Rust Riferimento Rapido

Ownership, tipi, trait, pattern matching essenziali

Basi

Hello World

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

Comandi Cargo

```
cargo new my_project    # crea nuovo progetto
cargo build             # compila (debug)
cargo build --release   # compila (ottimizzato)
cargo run               # compila ed esegui
cargo test              # esegui i test
```

Struttura del Progetto

`Cargo.toml`	Manifesto del progetto (dipendenze, metadati)
`src/main.rs`	Punto di ingresso del crate binario
`src/lib.rs`	Radice del crate libreria
`tests/`	Directory dei test di integrazione

Variabili e Mutabilità

Binding e Mutabilità

```
let x = 5;              // immutabile per default
let mut y = 10;         // mutabile
y += 1;
const MAX: u32 = 100;   // costante a tempo di compilazione
```

Shadowing

```
let x = 5;
let x = x + 1;          // oscura il precedente x
let x = "ora una stringa"; // può cambiare tipo
```

Tipi Scalari

`i8..i128, isize`	Interi con segno
`u8..u128, usize`	Interi senza segno
`f32, f64`	Virgola mobile (f64 default)
`bool`	**`true`** / **`false`**
`char`	Scalare Unicode (4 byte)

Tipi Composti

```
let tup: (i32, f64, char) = (42, 6.4, 'z');
let (a, b, c) = tup;        // destrutturazione
let arr: [i32; 3] = [1, 2, 3];
let first = arr[0];
```

Funzioni

Definizione

```
fn add(a: i32, b: i32) -> i32 {
    a + b   // senza punto e virgola = espressione di ritorno
}
```

Closure

```
let double = |x: i32| x * 2;
let sum: i32 = vec![1, 2, 3]
    .iter()
    .map(|x| x * 2)
    .sum();
```

Puntatori a Funzione e Trait

`fn(T) -> U`	Tipo puntatore a funzione
`Fn(T) -> U`	Closure che prende in prestito
`FnMut(T) -> U`	Closure che prende in prestito mutabilmente
`FnOnce(T) -> U`	Closure che prende la ownership

Flusso di Controllo

If / Else

```
let status = if score >= 90 { "A" }
             else if score >= 80 { "B" }
             else { "C" };  // if è un'espressione
```

Cicli

```
loop { break; }                 // infinito
while condition { }             // while
for item in &vec { }            // iteratore
for i in 0..10 { }             // range
for (i, v) in vec.iter().enumerate() { }
```

Etichette di Ciclo

```
'outer: for i in 0..5 {
    for j in 0..5 {
        if i + j > 6 { break 'outer; }
    }
}
```

Ownership e Borrowing

Regole dell'Ownership

1. Ogni valore ha esattamente un proprietario.
2. Quando il proprietario esce dallo scope, il valore viene rilasciato.
3. I valori possono essere spostati o clonati.

Move e Clone

```
let s1 = String::from("hello");
let s2 = s1;          // s1 è spostato, non più valido
let s3 = s2.clone();  // copia profonda, entrambi validi
```

Borrowing

```
fn len(s: &String) -> usize { s.len() }  // riferimento condiviso
fn push(s: &mut String) { s.push('!'); }  // riferimento mutabile
// Regola: molti &T OPPURE un &mut T, mai entrambi
```

Lifetime

```
fn longest<'a>(a: &'a str, b: &'a str) -> &'a str {
    if a.len() > b.len() { a } else { b }
}
```

Struct ed Enum

Struct

```
struct User {
    name: String,
    age: u32,
    active: bool,
}
let u = User { name: String::from("Alice"), age: 30, active: true };
```

Blocco Impl

```
impl User {
    fn new(name: &str, age: u32) -> Self {
        Self { name: name.to_string(), age, active: true }
    }
    fn greeting(&self) -> String {
        format!("Hi, {}", self.name)
    }
}
```

Enum

```
enum Shape {
    Circle(f64),
    Rect { w: f64, h: f64 },
    Point,
}
let s = Shape::Circle(5.0);
```

Pattern Matching

Espressione Match

```
match shape {
    Shape::Circle(r) => std::f64::consts::PI * r * r,
    Shape::Rect { w, h } => w * h,
    Shape::Point => 0.0,
}
```

If Let e While Let

```
if let Some(val) = optional {
    println!("{val}");
}
while let Some(top) = stack.pop() {
    println!("{top}");
}
```

Sintassi dei Pattern

`_`	Wildcard, corrisponde a qualsiasi cosa
`x @ 1..=5`	Lega il range corrisposto a **`x`**
`(a, b, ..)`	Destruttura tupla, ignora il resto
`Some(x) if x > 0`	Guard del match
`Foo { x, .. }`	Struct, ignora altri campi

Gestione degli Errori

Result e Option

```
enum Result<T, E> { Ok(T), Err(E) }
enum Option<T>    { Some(T), None }
```

L'Operatore ?

```
fn read_file(path: &str) -> Result<String, io::Error> {
    let mut s = String::new();
    File::open(path)?.read_to_string(&mut s)?;
    Ok(s)
}
```

Gestire gli Errori

```
match result {
    Ok(val) => println!("{val}"),
    Err(e) => eprintln!("Errore: {e}"),
}
let val = result.unwrap_or(0);
let val = result.unwrap_or_else(|_| default());
```

Metodi Comuni

`.unwrap()`	Ottieni valore o panic
`.expect(msg)`	Ottieni valore o panic con messaggio
`.unwrap_or(default)`	Ottieni valore o usa il default
`.map(f)`	Trasforma il valore Ok/Some
`.and_then(f)`	Concatena operazioni (flatmap)
`.is_ok() / .is_some()`	Controllo booleano

Rust Riferimento Rapido

Trait

Definire e Implementare

```
trait Summary {
    fn summarize(&self) -> String;
    fn preview(&self) -> String {   // impl di default
        format!("{}...", &self.summarize()[..20])
    }
}
impl Summary for User {
    fn summarize(&self) -> String { self.name.clone() }
}
```

Bound sui Trait

```
fn notify(item: &impl Summary) { }
fn notify<T: Summary + Display>(item: &T) { }
fn notify(item: &(impl Summary + Display)) { }
```

Trait Comuni

`Display`	Formattazione stringa per l'utente
`Debug`	Formattazione debug (**`{:?}`**)
`Clone, Copy`	Duplicazione (profonda / bit per bit)
`PartialEq, Eq`	Confronto di uguaglianza
`PartialOrd, Ord`	Confronto di ordinamento
`Iterator`	**`next()`** per l'iterazione
`From, Into`	Conversioni di tipo
`Default`	Costruttore del valore di default

Collezioni

Vec

```
let mut v: Vec<i32> = vec![1, 2, 3];
v.push(4);
v.pop();              // restituisce Option<i32>
let first = &v[0];   // panic se vuoto
let first = v.get(0); // restituisce Option<&i32>
```

HashMap

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("key", 42);
m.entry("key").or_insert(0);
if let Some(val) = m.get("key") { }
```

String

```
let s = String::from("hello");
let s = "hello".to_string();
let combined = format!("{} {}", s, "world");
for c in s.chars() { }   // itera i caratteri
```

Iteratori

```
let sum: i32 = vec![1, 2, 3].iter().sum();
let doubled: Vec<_> = v.iter().map(|x| x * 2).collect();
let evens: Vec<_> = v.iter().filter(|x| *x % 2 == 0).collect();
```

Concorrenza

Thread

```
use std::thread;
let handle = thread::spawn(|| {
    println!("dal thread spawned");
});
handle.join().unwrap();
```

Canali

```
use std::sync::mpsc;
let (tx, rx) = mpsc::channel();
tx.send(42).unwrap();
let val = rx.recv().unwrap();
```

Stato Condiviso

`Arc<T>`	Conteggio riferimenti atomico (Rc thread-safe)
`Mutex<T>`	Mutua esclusione, lock per accedere al valore
`RwLock<T>`	Più lettori o un solo scrittore
`Send`	Trait: sicuro da trasferire tra thread
`Sync`	Trait: sicuro da condividere tra thread

Macro e Attributi

Macro Comuni

`println!()`	Stampa con newline
`format!()`	Restituisce String formattata
`vec![]`	Crea Vec da letterali
`todo!()`	Segnaposto, panic a runtime
`assert!(expr)`	Panic se expr è false
`assert_eq!(a, b)`	Panic se a != b

Attributi Derive

```
#[derive(Debug, Clone, PartialEq)]
struct Point { x: f64, y: f64 }
// Implementa automaticamente Debug, Clone, PartialEq
```

Attributi per i Test

```
#[cfg(test)]
mod tests {
    use super::*;
    #[test]
    fn it_works() { assert_eq!(add(2, 2), 4); }
    #[test]
    #[should_panic]
    fn it_panics() { panic!("boom"); }
}
```