funzioni di moltiplicazione e geometria per calcolare l'area di un quadrato. Solo la funzione dell'area deve essere pubblica.

```
mod matematica {
    pub mod geometria {
        pub fn area_quadrato(lato: f64) -> f64 {
            lato * lato
        }
    }

    mod algebra {
        pub fn moltiplica(a: i32, b: i32) -> i32 {
            a * b
        }
    }
}

fn main() {
    let area = matematica::geometria::area_quadrato(5.0);
    println!("Area del quadrato: {}", area);

    // La funzione moltiplica non è accessibile da fuori il modulo
`matematica::algebra`.
}
```

I sotto-moduli sono definiti all'interno del modulo principale.

La funzione area_quadrato è pubblica e accessibile, mentre la funzione moltiplica non è visibile all'esterno.

3) Creare un programma che utilizzi un modulo per la gestione di un server. Il modulo server deve contenere due funzioni, una per avviare il server e una per gestire una richiesta. Solo la funzione che avvia il server deve essere pubblica.

```
mod server {
    pub fn avvia() {
        println!("Server avviato.");
        gestisci_richiesta();
    }

    fn gestisci_richiesta() {
        println!("Richiesta gestita.");
    }
}

fn main() {
    server::avvia();
}
```

La funzione gestisci_richiesta è privata e può essere chiamata solo all'interno del modulo server.

La funzione avvia è pubblica e accessibile dall'esterno.

4) Scrivere un programma che mostri come Rust organizza i moduli utilizzando file separati. Creare una directory moduli, con due file: moduli/matematica.rs e moduli/stringhe.rs. Importare e utilizzare le funzioni definite in questi moduli.

Struttura del progetto:

```
src/
 └── main.rs
 └── moduli/
```

```
    ├── matematica.rs
    └── stringhe.rs
```

main.rs:

```
mod moduli;

fn main() {
    let risultato = moduli::matematica::somma(5, 7);
    println!("Somma: {}", risultato);

    let stringa = moduli::stringhe::concatena("Rust", "Lang");
    println!("Stringa concatenata: {}", stringa);
}
```

moduli/mod.rs:

```
pub mod matematica;
pub mod stringhe;
```

moduli/matematica.rs:

```
pub fn somma(a: i32, b: i32) -> i32 {
    a + b
}
```

moduli/stringhe.rs:

```
pub fn concatena(s1: &str, s2: &str) -> String {
    format!("{}{}", s1, s2)
```

```
}
```

Spiegazione:

Ogni modulo è organizzato in file separati e referenziato in mod.rs.

Le funzioni pubbliche dei moduli possono essere utilizzate nel file main.rs.

5) Creare un programma che dimostri l'uso dei crate esterni. Aggiungere il crate rand per generare un numero casuale. Definire un modulo giochi con una funzione che genera un numero casuale tra 1 e 10 e lo restituisce.

Cargo.toml:

```
 [dependencies]
rand = "0.8"
```

Codice:

```
extern crate rand;
use rand::Rng;

mod giochi {
    use rand::Rng;

    pub fn numero_casuale() -> i32 {
        let mut rng = rand::thread_rng();
        rng.gen_range(1..=10)
    }
}
```

```
fn main() {
    let numero = giochi::numero_casuale();
    println!("Numero casuale generato: {}", numero);
}
```

Spiegazione:

extern crate e use rand::Rng permettono l'uso del crate esterno rand.

Il modulo giochi contiene una funzione che genera un numero casuale.

6) Creare un modulo che simula una semplice banca. Il modulo banca deve contenere una struct Conto con metodi per depositare e prelevare denaro. Rendere pubblici solo i metodi di deposito e prelievo.

```
mod banca {
    pub struct Conto {
        saldo: f64,
    }

    impl Conto {
        pub fn nuovo() -> Conto {
            Conto { saldo: 0.0 }
        }

        pub fn deposita(&mut self, importo: f64) {
            self.saldo += importo;
        }

        pub fn preleva(&mut self, importo: f64) -> Result<(), &'static str> {
            if self.saldo >= importo {
                self.saldo -= importo;
```

```
                Ok(())
            } else {
                Err("Saldo insufficiente")
            }
        }

        pub fn saldo(&self) -> f64 {
            self.saldo
        }
    }
}

fn main() {
    let mut mio_conto = banca::Conto::nuovo();
    mio_conto.deposita(100.0);
    mio_conto.preleva(30.0).unwrap();
    println!("Saldo corrente: {}", mio_conto.saldo());
}
```

Il modulo banca contiene una struct Conto con metodi pubblici per la gestione del saldo.

Il campo saldo è privato e può essere modificato solo tramite i metodi pubblici.

7) Scrivere un programma che dimostri la visibilità delle funzioni private nei sotto-moduli. Definire un modulo cliente con un sotto-modulo dati contenente una funzione privata. Verificare che la funzione sia accessibile solo all'interno del modulo cliente.

```
mod cliente {
    pub mod dati {
        pub fn mostra_dati() {
            println!("Mostra dati cliente.");
            dati_privati();
```

```
        }

        fn dati_privati() {
            println!("Dati privati del cliente.");
        }
    }
}

fn main() {
    cliente::dati::mostra_dati();
}
```

La funzione dati_privati è accessibile solo all'interno del modulo dati, e non può essere chiamata da main.

8) Creare un modulo utente con una struct Profilo che abbia campi pubblici e privati. Definire un metodo pubblico per visualizzare i campi privati solo se l'utente è autenticato.

```
mod utente {
    pub struct Profilo {
        pub nome: String,
        eta: u32,
    }

    impl Profilo {
        pub fn nuovo(nome: String, eta: u32) -> Profilo {
            Profilo { nome, eta }
        }

        pub fn mostra_eta(&self, autenticato: bool) {
            if autenticato {
```

```
                println!("Età: {}", self.eta);
            } else {
                println!("Accesso negato.");
            }
        }
    }
}

fn main() {
    let profilo = utente::Profilo::nuovo("Mario".to_string(), 30);
    println!("Nome: {}", profilo.nome);
    profilo.mostra_eta(true);
}
```

Il campo eta è privato e può essere visualizzato solo tramite il metodo mostra_eta se l'utente è autenticato.

9) Scrivere un programma che dimostri l'uso di pub(crate) per limitare la visibilità di una funzione a tutto il crate, ma non all'esterno.

```
mod rete {
    pub(crate) fn connetti() {
        println!("Connessione al server in corso...");
    }
}

fn main() {
    rete::connetti();
}
```

La funzione connetti è visibile in tutto il crate, ma non sarà visibile se questo modulo fosse incluso in un altro crate.

10) Creare un programma che organizza il codice in moduli per simulare una semplice applicazione di e-commerce. Definire moduli per prodotti, ordini, e clienti, con alcune funzioni e campi pubblici e privati.

```
mod ecommerce {
    pub mod prodotti {
        pub fn lista_prodotti() {
            println!("Lista dei prodotti disponibili.");
        }

        pub fn aggiungi_prodotto(nome: &str) {
            println!("Prodotto {} aggiunto.", nome);
        }
    }

    pub mod ordini {
        pub fn crea_ordine() {
            println!("Ordine creato.");
        }
    }

    pub mod clienti {
        pub fn nuovo_cliente(nome: &str) {
            println!("Cliente {} aggiunto.", nome);
        }
    }
}

fn main() {
```

```
    ecommerce::prodotti::lista_prodotti();
    ecommerce::prodotti::aggiungi_prodotto("Laptop");
    ecommerce::ordini::crea_ordine();
    ecommerce::clienti::nuovo_cliente("Luigi");
}
```

Il codice è organizzato in moduli distinti per gestire prodotti, ordini e clienti.

11) Scrivere un programma che separa la logica in un crate di libreria e un crate binario. Il crate di libreria deve contenere una funzione per sommare due numeri, e il crate binario deve richiamare questa funzione.

Struttura del progetto:

```
my_project/
 └── Cargo.toml
 └── src/
     ├── main.rs
     └── lib.rs
```

Cargo.toml:

```
 [package]
name = "my_project"
version = "0.1.0"
edition = "2021"

[dependencies]
```

lib.rs:

```
pub fn somma(a: i32, b: i32) -> i32 {
    a + b
}
```

main.rs:

```
use my_project::somma;

fn main() {
    let risultato = somma(4, 5);
    println!("La somma è: {}", risultato);
}
```

Spiegazione:

lib.rs contiene la logica di una libreria con la funzione somma.

main.rs rappresenta il crate binario che utilizza la libreria.

12) Definire una libreria che contiene una funzione per verificare se un numero è pari o dispari. Scrivere un test per questa funzione.

Struttura del progetto:

```
my_library/
 └── Cargo.toml
 └── src/
     └── lib.rs
```

lib.rs:

```
pub fn is_pari(numero: i32) -> bool {
    numero % 2 == 0
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_is_pari() {
        assert!(is_pari(2));
        assert!(!is_pari(3));
    }
}
```

Cargo.toml:

```
 [package]
name = "my_library"
version = "0.1.0"
edition = "2021"
```

[dependencies]

Spiegazione:

La funzione is_pari è dichiarata nel crate di libreria.

La sezione #[cfg(test)] contiene test unitari.

13) Utilizzare il crate rand per generare un numero casuale all'interno di un range e visualizzarlo nel main.

Cargo.toml:

```
 [package]
name = "random_number"
version = "0.1.0"
edition = "2021"

[dependencies]
rand = "0.8"
```

main.rs:

```
use rand::Rng;

fn main() {
    let mut rng = rand::thread_rng();
    let numero_casuale: i32 = rng.gen_range(1..=100);
    println!("Numero casuale: {}", numero_casuale);
}
```

Spiegazione:

rand è specificato come dipendenza nel file Cargo.toml.

La funzione gen_range genera un numero casuale all'interno di un intervallo specificato.

14) Creare una libreria che utilizza la dipendenza serde solo se specificata come feature opzionale in Cargo.toml.

Cargo.toml:

```
 [package]
name = "optional_feature"
version = "0.1.0"
edition = "2021"

[dependencies]
serde = { version = "1.0", optional = true }

[features]
default = []
with_serde = ["serde"]
```

lib.rs:

```
#[cfg(feature = "with_serde")]
use serde::{Serialize, Deserialize};

#[cfg(feature = "with_serde")]
#[derive(Serialize, Deserialize)]
pub struct Utente {
    pub nome: String,
    pub eta: u32,
}

#[cfg(not(feature = "with_serde"))]
pub struct Utente {
    pub nome: String,
    pub eta: u32,
```

```
}
```

Spiegazione:

La dipendenza serde è opzionale e attivabile tramite la feature with_serde.

Il crate può essere compilato senza serde, ma se viene abilitata la feature, si utilizza la serializzazione.

15) Modificare il file Cargo.toml per indicare che si desidera utilizzare una versione specifica di un crate che segue il versionamento semantico.

Cargo.toml:

```
 [package]
name = "version_management"
version = "0.1.0"
edition = "2021"

[dependencies]
rand = "0.8.3"
```

Spiegazione:

Il numero di versione rand = "0.8.3" indica che verranno utilizzate versioni compatibili con 0.8.x, secondo il versionamento semantico.

16) Creare un workspace con due crate distinti: uno binario e un crate di libreria.

Struttura del progetto:

```
my_workspace/
 ├── Cargo.toml
 ├── bin_crate/
 │    └── Cargo.toml
 │    └── src/main.rs
 └── lib_crate/
      └── Cargo.toml
      └── src/lib.rs
```

Cargo.toml (workspace):

```
 [workspace]
members = ["bin_crate", "lib_crate"]
```

bin_crate/Cargo.toml:

```
 [package]
name = "bin_crate"
version = "0.1.0"
edition = "2021"

[dependencies]
lib_crate = { path = "../lib_crate" }
```

lib_crate/lib.rs:

```
pub fn saluta() -> &'static str {
    "Ciao dal crate di libreria!"
}
```

bin_crate/main.rs:

```
use lib_crate::saluta;

fn main() {
    println!("{}", saluta());
}
```

Spiegazione:

Il workspace consente di gestire più crate come parte di un unico progetto.

Il crate binario dipende dal crate di libreria, definito nel workspace.

17) Creare un crate, configurare correttamente il file Cargo.toml per la pubblicazione su crates.io, e simulare la procedura di pubblicazione.

Cargo.toml:

```
 [package]
name = "crate_pubblicato"
version = "0.1.0"
edition = "2021"
description = "Un semplice crate di esempio"
repository = "https://github.com/tuo_utente/repo"
license = "MIT"
documentation = "https://docs.rs/crate_pubblicato"
homepage = "https://tuo_sito.com"

[dependencies]
```

Spiegazione:

Per pubblicare il crate, è necessario configurare correttamente il file Cargo.toml con informazioni come il repository e la licenza.

Dovremo poi servirci del comando *cargo publish* per pubblicare il crate su crates.io.

18) Modificare un progetto in modo che possa essere compilato sia come binario che come libreria.

Struttura del progetto:

```
my_crate/
 └── Cargo.toml
 └── src/
     ├── main.rs
     └── lib.rs
```

Cargo.toml:

```
 [package]
name = "my_crate"
version = "0.1.0"
edition = "2021"

[dependencies]
```

lib.rs:

```
pub fn funzione_lib() {
    println!("Funzione chiamata dalla libreria");
}
```

main.rs:

```
use my_crate::funzione_lib;

fn main() {
    funzione_lib();
}
```

Spiegazione:

Il crate può essere compilato sia come binario (con main.rs) che come libreria (con lib.rs).

Il codice in lib.rs può essere usato sia dal crate binario che da altri crate esterni.

19) Creare un crate che dipende da un altro crate locale. Configurare Cargo.toml per indicare la dipendenza.

Cargo.toml (crate principale):

```
 [package]
name = "main_crate"
version = "0.1.0"
edition = "2021"

[dependencies]
lib_crate = { path = "../lib_crate" }
```

Spiegazione:

La dipendenza lib_crate è inclusa indicando il percorso locale del crate.

20) Modificare il file Cargo.toml per includere una dipendenza a una versione di pre-release di un crate.

Cargo.toml:

```
 [package]
name = "pre_release_example"
version = "0.1.0"
edition = "2021"

[dependencies]
serde = "1.0.130-beta"
```

Spiegazione:

Il numero di versione 1.0.130-beta indica una dipendenza su una versione di pre-release del crate serde.

21) Creare un crate di libreria che definisca una macro personalizzata, utilizzando un altro crate per una procedural macro che genera codice in fase di compilazione. Lo deve fare con una struttura con metodi di accesso per ogni campo definito.

Struttura del progetto:

```
macro_crate/
 ├── Cargo.toml
 └── src/
     └── lib.rs
generated_struct/
 ├── Cargo.toml
```

```
└── src/
    └── main.rs
```

Cargo.toml (macro_crate):

```
[package]
name = "macro_crate"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true

[dependencies]
syn = "1.0"
quote = "1.0"
```

lib.rs (macro_crate):

```
extern crate proc_macro;
use proc_macro::TokenStream;
use syn::{parse_macro_input, DeriveInput};
use quote::quote;

#[proc_macro_derive(GeneraAccessor)]
pub fn genera_accessor(input: TokenStream) -> TokenStream {
    let ast = parse_macro_input!(input as DeriveInput);
    let nome_struttura = &ast.ident;

    let mut getter_methods = Vec::new();
```

```
    if let syn::Data::Struct(data_struct) = &ast.data {
        for campo in data_struct.fields.iter() {
            let nome_campo = &campo.ident;
            let tipo_campo = &campo.ty;
            getter_methods.push(quote! {
                pub fn #nome_campo(&self) -> &#tipo_campo {
                    &self.#nome_campo
                }
            });
        }
    }

    let expanded = quote! {
        impl #nome_struttura {
            #(#getter_methods)*
        }
    };

    TokenStream::from(expanded)
}
```

Cargo.toml (generated_struct):

```
 [package]
name = "generated_struct"
version = "0.1.0"
edition = "2021"

[dependencies]
macro_crate = { path = "../macro_crate" }
```

main.rs (generated_struct):

```
use macro_crate::GeneraAccessor;

#[derive(GeneraAccessor)]
struct Persona {
    nome: String,
    eta: u32,
}

fn main() {
    let persona = Persona { nome: "Mario".to_string(), eta: 30 };
    println!("Nome: {}", persona.nome());
    println!("Età: {}", persona.eta());
}
```

Spiegazione:

Questo esempio utilizza un crate procedural macro per generare automaticamente metodi di accesso ai campi di una struttura in fase di compilazione.

Il crate macro_crate implementa la logica di generazione del codice tramite le macro.

22) Creare un crate di libreria che implementi una cache LRU, in cui la chiave meno recentemente usata viene rimossa quando la cache raggiunge la capacità massima. L'implementazione deve essere generica e supportare riferimenti con lifetimes.

Struttura del progetto:

```
lru_cache/
  └── Cargo.toml
```

```
└── src/
    └── lib.rs
```

Cargo.toml:

```
[package]
name = "lru_cache"
version = "0.1.0"
edition = "2021"

[dependencies]
```

lib.rs:

```
use std::collections::HashMap;
use std::collections::VecDeque;

pub struct LRUCache<K, V> {
    capacity: usize,
    map: HashMap<K, V>,
    queue: VecDeque<K>,
}

impl<K: std::cmp::Eq + std::hash::Hash + Clone, V> LRUCache<K, V> {
    pub fn new(capacity: usize) -> Self {
        LRUCache {
            capacity,
            map: HashMap::new(),
            queue: VecDeque::new(),
        }
    }
```

```
    pub fn get(&mut self, key: &K) -> Option<&V> {
        if let Some(value) = self.map.get(key) {
            self.queue.retain(|x| x != key);
            self.queue.push_front(key.clone());
            return Some(value);
        }
        None
    }

    pub fn put(&mut self, key: K, value: V) {
        if self.map.len() >= self.capacity {
            if let Some(lru) = self.queue.pop_back() {
                self.map.remove(&lru);
            }
        }
        self.queue.push_front(key.clone());
        self.map.insert(key, value);
    }
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_lru_cache() {
        let mut cache = LRUCache::new(2);
        cache.put("a", 1);
        cache.put("b", 2);
        assert_eq!(cache.get(&"a"), Some(&1));
```

```
        cache.put("c", 3);
        assert_eq!(cache.get(&"b"), None);
        assert_eq!(cache.get(&"c"), Some(&3));
    }
}
```

Spiegazione:

Questo esercizio introduce una gestione avanzata della memoria tramite borrowing, lifetimes e l'uso di una struttura per gestire una cache LRU.

L'implementazione gestisce la rimozione degli elementi meno utilizzati quando la cache raggiunge la capacità massima.

23) Creare una libreria che gestisca le connessioni TCP in maniera sicura usando Arc<Mutex<>>, garantendo che più thread possano accedere alla connessione senza violare la mutabilità.

Struttura del progetto:

```
tcp_lib/
 └── Cargo.toml
 └── src/
     └── lib.rs
```

Cargo.toml:

```
 [package]
name = "tcp_lib"
version = "0.1.0"
edition = "2021"
```

```
[dependencies]
tokio = { version = "1", features = ["full"] }
```

lib.rs:

```
use std::sync::{Arc, Mutex};
use tokio::net::TcpStream;
use tokio::io::{AsyncReadExt, AsyncWriteExt};

pub struct ConnessioneTCP {
    stream: Arc<Mutex<TcpStream>>,
}

impl ConnessioneTCP {
    pub async fn new(addr: &str) -> tokio::io::Result<Self> {
        let stream = TcpStream::connect(addr).await?;
        Ok(ConnessioneTCP {
            stream: Arc::new(Mutex::new(stream)),
        })
    }

    pub async fn invia(&self, msg: &str) -> tokio::io::Result<()> {
        let mut stream = self.stream.lock().unwrap();
        stream.write_all(msg.as_bytes()).await?;
        Ok(())
    }

    pub async fn ricevi(&self) -> tokio::io::Result<String> {
        let mut buffer = vec![0; 1024];
        let mut stream = self.stream.lock().unwrap();
        let n = stream.read(&mut buffer).await?;
```

```
        Ok(String::from_utf8_lossy(&buffer[..n]).to_string())
    }
}
```

Spiegazione:

Arc<Mutex<>> è usato per consentire l'accesso sicuro e concorrente a una connessione TCP tra più thread.

Viene utilizzato il crate tokio per la gestione asincrona delle operazioni di I/O.

24) Creare un crate di libreria che utilizzi il crate regex per analizzare e filtrare i dati provenienti da un file di log. La funzione deve estrarrne tutti gli indirizzi IP e restituirli in un vettore.

Struttura del progetto:

```
log_parser/
 └── Cargo.toml
 └── src/
     └── lib.rs
```

Cargo.toml:

```
 [package]
name = "log_parser"
version = "0.1.0"
edition = "2021"

[dependencies]
regex = "1"
```

lib.rs:

```
use regex::Regex;

pub fn estrai_ips(log: &str) -> Vec<String> {
    let re = Regex::new(r"(\d{1,3}\.){3}\d{1,3}").unwrap();
    re.find_iter(log).map(|mat| mat.as_str().to_string()).collect()
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn test_estrai_ips() {
        let log = "Accesso da 192.168.1.1 e poi da 10.0.0.2.";
        let ips = estrai_ips(log);
        assert_eq!(ips, vec!["192.168.1.1", "10.0.0.2"]);
    }
}
```

Spiegazione:

Questo esercizio introduce l'uso del crate regex per il parsing di stringhe complesse (in questo caso, indirizzi IP da un file di log).

L'implementazione utilizza espressioni regolari per estrarre e restituire i risultati.

25) Creare un crate binario che esponga un'API REST utilizzando warp, gestendo più versioni dell'API (v1 e v2), ognuna con metodi e risposte differenti. Devono essere gestiti sia i metodi GET che POST.

Struttura del progetto:

```
rest_service/
 └── Cargo.toml
 └── src/
     └── main.rs
```

Cargo.toml:

```
 [package]
name = "rest_service"
version = "0.1.0"
edition = "2021"

[dependencies]
warp = "0.3"
tokio = { version = "1", features = ["full"] }
serde = { version = "1", features = ["derive"] }
serde_json = "1"
```

main.rs:

```
use warp::Filter;
use serde::{Serialize, Deserialize};

#[derive(Serialize, Deserialize)]
struct RichiestaV2 {
    nome: String,
    eta: u32,
}
```

```
#[tokio::main]

async fn main() {

    let api_v1 = warp::path("v1")

        .and(warp::path("saluto"))

        .and(warp::get())

        .map(|| warp::reply::json(&"Benvenuto alla versione 1"));


    let api_v2 = warp::path("v2")

        .and(warp::path("saluto"))

        .and(warp::post())

        .and(warp::body::json())

        .map(|richiesta: RichiestaV2| {

            let risposta = format!("Ciao, {}. Hai {} anni.", richiesta.nome,
richiesta.eta);

            warp::reply::json(&risposta)

        });


    let routes = api_v1.or(api_v2);


    warp::serve(routes).run(([127, 0, 0, 1], 3030)).await;

}
```

Spiegazione:

Utilizza il crate warp per gestire un servizio REST con versionamento delle API.

La versione 1 (v1) gestisce un semplice GET, mentre la versione 2 (v2) gestisce un POST con parsing del body in formato JSON.

L'uso di crate asincroni come tokio e warp consente di gestire richieste HTTP in modo efficiente.

Capitolo 8

1) Spiega la differenza tra concorrenza e parallelismo e scrivi un programma che esegua due operazioni in parallelo su thread distinti.

```
use std::thread;


fn main() {
    let handle1 = thread::spawn(|| {
        for i in 1..5 {
            println!("Thread 1: {}", i);
        }
    });

    let handle2 = thread::spawn(|| {
        for i in 1..5 {
            println!("Thread 2: {}", i);
        }
    });

    handle1.join().unwrap();
    handle2.join().unwrap();
}
```

Concorrenza si riferisce alla gestione di più operazioni apparentemente simultanee su una singola CPU, mentre il parallelismo implica l'esecuzione effettiva su più core. Rust gestisce il parallelismo tramite il crate std::thread.

2) Scrivi un programma che utilizzi Mutex per condividere una variabile tra thread in modo sicuro.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Contatore: {}", *counter.lock().unwrap());
}
```

Il Mutex garantisce che solo un thread alla volta possa accedere alla variabile condivisa. Arc (Atomic Reference Counting) viene utilizzato per consentire la condivisione sicura di dati tra thread.

3) Modifica un programma che incrementa un contatore su più thread e dimostra come un Mutex prevenga le race condition.

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            for _ in 0..1000 {
                let mut num = counter.lock().unwrap();
                *num += 1;
            }
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Valore finale: {}", *counter.lock().unwrap());
}
```

Una race condition si verifica quando più thread accedono contemporaneamente a una risorsa condivisa. Mutex impedisce l'accesso concorrente non sicuro.

4) Gestisci due variabili condivise tramite Mutex e dimostra come evitare un deadlock.

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let a = Arc::new(Mutex::new(1));
    let b = Arc::new(Mutex::new(2));

    let a1 = Arc::clone(&a);
    let b1 = Arc::clone(&b);

    let handle1 = thread::spawn(move || {
        let mut a_lock = a1.lock().unwrap();
        thread::sleep(std::time::Duration::from_millis(50));
        let _b_lock = b1.lock().unwrap();
        *a_lock += 1;
    });

    let a2 = Arc::clone(&a);
    let b2 = Arc::clone(&b);

    let handle2 = thread::spawn(move || {
        let mut b_lock = b2.lock().unwrap();
        thread::sleep(std::time::Duration::from_millis(50));
        let _a_lock = a2.lock().unwrap();
        *b_lock += 1;
    });

    handle1.join().unwrap();
    handle2.join().unwrap();

    println!("a: {}, b: {}", *a.lock().unwrap(), *b.lock().unwrap());
```

```
}
```

Un deadlock si verifica quando due thread si bloccano a vicenda cercando di ottenere il lock sulle stesse risorse. In questo caso, l'utilizzo attento di Mutex e una corretta gestione della sequenza di lock prevengono il deadlock.

5) Utilizza RwLock per consentire accessi concorrenti immutabili a una risorsa condivisa.

```
use std::sync::{Arc, RwLock};
use std::thread;

fn main() {
    let data = Arc::new(RwLock::new(5));
    let mut handles = vec![];

    for _ in 0..10 {
        let data = Arc::clone(&data);
        let handle = thread::spawn(move || {
            let read_data = data.read().unwrap();
            println!("Lettura: {}", *read_data);
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
}
```

RwLock permette a più thread di accedere a una risorsa immutabilmente in parallelo, ma consente solo a un thread di scrivere alla volta.

6) Utilizza Arc e Cell per gestire dati condivisi tra thread con mutabilità interna.

```
use std::cell::Cell;
use std::sync::Arc;
use std::thread;

fn main() {
    let data = Arc::new(Cell::new(5));
    let mut handles = vec![];

    for _ in 0..10 {
        let data = Arc::clone(&data);
        let handle = thread::spawn(move || {
            data.set(data.get() + 1);
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Valore finale: {}", data.get());
}
```

Cell consente la mutabilità interna, permettendo la modifica del valore anche se il puntatore che lo contiene è immutabile.

7) Utilizza Arc e Mutex per garantire la sicurezza nell'accesso a dati condivisi tra thread.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let data = Arc::new(Mutex::new(5));
    let mut handles = vec![];

    for _ in 0..10 {
        let data = Arc::clone(&data);
        let handle = thread::spawn(move || {
            let mut num = data.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Valore finale: {}", *data.lock().unwrap());
}
```

Utilizzando Arc e Mutex, possiamo condividere e modificare in sicurezza un valore tra più thread.

8) Simula un server concorrente con accesso sicuro a dati condivisi tra thread utilizzando Arc, Mutex e operazioni atomiche.

```
use std::sync::{Arc, Mutex};
use std::thread;
```

```
use std::sync::atomic::{AtomicUsize, Ordering};

fn main() {
    let counter = Arc::new(AtomicUsize::new(0));
    let data = Arc::new(Mutex::new(vec![]));
    let mut handles = vec![];

    for i in 0..10 {
        let counter = Arc::clone(&counter);
        let data = Arc::clone(&data);
        let handle = thread::spawn(move || {
            counter.fetch_add(1, Ordering::SeqCst);
            let mut data = data.lock().unwrap();
            data.push(i);
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Contatore: {}", counter.load(Ordering::SeqCst));
    println!("Dati: {:?}", *data.lock().unwrap());
}
```

Questo esempio mostra l'utilizzo di operazioni atomiche tramite AtomicUsize insieme a Arc e Mutex per la gestione concorrente di un contatore e di dati condivisi.

9) Gestisci correttamente il deadlock su più risorse utilizzando Mutex, assicurando che i thread acquisiscano i lock in un ordine coerente.

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let resource1 = Arc::new(Mutex::new(1));
    let resource2 = Arc::new(Mutex::new(2));

    let r1 = Arc::clone(&resource1);
    let r2 = Arc::clone(&resource2);

    let handle1 = thread::spawn(move || {
        let _lock1 = r1.lock().unwrap();
        println!("Thread 1: acquisito il lock su resource1");
        thread::sleep(std::time::Duration::from_millis(50));
        let _lock2 = r2.lock().unwrap();
        println!("Thread 1: acquisito il lock su resource2");
    });

    let r1 = Arc::clone(&resource1);
    let r2 = Arc::clone(&resource2);

    let handle2 = thread::spawn(move || {
        let _lock1 = r1.lock().unwrap();
        println!("Thread 2: acquisito il lock su resource1");
        thread::sleep(std::time::Duration::from_millis(50));
        let _lock2 = r2.lock().unwrap();
        println!("Thread 2: acquisito il lock su resource2");
    });

    handle1.join().unwrap();
```

```
    handle2.join().unwrap();
}
```

Per prevenire il deadlock, i thread devono acquisire i lock nell'ordine corretto. In questo esempio, entrambi i thread acquisiscono prima il lock sulla resource1 e poi su resource2. Questo approccio ordinato impedisce il deadlock.

10) Scrivi un programma che implementi una coda concorrente (queue) utilizzando Arc, Mutex e Condvar per sincronizzare produttori e consumatori su più thread.

```
use std::sync::{Arc, Mutex, Condvar};
use std::thread;

struct Queue<T> {
    data: Mutex<Vec<T>>,
    available: Condvar,
}

impl<T> Queue<T> {
    fn new() -> Arc<Self> {
        Arc::new(Queue {
            data: Mutex::new(Vec::new()),
            available: Condvar::new(),
        })
    }

    fn push(&self, item: T) {
        let mut data = self.data.lock().unwrap();
        data.push(item);
        self.available.notify_one();
    }
```

```
    fn pop(&self) -> T {
        let mut data = self.data.lock().unwrap();
        while data.is_empty() {
            data = self.available.wait(data).unwrap();
        }
        data.remove(0)
    }
}

fn main() {
    let queue = Queue::new();
    let producer_queue = Arc::clone(&queue);

    let producer = thread::spawn(move || {
        for i in 0..5 {
            println!("Produttore: inserisce {}", i);
            producer_queue.push(i);
            thread::sleep(std::time::Duration::from_millis(100));
        }
    });

    let consumer_queue = Arc::clone(&queue);

    let consumer = thread::spawn(move || {
        for _ in 0..5 {
            let item = consumer_queue.pop();
            println!("Consumatore: rimosso {}", item);
        }
    });
```

```
    producer.join().unwrap();
    consumer.join().unwrap();
}
```

In questo esempio, abbiamo una coda concorrente che utilizza Mutex per proteggere l'accesso ai dati, e Condvar per sincronizzare i thread, assicurando che i consumatori attendano che ci siano elementi disponibili nella coda. Condvar permette la sincronizzazione dei thread su una condizione condivisa.

Condvar, o "variabile di condizione", è uno strumento utile per la sincronizzazione tra thread. Consente a un thread di "mettersi in attesa" fino a quando non viene notificato da un altro thread, ed è spesso utilizzata in combinazione con i mutex per gestire l'accesso a risorse condivise.
Il suo funzionamento può essere compreso meglio attraverso un esempio pratico. Immagina di avere una situazione in cui un thread deve aspettare che un'altra operazione sia completata prima di procedere. Per fare ciò, un thread può entrare in attesa su una variabile di condizione. Questo avviene chiamando il metodo *wait()* della *Condvar,* che si occupa di rilasciare il *mutex* associato e mettere il thread in uno stato di attesa. In questo modo, il thread non consuma risorse CPU mentre aspetta. Quando l'operazione che il thread stava aspettando è completata, un altro thread può invocare *notify_one()* o *notify_all()* sulla stessa Condvar. *notify_one()* sveglia un singolo thread in attesa, mentre *notify_all()* sveglia tutti i thread in attesa. Quando il thread viene risvegliato, *wait()* restituisce il controllo al thread risvegliato, che riacquista il mutex e può quindi procedere con l'operazione successiva.
È fondamentale notare che, per evitare condizioni di gara, il mutex deve sempre essere utilizzato insieme alla variabile di condizione. Ad esempio, il thread che chiama *wait()* deve prima acquisire il *mutex* e poi chiamare *wait()* in modo che il *mutex* venga rilasciato durante l'attesa, permettendo ad altri thread di accedere alla risorsa condivisa. Quando il thread viene risvegliato, riacquista il *mutex* prima di continuare.
In questo modo, *Condvar* diventa un potente meccanismo per coordinare l'esecuzione tra diversi thread, permettendo a un thread di aspettare una condizione specifica senza occupare inutilmente risorse, e garantendo al contempo la sicurezza dei dati attraverso il mutex. Questo approccio è particolarmente utile in scenari di produttore-consumatore, dove un produttore deve attendere che ci sia spazio nel buffer prima di produrre, e un consumatore deve attendere che ci siano elementi da consumare.

11) Scrivi un programma che crea un thread che invia 10 messaggi numerici a un canale. Il thread principale deve ricevere e stampare questi messaggi.

```
use std::thread;
use std::sync::mpsc;
```

```
fn main() {
    // Crea un canale
    let (tx, rx) = mpsc::channel();

    // Avvia un nuovo thread
    thread::spawn(move || {
        for i in 0..10 {
            // Invia messaggi
            tx.send(i).unwrap();
        }
    });

    // Ricevi e stampa i messaggi nel thread principale
    for _ in 0..10 {
        let msg = rx.recv().unwrap();
        println!("Ricevuto: {}", msg);
    }
}
```

12) Crea un thread per inviare cinque stringhe a un canale. Il thread principale deve ricevere e stampare queste stringhe.

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let messages = vec!["Ciao", "come", "va?", "Spero", "bene!"];
```

```
        for msg in messages {
            tx.send(msg).unwrap();
        }
    });

    for _ in 0..5 {
        let msg = rx.recv().unwrap();
        println!("Ricevuto: {}", msg);
    }
}
```

13) utilizza il pattern produttore-consumatore, dove un thread produce numeri e un altro li consuma.

```
use std::thread;
use std::sync::mpsc;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    // Thread produttore
    thread::spawn(move || {
        for i in 0..5 {
            tx.send(i).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    // Thread consumatore
    thread::spawn(move || {
        for _ in 0..5 {
```

```
            let num = rx.recv().unwrap();
            println!("Consumando: {}", num);
        }
    });

    // Attendi la terminazione del thread consumatore
    thread::sleep(Duration::from_secs(6));
}
```

14) Scrivi un programma che crea un thread che invia messaggi numerati (come "Messaggio 0", "Messaggio 1", ecc.) a un canale. Il thread principale deve ricevere e stampare questi messaggi.

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        for i in 0..5 {
            let msg = format!("Messaggio {}", i);
            tx.send(msg).unwrap();
        }
    });

    for _ in 0..5 {
        let msg = rx.recv().unwrap();
        println!("Ricevuto: {}", msg);
    }
}
```

15) Scrivi un programma che crea tre thread, ognuno dei quali invia un messaggio al thread principale, che dovrà stampare tutti i messaggi ricevuti.

```
use std::thread;
use std::sync::mpsc;

fn main() {
    let (tx, rx) = mpsc::channel();

    for i in 0..3 {
        let tx = tx.clone(); // Clona il trasmettitore per ciascun thread
        thread::spawn(move || {
            let msg = format!("Messaggio da thread {}", i);
            tx.send(msg).unwrap();
        });
    }

    // Stampa i messaggi ricevuti
    for _ in 0..3 {
        let msg = rx.recv().unwrap();
        println!("Ricevuto: {}", msg);
    }
}
```