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

***Sicurezza e prestazioni senza compromessi nel linguaggio di sistema del futuro, con oltre 200 esercizi risolti***

***Di Tony Chan***

---

# *L'autore*

***Tony Chan*** è nato a South Gate, in California, Stati Uniti, da genitori immigrati cinesi. A 11 anni scrive il suo primo programma software in Basic, a 17 insegna agli amici programmazione e a 20 si laurea in informatica.

Recentemente ha partecipato come consulente alla redazione di alcuni testi sull'intelligenza artificiale e il linguaggio macchina. Successivamente decide di pubblicare i suoi libri sulla programmazione sia negli Stati Uniti, dove vive e lavora, ma anche in Italia, che ama e visita spesso. Un giorno, durante una *hackathon* (maratona di hacking) a New York, Tony scherzava sul fatto che Rust è così sicuro che potrebbe tranquillamente programmarlo bendato e con una sola mano. Per dimostrarlo, ha accettato la sfida di scrivere un algoritmo di ordinamento con una mano fasciata. Anche se ci ha messo un po' più del previsto, alla fine il programma non solo funzionava, ma era ottimizzato! Alla fine, ha detto: "Se solo Rust potesse anche cucinare..."

Infine, Tony è noto anche per fare battute sulle complessità della gestione della memoria. In un talk a Parigi ha scherzato: "Quando scrivo in C++, mi sembra di fare bungee jumping senza corda. Con Rust, è come avere una rete di sicurezza sempre sotto di me... anche se probabilmente non ne ho bisogno. E, cosa più importante, non mi sento mai in colpa per aver dimenticato di liberare la memoria!". Nessuno rise, perché chi partecipava non sapeva nulla di Rust...

## *Indice dei contenuti*


**# Introduzione** 8

**# 1 - Le prime basi** 23

*Cargo e la sintassi di base* *24*

*Le costanti e le variabili* *37*

*Tipologia di variabili* *40*

*Let e altre dichiarazioni* *50*

*Operatori e metodi* *55*

*Macro* *62*

*Ownership e borrowing* *64*

*I dati* *68*

*Main e Match* *112*

*I cicli e la preferenza per gli iteratori* *117*

*Distinzione tra iterabile e iteratore* *122*

*Approfondiamo gli operatori* *123*

*I primi passi* *127*

*Cast e numeri* *128*

*Quiz & esercizi / Riassunto* *133*

**# 2 - L'espressione in Rust** 136

*Gli identificatori* *142*

*Espressioni condizionali e strutture di controllo* *146*

*Espressioni lambda o chiusure* *150*

*La gestione delle eccezioni* *152*

*La gestione dei file e directory* *159*

*La gestione del tempo* *162*

*Quiz & esercizi / Riassunto* *164*

**# 3 - Le funzioni** 168

*Gli argomenti della funzione* *170*

*Definizione di una funzione* *172*

*Ambito delle variabili nelle funzioni* *174*

*Funzioni, Ownership e Borrowing* *177*
*Funzioni generiche e con trait bounds* *183*
*Funzioni anonime e chiusure* *187*
*Funzioni di ordine superiore* *191*
*Lifetimes* *194*
*I callback* *197*
*Funzioni variadiche (argomento avanzato)* *201*
*Iteratori e funzioni di utilità* *203*
*Funzioni asincrone* *206*
*Quiz & esercizi / Riassunto* *209*
**# 4 - I trait e le struct** 212
*Programmazione a tipi forti* *215*
*Definizione e utilizzo delle struct* *217*
*Trait, astrazione e comportamento* *225*
*Interazione tra struct e trait* *232*
*Gli oggetti trait* *235*
*Generics* *237*
*Modularità del codice con i trait* *240*
*Lifetimes, borrowing e trait* *244*
*Trait automatici* *247*
*Trait avanzati (argomento avanzato)* *250*
*Trait nelle librerie standard e nella programmazione asincrona* *257*
*Pattern e anti-pattern nell'uso di trait e struct* *261*
*Marker Trait e Polimorfismo ad-hoc (argomento avanzato)* *266*
*Quiz & esercizi / Riassunto* *269*
**# 5 - Ownership e Borrowing** 273
*Ownership e trasferimenti complessi di proprietà* *274*
*Prestiti (Borrowing) complessi* *277*
*Pattern di borrowing in funzioni e strutture dati* *280*
*Smart Pointers* *284*
*Lifetimes avanzati* *287*

*Errori comuni e come evitarli* *291*
*Threading sicuro con ownership e borrowing* *299*
*Gestione delle risorse* *302*
*Performance e ottimizzazioni relative a ownership* *306*
*Gli attributi* *309*
*Quiz & esercizi / Riassunto* *311*
**# 6 - Le espressioni regolari** 315
*Qualificatori, caratteri e operatori nelle regex* *317*
*Sequenze di escape per caratteri predefiniti* *323*
*Le sequenze speciali o complesse* *332*
*Metodi e Regex* *338*
*Quiz & esercizi* *342*
**# 7 - Moduli, Crate e Cargo** 345
*I moduli* *348*
*Organizzazione del codice* *353*
*Introduzione al Crate* *356*
*La gestione delle dipendenze tra Crate* *361*
*Cargo* *367*
*Pubblicazione di un Crate* *380*
*Workspace* *383*
*Debugging con Cargo* *386*
*Considerazioni finali* *388*
*Quiz & esercizi / Riassunto* *392*
**# 8 - Multithreading** 396
*Concorrenza e parallelismo* *398*
*Il modulo std::thread* *401*
*Ownership e borrowing nel multithreading* *403*
*Condivisione dei dati tra thread* *406*
*Sincronizzazione e programmazione asincrona* *410*
*Quiz & esercizi / Riassunto* *414*
**Link download gratuiti: EBOOK e soluzioni -** Bibliografia 416

# # Introduzione

Rust è un linguaggio di programmazione moderno sviluppato inizialmente da *Graydon Hoare* e sponsorizzato da Mozilla, con la sua prima release stabile rilasciata nel 2015. È nato con l'obiettivo di combinare prestazioni elevate e controllo della memoria, tipici di linguaggi come C e C++, con una forte enfasi sulla sicurezza e la prevenzione di errori comuni come i "data races" e i "buffer overflows". Rust si distingue per il suo innovativo sistema di gestione della memoria, che elimina la necessità di un *garbage collector* e riduce la probabilità di vulnerabilità legate alla memoria.

Il contesto in cui è stato creato è legato al crescente bisogno di un linguaggio che permettesse lo sviluppo di software sicuro e performante, specialmente in applicazioni critiche come sistemi operativi e software per il web, senza sacrificare la velocità di esecuzione. Mozilla, infatti, vedeva in Rust un linguaggio che potesse migliorare le prestazioni del browser Firefox, offrendo un'alternativa più sicura e moderna a C++.

Rust supporta diversi paradigmi di programmazione, come quella imperativa, funzionale e orientata agli oggetti. Tuttavia, uno degli aspetti più apprezzati è il suo concetto di "ownership", che garantisce una gestione sicura della memoria e l'assenza di errori di concorrenza. Inoltre, la sua sintassi è progettata per essere familiare a sviluppatori di linguaggi come C e C++, facilitando la transizione a chi è abituato a lavorare a basso livello.

Un altro elemento chiave di Rust è il suo ricco ecosistema, che include Cargo, il suo sistema di gestione dei pacchetti e build tool, e una community in continua

crescita che contribuisce al miglioramento e alla diffusione del linguaggio. Rust viene spesso scelto per lo sviluppo di sistemi embedded, motori di gioco, e software per il web, grazie alla sua velocità e affidabilità.
Negli ultimi anni, Rust è diventato sempre più popolare, ricevendo riconoscimenti come il linguaggio più amato su Stack Overflow per più anni consecutivi. Questo riflette non solo le sue capacità tecniche, ma anche la soddisfazione della comunità di sviluppatori che ne apprezza l'equilibrio tra efficienza e sicurezza.
In poche parole, Rust rappresenta una svolta nel panorama dei linguaggi di programmazione, ponendo al centro la sicurezza e le prestazioni senza compromettere la produttività dello sviluppatore. Con il suo continuo sviluppo e il crescente supporto della comunità, è destinato a diventare un pilastro fondamentale per lo sviluppo software di alto livello negli anni a venire.

***Breve storia***

La storia inizia nel 2006, quando *Graydon Hoare*, come detto un ingegnere software di Mozilla, iniziò a lavorare a un nuovo linguaggio di programmazione nel suo tempo libero. *Hoare* era motivato dalla necessità di risolvere problemi di sicurezza e prestazioni riscontrati nei linguaggi esistenti, come C e C++, soprattutto nella gestione della memoria. Rust venne progettato per offrire le stesse prestazioni di questi linguaggi di basso livello, ma senza i loro rischi associati, come i "data races" e i "buffer overflows". Il progetto attirò presto l'interesse di Mozilla, che nel 2009 decise di sponsorizzarlo ufficialmente, riconoscendo il potenziale di Rust per migliorare le prestazioni del browser Firefox.
Il linguaggio iniziò a guadagnare visibilità pubblica nel 2010, quando Mozilla presentò Rust al pubblico durante la sua conferenza annuale. Una delle pietre miliari fu l'annuncio del compilatore "rustc", scritto interamente in Rust stesso, che consolidava il linguaggio come stabile e pronto per il testing. Esso infatti si distingueva per il suo innovativo sistema di "ownership", che regolava l'accesso alla memoria e riduceva drasticamente gli errori di concorrenza, un problema

critico nelle applicazioni moderne, soprattutto con la crescente diffusione di sistemi multi-core.

Nel 2015, Rust raggiunse la sua prima versione stabile, la 1.0, dopo anni di sviluppo e feedback dalla comunità. La versione 1.0 rappresentava un importante traguardo, stabilizzando la sintassi e il modello di memoria. Questa versione lo consolidò come una scelta promettente per sviluppatori alla ricerca di sicurezza, prestazioni e concorrenza, soprattutto in ambienti critici come la programmazione di sistemi, lo sviluppo di motori di gioco e l'embedded.

Uno dei progetti più importanti che adottò Rust in questa fase fu il nuovo motore di rendering di Firefox, noto come "Servo". Esso fu un banco di prova per le sue capacità, dimostrando che un linguaggio sicuro e moderno poteva essere utilizzato per sviluppare componenti ad alte prestazioni in un software complesso come un browser web.

Negli anni successivi, continuò a evolversi rapidamente. Nel 2017, vennero introdotte nuove funzionalità come i "non-lexical lifetimes", che semplificarono ulteriormente la gestione della memoria, e l'introduzione delle "async/await", che resero la programmazione asincrona più accessibile e potente. Questi miglioramenti permisero a Rust di posizionarsi come un linguaggio adatto non solo per lo sviluppo di sistemi, ma anche per applicazioni web e cloud, aumentando la sua adozione.

Un'altra svolta avvenne con l'integrazione in ambienti *cloud* e *DevOps*. Grazie a strumenti come "Cargo", il suo gestore di pacchetti e build tool, e una community attiva, si consolidò come un linguaggio moderno con un ecosistema in espansione. Questo gli permise di competere con avversari più tradizionali (come Python o C#) in settori come la blockchain, l'intelligenza artificiale e il machine learning.

Rust ha guadagnato popolarità anche tra i programmatori, venendo eletto per più anni consecutivi il "linguaggio più amato" su Stack Overflow. La community attiva e l'approccio collaborativo allo sviluppo hanno garantito un'adozione costante e un'evoluzione basata sui bisogni reali degli sviluppatori.

Nel 2021, con la versione 1.50, Rust continuò a migliorare il suo ecosistema e a introdurre ottimizzazioni per il compilatore e nuove librerie standard. Oggi, è utilizzato da giganti della tecnologia come Amazon, Microsoft e Google, e continua a guadagnare terreno in progetti critici dove sicurezza e prestazioni sono fondamentali.
In sintesi, la storia di Rust è una testimonianza di come la sicurezza e l'efficienza possano coesistere in un linguaggio moderno. Nato come un progetto indipendente, ha rapidamente attirato l'attenzione del settore tecnologico, diventando uno dei linguaggi più promettenti per il futuro dello sviluppo software. La sua evoluzione continua a essere guidata dall'innovazione e dalla ricerca di soluzioni a problemi complessi, assicurando che rimanga un punto di riferimento per lo sviluppo di applicazioni sicure e performanti.

### *Lo standard e le ultime versioni*

Rust è un linguaggio di programmazione che segue un modello di rilascio molto stabile e organizzato. Lo standard è mantenuto da una comunità attiva sotto la guida della *Rust Foundation*, che garantisce un ecosistema solido e moderno. Il linguaggio ha un ciclo di rilascio regolare, con nuove versioni stabili ogni sei settimane. Queste introducono miglioramenti nelle performance, correzioni di bug, nuove funzionalità e a volte anche modifiche significative ma compatibili con le versioni precedenti. Una caratteristica chiave è il forte impegno per la retrocompatibilità, che assicura che il codice scritto con istruzioni obsolete continui a funzionare senza modifiche anche con le versioni più recenti.
Le ultime uscite di Rust hanno portato importanti innovazioni come il miglioramento della programmazione asincrona, con ottimizzazioni del compilatore per supportare meglio task asincroni e runtime più efficienti. Si sono visti anche sviluppi legati al sistema dei *trait*, che è stato potenziato per migliorare la flessibilità del linguaggio in termini di polimorfismo. L'evoluzione di strumenti come *Cargo* ha semplificato ulteriormente la gestione dei pacchetti e delle dipendenze. Il

linguaggio continua ad espandere il suo ecosistema, mantenendo la sua enfasi sulla sicurezza della memoria, la gestione delle risorse e l'efficienza, il che lo rende sempre più attraente sia per i progetti industriali di ampie dimensioni sia per lo sviluppo open-source.

Il linguaggio al momento prevede (https://releases.rs/docs/1.83.0/) un aggiornamento costante, in cui circa ogni mese viene rilasciato un pacchetto di novità e aggiustamenti. Ad esempio, La versione 1.81, uscita a Settembre del 2024, ha portato con sé una serie di interessanti novità, una delle modifiche più importanti riguarda la gestione degli errori nelle funzioni esterne. In precedenza, un panico non gestito in una funzione dichiarata come *extern "C"* poteva causare comportamenti imprevedibili. Adesso, il compilatore interrompe l'esecuzione del programma in caso di panico non gestito in queste funzioni, garantendo una maggiore sicurezza.

Un'altra novità riguarda la risoluzione dei metodi. In alcuni casi ambigui, poteva portare a risultati inaspettati. La nuova versione risolve questo problema, rendendo la risoluzione dei metodi più prevedibile e affidabile.

Inoltre, sono state stabilizzate alcune funzionalità sperimentali, come l'attributo *#[expect]* per i *lint,* che permette di sopprimere avvisi del compilatore in modo più controllato.

Oltre a ciò, Rust continua a rafforzare la sua programmazione sicura per il sistema, rendendolo una scelta sempre più comune per lo sviluppo di sistemi operativi, browser e altre applicazioni a basso livello dove le prestazioni e la sicurezza della memoria sono cruciali. L'interesse è in costante crescita, con molte aziende di rilievo che adottano Rust, e ogni nuova versione lo spinge verso una maturità sempre maggiore, mantenendo l'equilibrio tra innovazione e stabilità.

### *Rust: pro e contro*

La creatura di *Hoare* ha acquisito rapidamente popolarità grazie alla sua enfasi su prestazioni, sicurezza e gestione della memoria. Sebbene sia un linguaggio

giovane, ha già dimostrato di essere uno strumento potente e flessibile per una vasta gamma di applicazioni, ma come tutte le tecnologie, ha sia vantaggi che svantaggi da considerare.

Uno dei suoi principali punti di forza è, come detto, la sicurezza nella gestione della memoria. Rust è progettato per prevenire errori comuni come i "null pointer dereference", i "buffer overflows" e le condizioni di competizione nella concorrenza, senza l'uso di un garbage collector. Questo lo rende particolarmente adatto per lo sviluppo di software critico, dove la sicurezza e l'affidabilità sono fondamentali, come sistemi embedded, motori di gioco e software di rete. Il sistema di "ownership" e "borrow checker" è una delle caratteristiche distintive, che garantisce una gestione della memoria sicura a tempo di compilazione.

Un altro vantaggio è la sua velocità. Essendo un linguaggio di basso livello, permette di scrivere codice ad alte prestazioni comparabili a quelle di C e C++, rendendolo ideale per applicazioni dove le prestazioni sono cruciali. A differenza di molti linguaggi moderni, Rust offre questo livello di controllo sulle risorse senza sacrificare la sicurezza, il che lo rende unico nel suo genere.

L'ecosistema di Rust è anche uno dei suoi punti di forza. *Cargo*, il sistema di gestione dei pacchetti e build tool, semplifica enormemente il processo di gestione delle dipendenze e della compilazione del codice. Inoltre, la comunità è molto attiva e accogliente, con una vasta quantità di risorse, librerie open source e un forte supporto da parte degli sviluppatori.

Rust supporta anche una varietà di paradigmi di programmazione, tra cui quella imperativa, funzionale e concorrente. Questo lo rende flessibile e adatto a diversi tipi di progetti, permettendo agli sviluppatori di scegliere l'approccio più adatto per ogni contesto.

Tuttavia, ci sono anche degli svantaggi nell'uso di questo linguaggio. Uno dei principali è la curva di apprendimento. Sebbene sia progettato per essere sicuro e affidabile, la sua sintassi e le sue caratteristiche avanzate, come il sistema di "ownership", possono risultare complesse per i nuovi arrivati. Gli sviluppatori

provenienti da C o C++ potrebbero trovarsi disorientati dalle regole rigide di Rust per la gestione della memoria, e padroneggiarle richiede tempo e pratica.
Un altro aspetto da considerare è che, pur essendo un linguaggio versatile, può risultare eccessivo per progetti più semplici. Il livello di complessità e controllo che offre può essere superfluo in contesti in cui le prestazioni e la gestione precisa della memoria non sono priorità, rendendo linguaggi come Python o JavaScript più adatti a tali scopi.
Rust, pur essendo cresciuto rapidamente in popolarità, non ha ancora la stessa diffusione di linguaggi come C++, Java o Python. Questo significa che le risorse, gli strumenti e le librerie disponibili sono meno mature rispetto a linguaggi più affermati, anche se stanno crescendo rapidamente grazie al contributo della comunità.
Infine, uno degli svantaggi legati all'adozione di Rust è il tempo di compilazione, che può essere più lungo rispetto a linguaggi più tradizionali. Questo può rallentare il ciclo di sviluppo in alcuni contesti, soprattutto per progetti di grandi dimensioni.

### *Strumenti (Tools) di lavoro*

Anche se, teoricamente, per scrivere del codice in Rust potremmo utilizzare solo un semplice editor di testo come il blocco note, quando i progetti diventano più complessi avremo bisogno degli strumenti giusti per lavorare in modo efficiente. In questo contesto, è essenziale distinguere tra un IDE (ambiente di sviluppo integrato) e un editor di testo. Il primo non solo ci permette di scrivere il codice, ma anche di gestire l'intero processo di sviluppo, dalla scrittura alla creazione del programma, sfruttando il compilatore di Rust che traduce il codice sorgente in linguaggio macchina *direttamente eseguibile dal sistema operativo*.
La compilazione del codice Rust ha un impatto diretto sulle prestazioni, poiché il programma compilato viene eseguito nativamente e risulta molto più veloce rispetto a linguaggi interpretati. Un altro vantaggio della compilazione è che rende più difficile leggere il codice sorgente rispetto al codice interpretato, proteggendolo

da occhi indiscreti. Tuttavia, Rust è un linguaggio interamente compilato, quindi la fase di compilazione è sempre necessaria per eseguire il programma, contrariamente a linguaggi come Python o JavaScript che possono essere interpretati.
Rust non utilizza un modello JIT (*Just-In-Time*) come C#, ma adotta un modello di compilazione *ahead-of-time* (AOT), dove il codice viene tradotto in binari eseguibili già durante la fase di compilazione, prima dell'esecuzione. Questo garantisce che il programma sia ottimizzato per le prestazioni, ma allo stesso tempo richiede che tutto il codice venga validato in fase di compilazione, garantendo la sicurezza della memoria e l'assenza di determinati tipi di errori.
Nelle sezioni successive, esamineremo i principali strumenti, molti dei quali gratuiti, che possono essere utilizzati per scrivere, correggere e compilare codice in Rust, inclusi editor e IDE che supportano estensioni specifiche per il linguaggio.

### *IDE*

Come detto, per poter programmare in maniera seria non potremo fare a meno di un IDE (*Integrated Development Environment*), cioè di un ambiente di sviluppo integrato. In poche parole stiamo parlando di un programma che ci servirà per scrivere un listato, per testarlo ed eventualmente compilarlo, per renderlo eseguibile e indipendente. Di seguito ne elencheremo alcuni utili per i nostri esperimenti e per la realizzazione delle applicazioni.
La scelta del miglior IDE spesso dipende dalle preferenze personali e dalle specifiche esigenze di ciascun sviluppatore. Mentre quelli ufficiali come *RustRover* (https://www.jetbrains.com/rust/) offrono un'integrazione profonda con il linguaggio, esistono altre ottime opzioni che vale la pena considerare, eccone alcune di seguito:

- *Visual Studio Code* (https://code.visualstudio.com/): probabilmente l'editor più popolare al mondo, è gratuito, leggero e multipiattaforma, offre un'esperienza di sviluppo eccezionale grazie a un'ampia gamma di estensioni per Rust. È particolarmente popolare nella comunità, grazie al supporto fornito

dall'estensione ufficiale *rust-analyzer*, che offre funzionalità essenziali come l'evidenziazione della sintassi, l'auto-completamento intelligente, la refactoring e il debugging integrato. Visual Studio Code supporta anche il controllo di versione tramite Git, rendendo facile gestire i progetti versionati. Un altro strumento utile per questo editor è l'integrazione con il terminale interno, che consente di eseguire comandi di compilazione e test senza uscire dall'ambiente di sviluppo.

- *IntelliJ IDEA* (https://www.jetbrains.com/idea/): con il plugin Rust offre un potente ambiente più orientato agli sviluppatori che desiderano un'esperienza IDE completa, con strumenti avanzati per il debugging, l'analisi del codice e il refactoring. Il plugin include l'integrazione con Cargo, il gestore dei pacchetti e il build system, fornendo un supporto completo per lo sviluppo, compreso il controllo del codice, l'auto completamento avanzato e suggerimenti per la correzione degli errori.

- *Vim e Neovim* (https://www.vim.org/ - https://neovim.io/): questi editor altamente configurabili sono molto amati dagli sviluppatori che preferiscono un ambiente di lavoro minimal e personalizzabile. Esistono numerose plugin che li rendono eccellenti strumenti per lo sviluppo in Rust.

- *Emacs* (https://www.gnu.org/software/emacs/): un altro editor molto popolare e personalizzabile, offre un'ampia gamma di modalità e pacchetti per lo sviluppo.

- *Helix* (https://helix-editor.com/): un editor relativamente nuovo, è scritto in Rust e offre un'esperienza di sviluppo molto fluida.

La scelta dipende da molti fattori, se ti piace personalizzare il tuo ambiente di sviluppo, *Vim, Neovim* o *Emacs* potrebbero essere le scelte migliori, mentre se hai bisogno di funzionalità avanzate come il debugging integrato, la navigazione nel codice e il refactoring, *Visual Studio Code* o *IDEA* potrebbero essere più adatti.
Per garantire che il codice sia privo di errori sintattici e rispetti le migliori pratiche di programmazione, si utilizzano strumenti come *Clippy* (https://doc.rust-lang.org/clippy/index.html), un linter specifico per Rust, integrato nell'ecosistema di sviluppo, fornisce consigli e segnalazioni su possibili miglioramenti nei listati, correggendo errori comuni o suggerendo modifiche per rendere il codice più idiomatico. *Clippy si integra facilmente con Visual Studio Code e altri editor*, offrendo un feedback immediato durante lo sviluppo.

***Linguaggio macchina***

Il linguaggio macchina è il formato binario, costituito da sequenze di uno e zero, che il processore del computer può eseguire direttamente. I linguaggi di programmazione, come Rust, servono a tradurre concetti e logiche di alto livello in istruzioni che il computer possa comprendere. La programmazione si distingue in linguaggi di basso livello e linguaggi di alto livello, a seconda di quanto il linguaggio sia vicino o distante dal linguaggio macchina. I linguaggi di basso livello, come l'Assembly, richiedono agli sviluppatori di lavorare molto vicino all'hardware, gestendo manualmente risorse come la memoria e i registri del processore. Rust, invece, è un linguaggio di alto livello, progettato per fornire astrazioni moderne pur mantenendo il controllo sulle risorse, come la gestione della memoria, rendendolo ideale per lo sviluppo di sistemi ad alte prestazioni e sicuri.

Abbiamo già anticipato che Rust è un linguaggio compilato *ahead-of-time* (AOT), il che significa che il codice sorgente viene interamente tradotto in linguaggio macchina tramite un processo di compilazione, prima che l'applicazione venga eseguita. Il compilatore, chiamato *rustc*, prende il listato scritto dallo sviluppatore e lo trasforma in un file binario eseguibile direttamente dal sistema operativo. Questo lo differenzia da linguaggi come C# o Java, che tipicamente utilizzano una compilazione JIT (*Just-In-Time*), dove il codice viene parzialmente interpretato o compilato al momento dell'esecuzione.

Un'altra caratteristica fondamentale è l'enfasi sulla gestione sicura della memoria. A differenza di linguaggi come C++ o Assembly, dove lo sviluppatore deve manualmente allocare e deallocare la memoria, Rust gestisce automaticamente queste operazioni grazie al suo sistema di *ownership* e *borrow checking,* evitando problemi comuni come i *memory leaks* o i *null pointer dereference*. Questo lo rende una scelta eccellente per la scrittura di software di sistema o di applicazioni ad alte prestazioni, dove sia la sicurezza della memoria che l'efficienza sono cruciali.

Una volta che il codice è stato compilato, l'applicazione risultante può essere eseguita direttamente su diverse piattaforme (Windows, Linux, macOS), a patto che sia stata compilata per l'architettura e il sistema operativo corretti. Questo processo, chiamato *cross-compilation*, è supportato dal gestore dei pacchetti e sistema di build di Rust, *Cargo*, che consente di costruire il programma per diverse piattaforme senza modificare il codice sorgente.
In poche parole, rispetto a linguaggi come C# o Java, Rust non richiede ambienti runtime o macchine virtuali, poiché il codice viene eseguito direttamente dal sistema operativo.

### *Intelligenze Artificiali (IA)*

Negli ultimi anni questa tecnologia è stata consacrata all'uso del grande pubblico. A oggi esistono molte opzioni disponibili gratuitamente, alcune specializzate solo in determinati ambiti, come ad esempio la creazione di video, di immagini o la guida autonoma. Quasi tutti hanno sicuramente provato e sperimentato il fantastico giocattolo di OpenAI, cioè ChatGPT (*Generative Pretrained Transformer*), che è un sistema di apprendimento pre-addestrato generativo. Esso si serve di una rete neurale e di algoritmi complessi di deep learning per comprendere e generare testo di risposta in base all'input dell'utente.
Questa tecnologia è utile anche nel nostro ambito, poiché grazie a esso è possibile avere un dialogo costruttivo su argomenti legati alla programmazione e anche ottenere del codice come esempio. In alcuni casi, potrebbe sostituire un tutor o risolvere una situazione in cui un listato crea dei problemi. Oltre a ChatGPT, ci sono anche altre opzioni disponibili, tra cui sicuramente Gemini (prima Bard) di Google. Consigliamo di dare un'occhiata a entrambi.

### *Debug, Boilerplate code e convenzioni*

In questo, come in qualsiasi altro linguaggio di programmazione, il debugging e l'uso di convenzioni chiare sono fondamentali per mantenere il codice robusto e

leggibile. Rust è noto per il suo forte sistema di sicurezza e per la gestione della memoria senza garbage collector, tuttavia, richiede attenzione in alcuni aspetti.
Il debugging può essere facilitato dall'uso della macro *dbg!(),* che stampa su console il valore e la posizione del codice. Questa funzione è molto utile durante lo sviluppo per verificare il valore di una variabile o l'esecuzione di un determinato blocco di codice:

```
let x = 5;
dbg!(x); // Stampa x = 5
```

Un altro strumento utile è la macro *println!(),* che consente di stampare variabili in fase di esecuzione. Tuttavia, si richiede che i tipi da stampare implementino il trait *Debug.* Molti tipi standard lo fanno di default, ma se crei un tipo personalizzato, dovrai derivarlo manualmente con *#[derive(Debug)].*
Rust cerca di ridurre il boilerplate code, ovvero il codice ripetitivo e ridondante, sempre grazie a funzionalità come i trait e le macro. Le prime permettono di generare codice ripetitivo automaticamente. Ad esempio, la macro *vec![]* crea facilmente un vettore senza dover dichiarare ogni elemento singolarmente.
Rust usa anche il concetto di "*zero-cost abstractions*", il che significa che strutture ad alto livello come *Result* e *Option* non comportano un costo extra in termini di prestazioni, riducendo la necessità di scrivere codice boilerplate per la gestione degli errori o dei valori nulli. Ovviamente tutti questi concetti verranno ampliamente trattati nel corso dei vari capitoli.
Infine il linguaggio segue una serie di convenzioni stilistiche che aiutano a mantenere il codice leggibile e uniforme. Queste includono:

- *Nomi delle variabili e delle funzioni*: qui si usa lo snake_case (es. my_variable).
- *Nomi delle strutture e dei trait*: si usa il CamelCase per nomi di strutture e trait (es. MyStruct).
- *Indentazione*: lo standard è di 4 spazi per ogni livello di annidamento.
- *Gestione degli errori*: è fortemente incoraggiato l'uso dei tipi *Result* e *Option* per gestire gli errori e i valori opzionali, piuttosto che fare affidamento su eccezioni o valori nulli.

Oltre a ciò è bene conoscere alcuni termini comuni in Rust:

*Inferire*: in Rust, il compilatore può inferire (ricavare) i tipi di variabili senza doverli dichiarare esplicitamente, grazie alla sua capacità di dedurre automaticamente il tipo da contesto, riducendo così la verbosità del codice pur mantenendo la sicurezza dei tipi.
*Monomorfizzazione*: processo in cui il compilatore genera versioni concrete di funzioni generiche per ciascun tipo utilizzato, ottimizzando il codice per eliminare l'overhead associato al dispatch dinamico.
*Puntatori e puntatori raw*: i puntatori in Rust, come *&* e *&mut*, sono sicuri perché rispettano le regole del borrowing e dell'ownership; i puntatori raw, come **const* e **mut*, sono meno sicuri perché permettono l'accesso diretto alla memoria senza le garanzie del borrowing.
*Heap e stack*: il primo è usato per dati dinamici con dimensioni variabili, richiedendo però più risorse per gestire la memoria. L'altro per dati di dimensioni fisse. Ha un accesso molto più rapido.
*Ownership e borrowing*: l'ownership rappresenta il controllo esclusivo di una variabile, mentre il borrowing permette di prendere in prestito il valore di una variabile temporaneamente, sia in modo mutabile che immutabile, mantenendo la sicurezza della memoria.
*Operazioni atomiche*: sono operazioni che vengono eseguite come un'unica operazione indivisibile, usate per sincronizzare l'accesso concorrente ai dati tra thread senza incorrere in race conditions.
*Struct, trait e crate*: una struct è una struttura dati che raggruppa variabili; i trait definiscono il comportamento che può essere implementato dalle strutture, mentre un crate è un'unità di compilazione, come una libreria o un pacchetto.
*Null pointer, race condition*: Un "null pointer" rappresenta un puntatore che non punta a nessun valore valido, mentre una "race condition" si verifica quando più thread accedono a dati condivisi senza la corretta sincronizzazione, causando comportamenti imprevedibili.
*Attributi e riferimenti*: i primi servono per modificare il comportamento del compilatore o del codice, mentre i riferimenti (*&* e *&mut*) sono puntatori sicuri che permettono l'accesso a valori senza trasferirne l'ownership.
*Pattern Matching:* consente di destrutturare, confrontare e gestire i dati in modo sicuro e conciso. Utilizzando la parola chiave *match*, si possono esaminare diversi casi di un tipo, come un *enum* o *Option*, e gestire tutte le possibilità in modo esaustivo, migliorando la sicurezza del codice.
*Polimorfismo ad hoc*: si riferisce alla capacità di una funzione o di un metodo di lavorare con diversi tipi di dati, ma con comportamenti diversi per ciascuno. Ciò viene realizzato tramite i trait, che consentono di definire un insieme di metodi che devono essere implementati dai tipi che "aderiscono" ad esso. Ogni tipo può avere un'implementazione specifica per quei metodi, permettendo comportamenti differenziati.
*Pattern*: è una soluzione ricorrente e consolidata per risolvere problemi comuni di progettazione del

software. Sono schemi efficaci, ripetibili e utilizzabili in molteplici contesti. Un esempio di pattern è l'uso di Option e Result per gestire i valori opzionali e gli errori in modo sicuro.
*Anti-pattern*: è una pratica comune ma inefficace o dannosa che porta a cattive prestazioni, bassa manutenibilità o errori. Un esempio potrebbe essere l'uso eccessivo di puntatori raw, che può introdurre vulnerabilità di memoria, poiché bypassa il sistema di ownership e borrowing.

### *Commentare il codice*

Durante il nostro studio, ci troveremo ad avere dei listati da interpretare o semplicemente da copiare. In molti di questi saranno presenti dei commenti per aiutarci a comprendere meglio il metodo, la sintassi, la classe o la porzione di codice in esame. Ogni linguaggio, per permettere d'integrare del testo avulso dal codice, ha una sua grammatica specifica. Rust supporta tre tipi di commenti:

- *Commenti su una riga*: usano //. Sono simili ad altri linguaggi e vengono utilizzati per spiegazioni brevi o annotazioni inline:

```
// Questo è un commento su una riga
```

- *Commenti multilinea*: si usano /* */. Sono utili per spiegazioni più lunghe o blocchi di testo:

```
/* Questo è un commento
   su più righe */
```

- *Commenti per la documentazione*: Rust ha una sintassi per la generazione automatica della documentazione. Usando ///, puoi scrivere commenti che vengono incorporati nella documentazione generata dal compilatore:

```
/// Questa funzione restituisce il massimo tra due numeri.
fn max(a: i32, b: i32) -> i32 {
    if a > b { a } else { b }
}
```

Questi sono spesso associati a funzioni, strutture e moduli per spiegare come utilizzarli. Rust offre un'utility chiamata *rustdoc* che genera documentazione HTML a partire da questi commenti.

# # 1 - Le prime basi

*Costanti e variabili*
*Tipi di dati*
*Le iterazioni*
*Gli operatori*
*Cast*

Rust è un linguaggio di programmazione moderno che si distingue per la sua enfasi sulla sicurezza della memoria e sul controllo della concorrenza. Nato con l'intento di combinare le prestazioni e il controllo tipici del linguaggio C/C++ con una gestione della memoria sicura, Rust evita una delle principali insidie della programmazione a basso livello: i bug legati alla memoria, come dereferenziamenti di puntatori nulli o buffer overflow. Questo è possibile grazie a un sistema di gestione della memoria che, come abbiamo già anticipato, non si affida a un garbage collector, ma utilizza invece un concetto chiamato "ownership", il quale permette di sapere esattamente chi possiede una risorsa in ogni momento, evitando così problemi di accesso concorrente.
Rust si rivela particolarmente adatto in contesti in cui le prestazioni e l'efficienza sono fondamentali, come lo sviluppo di sistemi operativi, motori di gioco e software di rete. Inoltre, offre un'esperienza di sviluppo robusta, con un compilatore che funge da guida durante la scrittura del codice, segnalando errori non appena si presentano, il che riduce significativamente i cicli di debug. Il suo ecosistema, in continua crescita, include una comunità attiva e strumenti come *Cargo*, un gestore di pacchetti e build system che semplifica enormemente la gestione dei progetti.
Un altro aspetto interessante è la capacità di integrazione con altri linguaggi, in

particolare con C e C++, grazie a una FFI (*Foreign Function Interface*) ben progettata. Ciò gli consente di essere adottato progressivamente in progetti legacy senza la necessità di riscrivere interamente il codice esistente. Rust, quindi, non solo promuove la scrittura di codice sicuro ed efficiente, ma lo fa senza sacrificare la flessibilità, rendendolo una scelta interessante sia per progetti nuovi sia per quelli già avviati.

### *Cargo e la sintassi di base*

Cargo è lo strumento di gestione dei pacchetti e il build system per questo linguaggio. È un componente essenziale dell'ecosistema Rust, che semplifica notevolmente il processo di gestione delle dipendenze, compilazione del codice, esecuzione dei test e creazione di progetti. Quando si lavora con il codice, Cargo gestisce quasi tutti gli aspetti del ciclo di vita di un progetto, dall'inizio dello sviluppo fino alla distribuzione del software.
Per installarlo, è necessario dotarsi dell'intero ecosistema, poiché Cargo è incluso nel pacchetto di Rust. Il modo più semplice per farlo è utilizzare il gestore di pacchetti *rustup* (https://www.rust-lang.org/tools/install), che è lo strumento ufficiale per installarlo e gestirlo. Se disponiamo di un sistema UNIX (come Linux o macOS) o Windows, basta eseguire il seguente comando nel terminale o nel prompt dei comandi:

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Una volta completata l'installazione, è possibile verificare che tutto sia stato eseguito correttamente digitando:

```
cargo --version
```

Se il comando restituisce la versione corrente, significa che è andato tutto liscio e sei pronto per partire.
Cargo serve principalmente per inizializzare, compilare e gestire progetti Rust.

Quando si crea un nuovo progetto, genera la struttura di directory necessaria e un file *Cargo.toml,* che contiene informazioni sulle dipendenze, la versione del progetto, il nome e altre configurazioni. Per creare un nuovo progetto, basta eseguire il comando:

```
cargo new nome_progetto
```

Ad esempio, se vuoi creare un progetto chiamato, senza troppa fantasia, *primo_progetto*, esegui:

```
cargo new primo_progetto
```

Questa riga creerà la cartella *primo_progetto* con questa struttura di base:

```
primo_progetto
├── Cargo.toml
└── src
    └── main.rs
```

Il file *Cargo.toml* contiene i metadati del progetto e la lista delle dipendenze. Il file *main.rs* è il punto di ingresso del programma. Per compilarlo ed eseguirlo, basta entrare nella sua directory e digitare:

```
cd primo_progetto
cargo run
```

Così Cargo compila il progetto e, se va tutto liscio, esegue il programma. Il comando *cargo run* è molto utile durante lo sviluppo, *poiché combina la compilazione e l'esecuzione in un unico passaggio.*
Un esempio di codice basilare che puoi inserire nel file *main.rs* è il seguente:

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

Questo è un codice molto semplice che stampa "Ciao, mondo!" sulla console.

Quando esegui *cargo run*, si eseguono due azioni, la compilazione e l'esecuzione del programma, che produrrà l'output:

```
Ciao, mondo!
```

Cargo non si limita alla gestione dei progetti e delle dipendenze, ma offre anche funzionalità per il testing. In Rust esiste già un sistema integrato molto performante, e con Cargo si dispone di una struttura esclusiva e più funzionale. Per eseguire tutti i test del nostro programma basterà eseguire:

```
cargo test
```

Questa riga compila il progetto, esegue i test definiti e mostra i risultati.
Cargo è uno strumento estremamente efficace che facilita l'intero processo di sviluppo. Sia che tu stia gestendo un piccolo lavoro o programmando un sistema complesso con molte dipendenze, questo ambiente di sviluppo è essenziale per mantenere il codice organizzato, testato e pronto per la distribuzione.
Approfondiremo Cargo più avanti, passiamo adesso alla sintassi di base in Rust, la quale segue una struttura ben definita che consente di scrivere codice sicuro ed efficiente. Si utilizza la dichiarazione *use* per importare moduli e funzionalità specifiche da librerie esterne o interne. Questo semplifica l'accesso a funzioni, strutture e tipi senza dover specificare il percorso completo ogni volta:

```
use std::io; // Importa il modulo di input/output dalla libreria standard
```

Il punto di ingresso di ogni programma è la funzione *main*. Tutto il codice eseguibile deve partire da qui. La funzione *main* non accetta argomenti per impostazione predefinita, ma è possibile modificarla per farlo. Il blocco di codice al suo interno contiene le istruzioni che il programma eseguirà:

```
fn main() {
    // Il codice esegue il programma
}
```

Le variabili sono immutabili per default, il che significa che una volta assegnato un valore, questo non può essere modificato a meno che non lo si dichiari come mutabile usando *mut*. Oltre a ciò è richiesta la dichiarazione esplicita del tipo di dati *solo se non è deducibile dal contesto*:

```
fn main() {
    let messaggio = "Ciao, mondo!"; // Variabile immutabile
    println!("{}", messaggio); // Stampa sulla console

    let mut numero = 42; // Variabile mutabile
    numero = 43; // È possibile modificarla
    println!("Il numero è {}", numero);
}
```

Rust supporta le strutture di controllo comuni come *if, else, loop, while* e *for*. Queste permettono di controllare il flusso del programma in base a condizioni o cicli ripetuti. La sintassi è simile a quella di altri linguaggi, ma con alcune differenze, come la possibilità di usare *if* come espressione:

```
fn main() {
    let numero = 42;

    if numero > 40 {
        println!("Il numero è maggiore di 40.");
    } else {
        println!("Il numero è 40 o minore.");
    }
}
```

Il linguaggio adotta un approccio esplicito per la gestione degli errori attraverso i tipi *Result* e *Option*. Questo ne incoraggia una gestione sicura, anche degli stati opzionali, riducendo la probabilità di crash del programma. È comune vedere funzioni che restituiscono un *Result*, che poi viene gestito con un *match* o utilizzando i metodi *.unwrap(), .expect(),* o operatori come ? per propagare l'errore:

```
fn main() {
    let risultato = divisione(10, 2);

    match risultato {
        Ok(valore) => println!("Risultato: {}", valore),
        Err(e) => println!("Errore: {}", e),
    }
}

fn divisione(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err(String::from("Divisione per zero"))
    } else {
        Ok(a / b)
    }
}
```

Rust permette di organizzare il codice in moduli (*mod*), che aiutano a suddividere il programma in parti più piccole e gestibili. Possono essere definiti nello stesso file o distribuiti su più file. Facilitano la gestione del codice, specialmente in progetti più grandi, e possono essere resi pubblici usando *pub* per consentirne l'accesso da altri moduli:

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

fn main() {
    let risultato = calcoli::somma(5, 3);
    println!("Il risultato della somma è: {}", risultato);
}
```

Mettiamo insieme queste componenti per creare un semplice programma che legge un input dall'utente, effettua un calcolo e gestisce eventuali errori:

```
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");

    let risultato = if numero > 40 {
        "Il numero è maggiore di 40."
    } else {
        "Il numero è 40 o minore."
    };

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

Partiamo con *use std::io* per importare le funzionalità di input/output. La funzione *main* legge un numero dall'utente, lo converte da stringa a intero e lo valuta con una struttura *if*. Il programma gestisce eventuali errori durante la lettura dell'input o la conversione del numero, utilizzando *.expect()* per fornire messaggi di errore chiari in caso di fallimento.

Anticipiamo una delle caratteristiche più distintive, il sistema di *ownership*, che gestisce la memoria in modo sicuro. Ogni valore ha un proprietario, e quando esce dallo scope (ambito), questo viene automaticamente deallocato. Ciò, come già detto, elimina la necessità di un garbage collector e previene molti tipi di bug legati alla gestione della memoria.

Il concetto di "borrowing" permette di prestare una variabile senza trasferirne la proprietà, sia in modo mutabile che immutabile:

```
fn main() {
    let s = String::from("ciao");
    prendi_ownership(s); // Ownership trasferita
    // Non possiamo più usare `s` qui

    let x = 10;
    prendi_in_prestito(&x); // `x` viene preso in prestito, ownership non trasferita
    // Possiamo ancora usare `x` qui
```

```
}

fn prendi_ownership(s: String) {
    println!("{}", s);
}

fn prendi_in_prestito(y: &i32) {
    println!("{}", y);
}
```

Questa sintassi di base fornisce le fondamenta per scrivere programmi sicuri ed efficienti, combinando controllo rigoroso e flessibilità nello sviluppo di software.
Abbiamo visto poco fa che la dichiarazione *use* serve per importare moduli, funzioni, strutture e altri elementi da librerie esterne o interne, rendendoli accessibili senza dover specificare ogni volta il percorso completo. Che cosa si può importare con *use*?

*Moduli*: interi moduli o sottosezioni di essi.
*Funzioni, strutture e tipi*: specifici elementi di un modulo.
*Enum e varianti*: tipi enum e le loro varianti.
*Elementi di un modulo standard o di librerie esterne*: permette di accedere facilmente alle funzionalità delle librerie standard o di terze parti.

Rust ha una libreria standard (*std*) ricca di moduli che forniscono le funzionalità di base. Di seguito alcuni dei moduli principali con esempi su come importarli e utilizzarli:

- **std::io**: gestisce l'input e l'output, consentendo di leggere dall'input standard (come la tastiera) e scrivere sull'output standard (come lo schermo). Inoltre, fornisce strumenti per la gestione dei flussi di dati (*streams*), lettura e scrittura su file, e gestione degli errori legati all'I/O:

```
use std::io;

fn main() {
```

```
    let mut input = String::new();
    println!("Inserisci qualcosa:");

    // Legge una riga dall'input standard e la memorizza in `input`
    io::stdin().read_line(&mut input).expect("Errore nella lettura dell'input");

    println!("Hai inserito: {}", input);
}
```

Importiamo il modulo *std::io* e utilizziamo *io::stdin()* per leggere una riga di testo dall'input dell'utente.

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

fn main() {
    let mut input = String::new();
    print!("Inserisci il tuo nome: ");
    io::stdout().flush().unwrap();  // Forza la stampa immediata
    io::stdin().read_line(&mut input).expect("Errore nella lettura");
    println!("Ciao, {}!", input.trim());
}
```

In questo esempio, *io::stdin* legge una linea di input dall'utente, e *io::stdout().flush()* garantisce che il messaggio venga mostrato immediatamente.

- **std::fs**: fornisce funzionalità per lavorare con il file system, come strumenti per la lettura e scrittura di file, la gestione dei percorsi, la creazione di directory, e altre operazioni correlate ai file direttamente dal codice:

```
use std::fs;

fn main() {
    // Scrive una stringa in un file
    fs::write("output.txt", "Ciao, mondo!").expect("Impossibile scrivere nel file");

    // Legge il contenuto di un file
    let contenuto = fs::read_to_string("output.txt").expect("Impossibile leggere il file");

    println!("Contenuto del file: {}", contenuto);
```

```
}
```

Qui, *use std::fs* importa il modulo *fs*, e poi utilizziamo *fs::write* per scrivere una stringa in un file e *fs::read_to_string* per leggerne il contenuto.

- **std::collections**: fornisce una varietà di strutture dati utili come *HashMap, Vec, BTreeMap,* strutture dati fondamentali per organizzare e gestire le collezioni in modo efficiente:

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

fn main() {
    let mut mappa = HashMap::new();

    mappa.insert("Chiave1", 10);
    mappa.insert("Chiave2", 20);

    for (chiave, valore) in &mappa {
        println!("{}: {}", chiave, valore);
    }
}
```

In questo caso, importiamo un *HashMap* per creare e manipolare una mappa per associare chiavi a valori, come una sorta di dizionario.

- **std::thread**: permette di lavorare con la concorrenza, creando e gestendo thread. Questo è particolarmente utile per sfruttare i processori multi-core e migliorare le prestazioni delle applicazioni attraverso la concorrenza:

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Ciao dal thread secondario! {}", i);
            thread::sleep(Duration::from_millis(100));
```

```
        }
    });

    for i in 1..5 {
        println!("Ciao dal thread principale! {}", i);
        thread::sleep(Duration::from_millis(50));
    }

    handle.join().unwrap();
}
```

Qui, utilizziamo *use std::thread* per importare il modulo *thread*, creandone uno nuovo che esegue un loop, e poi facciamo "dormire" il thread principale per un breve periodo.

- **std::sync**: offre strumenti per la sincronizzazione tra thread, come *Mutex* e *Arc.* Questi consentono di condividere dati tra thread in modo sicuro, prevenendo race conditions e altri problemi comuni nella programmazione concorrente:

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

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

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

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

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

```
}
```

Utilizziamo *Mutex* per garantire l'accesso sicuro a un contatore condiviso tra più thread. Mentre con *Arc* gestiamo il contatore in maniera sicura e condivisa.

- **std::time**: gestisce il tempo e le durate temporali. È utile per misurare la durata di operazioni, creare timeout e gestire il tempo in applicazioni real-time:

```
use std::time::{Duration, Instant};

fn main() {
    let inizio = Instant::now();

    // Simula un'operazione che richiede tempo
    std::thread::sleep(Duration::from_secs(2));

    let durata = inizio.elapsed();
    println!("Operazione completata in: {:.2?} secondi", durata);
}
```

Qui, *Instant::now()* cattura il tempo corrente, e *elapsed()* calcola quanto ne è passato, permettendo di misurare la durata di un'operazione.

- **std::env**: permette di interagire con l'ambiente del sistema, come le variabili d'ambiente e gli argomenti della riga di comando:

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    println!("Argomenti passati:");
    for arg in args {
        println!("{}", arg);
    }
}
```

Come visto, *use std::env* importa il modulo *env*, e poi utilizziamo *env::args()* per

raccogliere gli argomenti della riga di comando in un vettore.
È possibile importare tutto il contenuto di un modulo usando *, oppure importare specifici elementi usando la virgola per separarli:

```
use std::collections::*;
```

Questo importa tutti i tipi e funzioni dal modulo *std::collections.*
Osserviamo adesso l'importazione selettiva:

```
use std::cmp::{min, max};

fn main() {
    let a = 10;
    let b = 20;

    println!("Minimo: {}", min(a, b));
    println!("Massimo: {}", max(a, b));
}
```

Qui, importiamo solo le funzioni *min* e *max* dal modulo *std::cmp*.
La dichiarazione *use* è uno strumento performante che semplifica l'accesso alle funzionalità dei moduli. Che si tratti di leggere input, lavorare con file, utilizzare strutture dati avanzate o gestire la concorrenza, Rust offre una vasta gamma di moduli standard accessibili tramite questa istruzione. Ciò permette di mantenere il codice organizzato e pulito, evitando ripetizioni inutili.
Essenzialmente rappresentano le fondamenta per molte operazioni comuni, offrendo strumenti potenti per gestire I/O, dati, concorrenza, sincronizzazione, e temporizzazione. Vediamone quindi un elenco di quelli principali:

| Moduli | Classi | Descrizione |
|---|---|---|
| `std::io` | `stdin, stdout, Read, Write` | *Gestisce l'input/output, permette di leggere e scrivere su flussi come la console.* |
| `std::fs` | `File, read_to_string, write` | *Fornisce funzionalità per interagire con il file system, inclusa la lettura e scrittura di file.* |
| `std::collections` | `HashMap, Vec, BTreeMap` | *Contiene strutture dati collezionistiche come mappe hash, vettori e alberi bilanciati.* |
| `std::thread` | `spawn, JoinHandle` | *Fornisce strumenti per la concorrenza tramite thread, permettendo l'esecuzione parallela del* |

| | | |
|---|---|---|
| | | *codice.* |
| `std::sync` | `Mutex, Arc, RwLock` | *Gestisce la sincronizzazione tra thread, garantendo accesso sicuro a risorse condivise.* |
| `std::time` | `Duration, Instant` | *Rappresenta il tempo e le durate temporali, utile per operazioni temporizzate e calcoli di timeout.* |
| `std::env` | `args, var, set_var` | *Permette di interagire con l'ambiente del sistema operativo, come leggere argomenti e variabili d'ambiente.* |
| `std::net` | `TcpListener, TcpStream, UdpSocket` | *Fornisce funzionalità di rete per creare e gestire connessioni TCP e UDP.* |
| `std::process` | `Command, exit, Child` | *Permette di eseguire processi esterni e di gestire l'esecuzione di comandi di sistema.* |
| `std::cmp` | `min, max, Ordering` | *Contiene funzioni e tipi per il confronto di valori, utile per ordinamento e comparazioni.* |
| `std::option` | `Option` | *Rappresenta un valore che può o non può essere presente, utilizzato per gestire valori opzionali.* |
| `std::result` | `Result` | *Gestisce il risultato di operazioni che possono fallire, con valore di successo o errore.* |
| `std::io` | `stdin, stdout, Read, Write` | *Gestisce l'input/output, permette di leggere e scrivere su flussi come la console.* |
| `std::fs` | `File, read_to_string, write` | *Fornisce funzionalità per interagire con il file system, inclusa la lettura e scrittura di file.* |
| `std::collections` | `HashMap, Vec, BTreeMap` | *Contiene strutture dati collezionistiche come mappe hash, vettori e alberi bilanciati.* |
| `std::thread` | `spawn, JoinHandle` | *Fornisce strumenti per la concorrenza tramite thread, permettendo l'esecuzione parallela del codice.* |
| `std::sync` | `Mutex, Arc, RwLock` | *Gestisce la sincronizzazione tra thread, garantendo accesso sicuro a risorse condivise.* |
| `std::time` | `Duration, Instant` | *Rappresenta il tempo e le durate temporali, utile per operazioni temporizzate e calcoli di timeout.* |
| `std::env` | `args, var, set_var` | *Permette di interagire con l'ambiente del sistema operativo, come leggere argomenti e variabili d'ambiente.* |

Rust è un linguaggio progettato con un obiettivo chiaro in mente: garantire che il codice sia sicuro e privo di errori legati alla gestione della memoria, senza sacrificare le prestazioni, come già detto, evita completamente il garbage collector, utilizzando l'ownership. Questo sistema si basa su regole che il compilatore verifica al momento della compilazione del codice, impedendo a priori errori comuni come i "dangling pointer", l'uso della memoria dopo che è stata liberata o i doppioni nella liberazione della memoria stessa. Tutto questo avviene senza costi aggiuntivi di runtime, rendendo il codice non solo sicuro, ma anche altamente efficiente.

Quando si tratta di mutuo accesso ai dati in ambienti concorrenti, Rust introduce un meccanismo di borrowing e reference counting che garantisce che due parti del codice non possano accedere mutualmente e simultaneamente ai dati in modo che possa causare conflitti. In altre parole, il linguaggio assicura che un dato non

possa essere mutato mentre viene letto da altre parti del programma, prevenendo così le race conditions, una delle cause principali di bug in applicazioni multithread. Questa protezione viene fornita attraverso il sistema dei "lifetime" e delle "reference", che consente al compilatore di tracciare chi sta accedendo a cosa e per quanto tempo, in modo da evitare qualsiasi accesso illecito ai dati.
L'efficienza di Rust si estende anche alla sua gestione delle operazioni concorrenti e asincrone. In molti linguaggi, la concorrenza può portare a complesse situazioni di errore che possono essere difficili da risolvere. Rust, invece, offre un supporto robusto per la concorrenza sicura, permettendo agli sviluppatori di scrivere codice che può eseguire più compiti in parallelo senza doversi preoccupare di deadlock o race conditions. Inoltre, la programmazione asincrona è resa sicura e intuitiva grazie all'uso delle future e del sistema di task asincroni, che gestiscono le operazioni senza bloccare il thread principale e mantenendo il controllo rigoroso sull'accesso alla memoria.
Queste caratteristiche rendono Rust unico nel suo approccio: la combinazione di sicurezza della memoria, controllo preciso sul mutuo accesso ai dati, e l'efficienza nella gestione della concorrenza e delle operazioni asincrone, rende possibile scrivere software altamente performante e affidabile, riducendo al minimo i rischi di errori gravi. Per un principiante, capire questi concetti è il primo passo verso l'apprezzamento del motivo per cui Rust è così apprezzato, soprattutto in contesti dove la sicurezza e le prestazioni sono fondamentali.

### *Le costanti e le variabili*

Nell'ambito della progettazione e sviluppo software, è essenziale comprendere e gestire dati in diverse forme. Tra questi, le costanti e le variabili rivestono un ruolo fondamentale. Le costanti sono valori immutabili che mantengono la loro integrità durante l'esecuzione del programma, mentre le variabili consentono la memorizzazione e la manipolazione dinamica dei dati. Entrambe sfruttano la memoria ad accesso casuale (RAM) del dispositivo utilizzato e si esauriscono al

termine dell'esecuzione, a meno che non siano state salvate su un dispositivo di archiviazione permanente.
In Rust, le variabili e le costanti sono fondamentali per memorizzare e manipolare i dati nel programma. La gestione di queste entità avviene in modo preciso e sicuro, grazie al sistema di tipizzazione del linguaggio.
Per iniziare, una variabile viene dichiarata con la parola chiave *let*. Per default, le variabili sono immutabili, il che significa che una volta assegnato un valore, non possono essere modificate. Tuttavia, se hai bisogno di una variabile che possa cambiare valore, puoi dichiararla come mutabile utilizzando la parola chiave *mut*. Osserviamo un esempio di variabile immutabile:

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

La variabile *x* è immutabile, quindi non può essere modificata dopo la sua dichiarazione. Se provassi a fare *x = 10;* subito dopo, il compilatore darebbe un errore. Passiamo adesso a un esempio di variabile mutabile:

```
fn main() {
    let mut y = 10;
    println!("Il valore iniziale di y è: {}", y);
    y = 20;
    println!("Il valore modificato di y è: {}", y);
}
```

Qui, la variabile *y* è dichiarata come mutabile, permettendo di cambiarne il valore da 10 a 20 senza problemi.
Le costanti, invece, vengono dichiarate con la parola chiave *const* e devono essere sempre immutabili. Inoltre, a differenza delle variabili, devono avere un tipo esplicitamente dichiarato e possono essere utilizzate ovunque nel programma, rendendole ideali per valori che non cambiano mai, come limiti o impostazioni configurabili:

```
fn main() {
    const MAX_PUNTI: u32 = 100_000;
    println!("Il numero massimo di punti è: {}", MAX_PUNTI);
}
```

In questo esempio, *MAX_PUNTI* è una costante che rappresenta un valore massimo di punti. Notiamo che, a differenza delle variabili, la costante ha un tipo (u32, un intero senza segno a 32 bit) specificato chiaramente.
Un altro concetto importante è lo *shadowing*, che consente di riutilizzare il nome di una variabile in uno *scope* successivo. Questo è utile quando hai bisogno di trasformare il valore di una variabile senza modificarne il tipo o la mutabilità. Con lo *shadowing*, una nuova variabile con lo stesso nome viene creata, nascondendo la precedente:

```
fn main() {
    let z = 6;
    let z = z + 1;
    let z = z * 2;
    println!("Il valore finale di z è: {}", z);
}
```

Quindi, il nome *z* viene riutilizzato più volte per calcolare un nuovo valore. Alla fine, *z* avrà il valore 14, risultato dell'operazione ((6 + 1) * 2).
Infine, è importante comprendere che tutte le variabili devono avere un tipo determinato, ma il compilatore spesso riesce a inferirlo automaticamente. Tuttavia, puoi dichiarare esplicitamente il tipo quando necessario, come in questo esempio:

```
fn main() {
    let a: f64 = 3.14;  // dichiarazione di una variabile di tipo floating point
    println!("Il valore di a è: {}", a);
}
```

Dove, *a* è dichiarato come f64, il tipo a virgola mobile a 64 bit, utilizzato per rappresentare numeri con la parte decimale.

Capire questi principi è essenziale per lavorare in modo efficace con Rust, dato che la gestione della mutabilità, la sicurezza nel trattamento dei dati e la corretta tipizzazione sono al cuore della filosofia del linguaggio. Questi concetti ti permettono di scrivere codice che non solo è sicuro e privo di errori legati alla memoria, ma anche chiaro e facilmente mantenibile.

### *Tipologia di variabili*

Come anticipato, le variabili costituiscono uno degli elementi fondamentali. Esse sono contenitori di dati che consentono ai programmatori di memorizzare e manipolare informazioni durante l'esecuzione di un programma. Ogni variabile ha un tipo di dato associato, che determina il tipo di valore che può essere memorizzato al suo interno e le operazioni che possono essere eseguite su di esso.
La tipologia delle variabili è un aspetto cruciale nella programmazione, in quanto definisce la natura dei dati che possono essere immagazzinati e manipolati all'interno del programma.
In Rust, le variabili possono essere suddivise in diverse tipologie in base ai tipi di dati che rappresentano. Ogni variabile è fortemente tipizzata, il che significa che il tipo di una variabile è noto al momento della compilazione, permettendo al compilatore di eseguire controlli di sicurezza e ottimizzazioni. Le principali tipologie di variabili includono:

- ***Tipi scalari***: comprendono i tipi base come numeri interi (i32, u32, i64, ecc.), numeri a virgola mobile (f32, f64), booleani (bool), e caratteri (char). Questi tipi rappresentano singoli valori:

```
fn main() {
    let intero: i32 = 42;
    let float: f64 = 3.14;
    let booleano: bool = true;
    let carattere: char = 'R';

    println!("Intero: {}, Float: {}, Booleano: {}, Carattere: {}", intero, float, booleano,
```

```
carattere);
}
```

Abbiamo quindi variabili che rappresentano un numero intero, un numero a virgola mobile, un valore booleano, e un carattere.

- ***Tipi composti***: combinano più valori in un singolo tipo, cioè le tuple e gli array. Le prime possono contenere valori di tipi diversi, mentre gli array contengono valori dello stesso tipo:

```
fn main() {
    let tupla: (i32, f64, char) = (42, 3.14, 'R');
    let (x, y, z) = tupla;

    println!("La tupla contiene: {}, {}, {}", x, y, z);
}
```

Qui, la tupla *tupla* contiene un intero, un numero a virgola mobile e un carattere. I valori possono essere destrutturati nelle variabili x, y, e z. Vediamo adesso l'array:

```
fn main() {
    let array: [i32; 3] = [1, 2, 3];

    println!("Il primo elemento dell'array è: {}", array[0]);
    println!("L'array ha {} elementi.", array.len());
}
```

L'array contiene tre numeri interi. Gli elementi possono essere accessibili tramite indici, e la funzione *len()* ne restituisce la lunghezza.
Oltre ai tipi scalari e composti, Rust offre altre caratteristiche interessanti legate alle variabili, come il *pattern matching*, che consente di verificare e destrutturare i valori in base al loro schema. Eccone un esempio con le tuple:

```
fn main() {
    let tupla = (1, 0);

    match tupla {
```

```
        (0, y) => println!("Il primo valore è zero e il secondo è: {}", y),
        (x, 0) => println!("Il secondo valore è zero e il primo è: {}", x),
        _ => println!("Nessuno dei valori è zero"),
    }
}
```

Qui, utilizziamo *match* per controllare la tupla *tupla* e determinare il comportamento in base ai valori che contiene.
Un concetto fondamentale in Rust è la proprietà delle variabili, che definisce chi possiede il valore di una variabile e determina quando e come essa viene liberata dalla memoria. Quando una variabile è assegnata a un'altra di esse o viene passata a un'altra funzione, la proprietà può essere trasferita, e la variabile originale non è più utilizzabile, evitando così errori legati alla gestione della memoria:

```
fn main() {
    let s1 = String::from("Ciao");
    let s2 = s1;  // La proprietà di s1 è trasferita a s2

    // println!("{}", s1); // Questo darà errore perché s1 non è più valido
    println!("{}", s2);  // Questo è valido
}
```

Ed ecco che, come anticipato, la proprietà della stringa *s1* viene trasferita a *s2*. Dopo questo passaggio, *s1* non è più accessibile.
Rust introduce anche il concetto di *borrow* (prestito), che consente di accedere ai dati di una variabile senza trasferirne la proprietà. Può essere utilizzato in due modi: prestito immutabile e prestito mutabile. Il primo permette di leggere i dati senza modificarli, mentre l'altro consente di modificarli:

```
fn main() {
    let mut s = String::from("Ciao");

    modifica(&mut s);  // Prestito mutabile
    println!("{}", s);
}
```

```
fn modifica(s: &mut String) {
    s.push_str(", mondo!");
}
```

In questo esempio, la stringa *s* viene passata alla funzione *modifica* tramite un prestito mutabile. Questo permette alla funzione di modificare il contenuto di *s* senza trasferirne la proprietà. Vediamo adesso il prestito immutabile:

```
fn main() {
    let s = String::from("Ciao");

    let s_modificata = modifica(&s);  // Prestito immutabile
    println!("{}", s_modificata);
}

fn modifica(s: &String) -> String {
    let mut nuova_stringa = s.clone();  // Crea una copia mutabile
    nuova_stringa.push_str(", mondo!");
    nuova_stringa
}
```

In questo caso *modifica* prende in prestito *s* in modo immutabile. La funzione *modifica* restituisce una nuova stringa che è una versione modificata dell'originale. La variabile *s* originale non viene mutata, mantenendo il prestito immutabile.
Per adesso abbiamo solo introdotto questi concetti, li rivedremo più avanti nel dettaglio.
Oltre alle variabili che abbiamo già discusso, come gli interi e i numeri a virgola mobile, Rust ne offre anche altre tipologie utili per gestire diverse esigenze di programmazione:

- ***Le stringhe***: sono più complesse rispetto a molti altri linguaggi a causa dell'accento posto sulla sicurezza e l'efficienza. Ce ne sono principalmente due tipi: *&str*, conosciute come stringhe *slice*, e *String*, che sono stringhe allocate dinamicamente. La prima è una vista immutabile su una stringa, di solito un suo

letterale. Invece, *String* è un tipo di dato che permette la sua gestione dinamica, con la possibilità di essere mutata e ampliata. Esempio con *&str*:

```
fn main() {
    let s_slice: &str = "Ciao, mondo!";
    println!("{}", s_slice);
}
```

Come visto, *s_slice* è una stringa *slice* che punta a una stringa immutabile, utile per operazioni leggere e quando non è necessario modificare la stringa. Esempio con *String*:

```
fn main() {
    let mut s_string = String::from("Ciao");
    s_string.push_str(", mondo!");
    println!("{}", s_string);
}
```

Qui, *s_string* è una stringa *String* che inizia con il valore "Ciao" e poi viene modificata aggiungendo ", mondo!". Questa mutabilità e gestione dinamica rendono *String* adatta per stringhe che necessitano di essere modificate durante l'esecuzione del programma.

- ***I booleani***: sono rappresentati dal tipo *bool*, che può assumere solo due valori: *true* o *false,* cioè vero o falso. Questo tipo è utilizzato principalmente nelle condizioni di controllo del flusso, come in strutture *if, while*, o nei pattern matching:

```
fn main() {
    let is_active: bool = true;

    if is_active {
        println!("Il sistema è attivo.");
    } else {
        println!("Il sistema è inattivo.");
    }
```

```
}
```

Come visto, *is_active* è un booleano che controlla il flusso del programma attraverso una struttura *if-else*.
Un'altra caratteristica interessante è la gestione dei valori nulli. In molti linguaggi, il concetto di "null" può portare a errori noti come *null pointer exceptions*. Rust evita questi problemi attraverso il tipo *Option<T>*, che rappresenta un valore che può essere presente (Some(T)) o assente (None). Questo meccanismo costringe il programmatore a gestire esplicitamente i casi in cui un valore potrebbe essere nullo, riducendo così la possibilità di errori:

```
fn main() {
    let some_value: Option<i32> = Some(42);
    let no_value: Option<i32> = None;

    match some_value {
        Some(x) => println!("Il valore è: {}", x),
        None => println!("Non c'è valore."),
    }

    match no_value {
        Some(x) => println!("Il valore è: {}", x),
        None => println!("Non c'è valore."),
    }
}
```

Nel programma, *some_value* contiene un valore (Some(42)), mentre *no_value* è *None*, il che rappresenta l'assenza di un valore. La gestione del tipo *Option* obbliga a considerare entrambe le possibilità, migliorando la sicurezza e la stabilità del codice.
Infine, Rust supporta i tipi unitari rappresentati da (), che è un tipo speciale utilizzato principalmente quando non è necessario restituire alcun valore da una funzione o un'espressione. Questo può essere visto come l'equivalente di *void* in altri linguaggi, ma viene trattato come un dato vero e proprio:

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

fn stampa_messaggio() {
    println!("Questa funzione non restituisce nulla.");
}
```

La funzione *stampa_messaggio* non restituisce alcun valore, quindi il suo tipo di ritorno è implicitamente (). Questo è utile quando l'unico scopo di una funzione è eseguire un'azione, come stampare un messaggio sulla console.
Ti ho descritto le principali tipologie di variabili in Rust, ma ci sono ancora alcuni concetti e tipi importanti che non abbiamo esplorato completamente. Ecco alcune altre caratteristiche significative relative a questi argomenti che verranno approfondite in seguito:

- *Variabili immutabili e mutabili*: in Rust, le variabili sono immutabili per default, il che significa che una volta assegnato un valore, questo non può essere cambiato. Tuttavia, se si vuole una variabile mutabile, bisogna dichiararla esplicitamente con la parola chiave *mut.* L'immutabilità per default è una caratteristica importante che incoraggia la scrittura di codice sicuro e privo di errori:

```
fn main() {
    let x = 5; // Immutabile
    // x = 6; // Questo darà un errore

    let mut y = 5; // Mutabile
    y = 6; // Questo è valido
    println!("Il valore di y è: {}", y);
}
```

- *Shadowing (Ombreggiatura)*: è possibile dichiarare una nuova variabile con lo stesso nome di un'altra. Questa mette in ombra quella precedente, permettendo una sorta di ridefinizione senza dover utilizzare la mutabilità. Ciò può essere utile, ad esempio, quando si vuole modificare il tipo di dato di una variabile o applicare

trasformazioni ai suoi valori:

```
fn main() {
    let x = 5;
    let x = x + 1;
    let x = x * 2;

    println!("Il valore di x è: {}", x); // Stampa 12
}
```

- *Enumerazioni (enum)*: sono un tipo di dato personalizzato che permette di definire una variabile che può essere uno tra vari valori possibili. Questo è molto utile per rappresentare dati che possono assumere una varietà di stati definiti:

```
enum Direzione {
    Su,
    Giù,
    Sinistra,
    Destra,
}

fn main() {
    let movimento = Direzione::Su;
    match movimento {
        Direzione::Su => println!("Stai andando su!"),
        Direzione::Giù => println!("Stai andando giù!"),
        Direzione::Sinistra => println!("Stai andando a sinistra!"),
        Direzione::Destra => println!("Stai andando a destra!"),
    }
}
```

- *Strutture (struct)*: sono simili alle classi in altri linguaggi. Permettono di raggruppare diversi tipi di variabili sotto un unico tipo di dato, rendendolo ideale per rappresentare oggetti complessi con molte proprietà.

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

```
fn main() {
    let punto = Punto { x: 10, y: 20 };
    println!("Punto: ({}, {})", punto.x, punto.y);
}
```

- *Smart Pointers (Puntatori intelligenti)*: permettono una gestione più flessibile e sicura della memoria rispetto ai puntatori tradizionali. NB. la differenza principale tra i due risiede nella gestione della memoria e della proprietà. Quelli tradizionali, in linguaggi come C e C++, puntano direttamente a una posizione di memoria e ne richiedono una gestione manuale per evitare perdite o accessi a memoria non valida. In Rust, tuttavia, i puntatori intelligenti (smart pointers) come *Box, Rc,* e *RefCell* forniscono funzionalità aggiuntive come il conteggio automatico dei riferimenti e la gestione sicura della mutabilità e della proprietà. Questi puntatori intelligenti non solo puntano a un oggetto in memoria, ma implementano anche la logica di gestione delle proprietà e della memoria, garantendo che questa venga liberata in modo sicuro quando non è più utilizzata, grazie alle regole di "ownership" di Rust. Ciò riduce notevolmente la possibilità di errori come doppia liberazione della memoria o puntatori dangling. Inoltre, sono cruciali per gestire situazioni di proprietà condivisa o modifiche interne senza infrangere le regole del *borrowing*:

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

fn main() {
    let valore = Rc::new(5);
    let clone = Rc::clone(&valore);
    println!("Valore: {}, Cloni: {}", valore, Rc::strong_count(&valore));
}
```

Ogni tipo di variabile è progettato per massimizzare la sicurezza e l'efficienza, riducendo al minimo gli errori comuni legati alla gestione della memoria e del controllo dei dati.

### *Scope delle variabili*

Lo *scope* di una variabile rappresenta la porzione di codice in cui essa è valida e utilizzabile. Quando ne dichiariamo una, questa entra in uno specifico scope, che termina quando il programma esce dal blocco in cui la variabile è stata definita. Alla fine di quello scope, la variabile viene automaticamente distrutta e la memoria

associata ad essa viene liberata grazie al sistema di ownership e borrow.
Ad esempio, se dichiari una variabile all'interno di una funzione, quella variabile sarà accessibile solo all'interno di quella funzione e smetterà di esistere quando la funzione termina. Lo scope permette di avere un controllo chiaro e preciso sulla durata delle variabili, riducendo il rischio di errori di accesso a memoria non valida. Considera questo esempio:

```
fn main() {
    let x = 5; // x entra nello scope
    {
        let y = 10; // y entra in un sottoscope
        println!("x: {}, y: {}", x, y); // x e y sono accessibili qui
    }
    // y è fuori dallo scope e non più accessibile
    println!("x: {}", x); // x è ancora accessibile
    // y non può più essere usato qui
}
```

Nel blocco più interno, la variabile *y* è accessibile solo lì, mentre *x*, dichiarata in uno scope più esterno, è visibile sia nel blocco interno sia in quello esterno. Questo meccanismo aiuta a mantenere il codice sicuro e prevedibile.
Il *lifetime* o ciclo di vita, è strettamente legato allo scope, ma riguarda più specificamente la durata di una *reference*. NB. una reference è un modo sicuro e controllato per accedere a una variabile senza trasferirne la proprietà. Consente di prendere in prestito una variabile per leggerla o modificarla, rispettando le regole di *borrow checker* del linguaggio. Ne esistono due tipi: immutabili (&T), che permettono solo la lettura, e mutabili (&mut T), che consentono modifiche. Rust garantisce che ci sia solo una reference mutabile attiva per una variabile, oppure più reference immutabili contemporaneamente. Questo sistema evita i tipici problemi di concorrenza e garantisce la sicurezza della memoria senza bisogno di un garbage collector. Il *lifetime* si riferisce al periodo di tempo in cui una reference rimane valida. Mentre lo scope gestisce la visibilità di una variabile, il lifetime stabilisce per quanto tempo una reference può essere utilizzata senza causare violazioni di accesso alla memoria. In Rust, ogni reference ha un lifetime e il compilatore usa le sue annotazioni per assicurarsi che la prima non sopravviva oltre il tempo in cui i dati a

cui fa riferimento sono ancora validi. Un esempio potrebbe essere:

```
fn prendi_riferimento<'a>(s: &'a str) -> &'a str {
    s
}

fn main() {
    let stringa = String::from("ciao");
    let riferimento = prendi_riferimento(&stringa);
    println!("{}", riferimento); // valido perché stringa vive più a lungo di riferimento
}
```

La funzione *prendi_riferimento* accetta una reference con un lifetime *'a* e ne restituisce un'altra con lo stesso lifetime. Questo vincolo assicura che il riferimento restituito non possa essere utilizzato oltre il tempo in cui stringa è valida. I lifetime garantiscono la sicurezza della memoria durante l'uso di reference, prevenendo errori come dangling pointers.
Lo scope delle funzioni è simile a quello delle variabili. Quando viene definita, il suo scope include il corpo della funzione stessa e le sue variabili locali. Tuttavia, le funzioni possono anche essere definite all'interno di altre, il che crea scope nidificati, limitando ulteriormente la visibilità delle variabili.

### *Let e altre dichiarazioni*

L'istruzione *let* è fondamentale per la dichiarazione e l'inizializzazione delle variabili. È un costrutto versatile che permette di gestire la maggior parte delle operazioni di assegnazione di valori a variabili, sia che queste siano immutabili o mutabili, che siano di tipo semplice o complesso.
In Rust, le variabili sono immutabili per impostazione predefinita. Questo significa che una volta assegnato un valore a una variabile tramite *let*, quel valore non può essere cambiato. Questa immutabilità predefinita è una scelta progettuale volta a migliorare la sicurezza del codice, impedendo modifiche involontarie ai dati. Ad esempio, se dichiari una variabile con *let x = 5;,* non potrai poi fare *x = 6;* perché verrà segnalato un errore di compilazione:

```
fn main() {
    let x = 5;
    // x = 6; // Questo genererà un errore di compilazione
    println!("Il valore di x è: {}", x);
}
```

Se invece hai bisogno di una variabile che possa essere modificata, puoi dichiararla esplicitamente come mutabile aggiungendo la parola chiave *mut* subito dopo *let*. Questo permette di cambiare il valore della variabile in seguito, mantenendo però la stessa proprietà e il controllo rigoroso da parte del compilatore:

```
fn main() {
    let mut y = 5;
    y = 6; // Questo è valido
    println!("Il valore di y è: {}", y);
}
```

L'istruzione *let* supporta anche il cosiddetto *shadowing*, che consente di dichiarare una nuova variabile con lo stesso nome di una già esistente, "oscurando" quella precedente. Questo può essere utile quando si desidera riutilizzare un nome di variabile per un valore calcolato o trasformato senza doverlo mutare:

```
fn main() {
    let z = 5;
    let z = z + 1; // Shadowing: z assume un nuovo valore
    let z = z * 2;
    println!("Il valore di z è: {}", z); // Stampa 12
}
```

Un'altra peculiarità dell'istruzione *let* è la possibilità di omettere il tipo di dato della variabile, lasciando che sia il compilatore a dedurlo automaticamente. Rust è un linguaggio con un sistema di tipi forte e statico, ma grazie alla deduzione dei tipi, spesso non è necessario specificarli esplicitamente:

```
fn main() {
    let a = 10; // Il compilatore deduce che a è un intero (i32)
```

```
    let b = 3.14; // Il compilatore deduce che b è un numero a virgola mobile (f64)
    println!("a: {}, b: {}", a, b);
}
```

Inoltre, l'istruzione *let* può essere utilizzata per destrutturare tuple, array o strutture complesse, assegnando direttamente i valori a più variabili contemporaneamente. Questo rende il codice più conciso e leggibile:

```
fn main() {
    let (c, d) = (30, 40); // Assegna 30 a c e 40 a d
    println!("c: {}, d: {}", c, d);
}
```

*let* può anche essere utilizzata insieme all'operatore _ per ignorare valori che non sono necessari. Questo è particolarmente utile quando si lavora con funzioni che restituiscono più valori, ma solo alcuni di essi sono rilevanti per il contesto:

```
fn main() {
    let (_, e) = (50, 60); // Ignora il primo valore e assegna 60 a e
    println!("Il valore di e è: {}", e);
}
```

L'istruzione *let*, quindi, non è solo un semplice strumento per la dichiarazione di variabili, ma un potente costrutto che offre controllo, sicurezza e flessibilità nella gestione dei dati all'interno dei programmi. La sua versatilità è una delle caratteristiche che rendono Rust un linguaggio robusto e sicuro, permettendo di scrivere codice chiaro, efficiente e privo di errori comuni.
In poche parole *let* è essenziale per dichiarare variabili, ma ci sono altre parole chiave e costrutti legati alla dichiarazione e alla gestione delle variabili che svolgono ruoli specifici. Ad esempio, Rust non ha esattamente parole chiave come *ref, out, readonly,* o *public* come in altri linguaggi come C# o C++, ma ha concetti simili che vengono implementati attraverso diverse parole chiave e costrutti.
Una delle parole chiave importanti è *const*, utilizzata per dichiarare costanti. A differenza delle variabili dichiarate con *let*, le costanti devono avere un tipo di dato

esplicito e il loro valore deve essere noto al momento della compilazione. Le costanti sono sempre immutabili e il loro valore non può essere cambiato dopo la dichiarazione. Poiché sono valutate al momento della compilazione, possono essere utilizzate in contesti dove le variabili non possono, come per definire dimensioni di array statici:

```
const MAX_VALORE: i32 = 100;

fn main() {
    println!("Il valore massimo è: {}", MAX_VALORE);
}
```

Un altro concetto chiave è quello della visibilità, che viene gestito attraverso le parole chiave *pub* e *priv*. Per default, tutti i moduli, strutture e funzioni sono privati, ovvero visibili solo all'interno del modulo in cui sono definiti. Se si desidera rendere una funzione, una struttura o un campo di una struttura accessibile dall'esterno, si utilizza la parola chiave *pub*. Questo concetto è simile a *public* in C# e altri linguaggi. La gestione della visibilità è fondamentale per mantenere l'incapsulamento e limitare l'accesso non autorizzato ai dati:

```
mod mio_modulo {
    pub struct MiaStruttura {
        pub campo_pubblico: i32,
        campo_privato: i32,
    }

    impl MiaStruttura {
        pub fn nuova(campo_pubblico: i32, campo_privato: i32) -> MiaStruttura {
            MiaStruttura { campo_pubblico, campo_privato }
        }
    }
}

fn main() {
    let struttura = mio_modulo::MiaStruttura::nuova(10, 20);
    println!("Campo pubblico: {}", struttura.campo_pubblico);
    // println!("Campo privato: {}", struttura.campo_privato); // Questo darà errore perché il
```

```
campo è privato
}
```

In Rust, non esistono parole chiave come *readonly* perché la natura immutabile delle variabili è già gestita direttamente dalla parola chiave *let.* Per garantire che un campo di una struttura sia immutabile una volta inizializzato, non c'è bisogno di una parola chiave aggiuntiva; basta non dichiararlo come *mut*.
Come già anticipato in precedenza, parlando di riferimenti e gestione dei puntatori, Rust utilizza il concetto di *borrowing* e *reference*. Qui, *&* è utilizzato per creare riferimenti immutabili, mentre *&mut* per riferimenti mutabili. Il borrowing consente di passare dati senza trasferirne la proprietà, un meccanismo fondamentale per la sicurezza della memoria. Anche se non è una parola chiave di dichiarazione in senso stretto, è un concetto cruciale per comprendere come Rust gestisce la proprietà e la mutabilità:

```
fn main() {
    let mut x = 5;
    let y = &x; // riferimento immutabile
    let z = &mut x; // riferimento mutabile
    println!("y: {}", y);
    *z += 1;
    println!("z: {}", z);
}
```

Mentre Rust non dispone di una parola chiave *out* come in C#, dove i parametri possono essere dichiarati con l'intento di restituire valori modificati, il concetto viene gestito tramite i riferimenti mutabili. Se una funzione deve modificare un valore, si passa una variabile come riferimento mutabile, e la funzione può alterarne lo stato senza necessità di una parola chiave specifica.
Questi costrutti e parole chiave, anche se meno numerosi rispetto ad altri linguaggi, sono progettati per lavorare insieme in modo da fornire un forte controllo della mutabilità, visibilità e gestione della memoria, garantendo al contempo che il codice rimanga sicuro e performante.

### *Operatori e metodi*

Gli operatori sono simboli speciali utilizzati per eseguire operazioni su variabili e valori. Quelli aritmetici includono addizione (+), sottrazione (-), moltiplicazione (*), divisione (/) e modulo (%), utilizzati per eseguire calcoli numerici. Gli operatori logici, come AND (&&), OR (||) e NOT (!), sono utilizzati per valutare condizioni booleane, restituendo *true* o *false* a seconda delle condizioni specificate. Gli operatori di confronto, come uguale a (==), diverso da (!=), maggiore di (>), minore di (<), maggiore o uguale a (>=) e minore o uguale a (<=), sono utilizzati per confrontare i valori e determinare le relazioni tra di essi.

Gli operatori di assegnazione, come (=) e gli operatori composti come +=, -=, *=, /= e %=, servono per assegnare valori a variabili. I primi, a differenza di altri linguaggi, per prevenire errori legati alle assegnazioni non intenzionali in espressioni condizionali, non restituiscono un valore.

Oltre a questi, Rust supporta anche operatori più avanzati come quelli per la manipolazione dei bit (&, |, ^, <<, >>) e il pattern matching.

| Operatore | Utilizzo | Esempio |
|---|---|---|
| = | *Attribuisce un valore (immutabile) a una variabile* | `let x = 3;` |
| == | *Confronta l'uguaglianza tra valori* | `if x == 3 { ... }` |
| != | *Confronta la differenza tra valori* | `if x != 3 { ... }` |
| < > | *Confronta se un valore è minore o maggiore* | `if x < 3 { ... }` |
| >= <= | *Come sopra, includendo l'uguaglianza* | `if x >= 3 { ... }` |

Rust utilizza il modello di immutabilità di default per le variabili. Ciò significa che quando si dichiara una variabile con *let*, come nell'esempio *let x = 3;,* la variabile è immutabile e non può essere modificata. Se si desidera dichiarare una variabile mutabile, è necessario usare il modificatore *mut,* ad esempio: *let mut x = 3;.*

Gli operatori logici di confronto, come == e !=, vengono usati con il supporto per tipi primitivi come interi (i32, u32, ecc.) e altre strutture di dati che implementano il trait *PartialEq* per consentire il confronto.

Rust non consente l'assegnazione di valori all'interno di condizioni, come avviene in altri linguaggi. Un'istruzione come *if (x = 3)* genererebbe un errore. Questo comportamento è intenzionale per evitare errori comuni dovuti all'uso accidentale dell'operatore di assegnazione = invece del confronto ==.

Andiamo a vedere come si comportano gli operatori nel concreto:

```
use std::io;

fn main() {
    println!("Inserisci il tuo peso:");

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

    // Converte l'input in un numero intero
    let peso: i32 = peso_input.trim().parse().expect("Errore nella conversione del peso");

    // Controlla il peso e fornisce il feedback appropriato
    if peso >= 70 {
        println!("Fai attenzione alla dieta!");
    } else {
        println!("Ogni tanto mangia...");
    }
}
```

Utilizziamo *println!* per stampare messaggi sulla console, mentre l'input dell'utente viene letto usando *io::stdin().read_line*, che riempie una stringa mutabile (*mut peso_input*). Questo viene quindi convertito in un numero intero (i32) con il metodo *trim().parse(),* che rimuove eventuali spazi bianchi e converte la stringa.
Infine, utilizziamo *if* per confrontare il valore e dare un feedback.
Oltre agli operatori, per plasmare il codice abbiamo i *metodi*, cioè blocchi di codice raggruppati che eseguono una specifica azione o calcolo. Sono elementi fondamentali del linguaggio che permettono di organizzare il codice in modo modulare e riutilizzabile.
In Rust, i metodi sono funzioni associate a un particolare tipo di dato o a una struttura. A differenza delle funzioni normali, che sono definite al di fuori di un

tipo, i metodi sono definiti all'interno di un *impl block*, che è un blocco di implementazione per un tipo specifico. I metodi consentono di interagire con i dati di una struttura o di un tipo, permettendo di manipolarli o accedere alle loro proprietà in modo più strutturato e sicuro.

Per dichiarare un metodo, si utilizza la parola chiave *fn* all'interno di un blocco *impl,* e il primo parametro è solitamente *self,* che rappresenta l'istanza su cui il metodo viene chiamato. A seconda di come è dichiarato, *self* può essere preso per valore, per riferimento immutabile (*&self*), o mutabile (*&mut self*), permettendo rispettivamente di consumare l'oggetto, di leggerne lo stato, o di modificarlo.

Ad esempio, supponiamo di avere una struttura *Rettangolo* che rappresenta un rettangolo con una larghezza e un'altezza. Possiamo definire un metodo per calcolare l'area del rettangolo all'interno di un *impl block*:

```
struct Rettangolo {
    larghezza: u32,
    altezza: u32,
}

impl Rettangolo {
    // Metodo che calcola l'area del rettangolo
    fn area(&self) -> u32 {
        self.larghezza * self.altezza
    }
}

fn main() {
    let mio_rettangolo = Rettangolo {
        larghezza: 30,
        altezza: 50,
    };

    println!("L'area del rettangolo è: {}", mio_rettangolo.area());
}
```

Il metodo *area* è associato alla struttura *Rettangolo* e utilizza il riferimento immutabile *&self* per accedere ai campi *larghezza* e *altezza* della struttura. Quando

chiamiamo *mio_rettangolo.area(),* Rust invoca il metodo *area* sull'istanza *mio_rettangolo* e restituisce il risultato della moltiplicazione di larghezza per altezza. Detto ciò andiamo a vedere come si suddividono i metodi:

*Metodi associati*: sono l'equivalente dei metodi statici in altri linguaggi. Non prendono self come parametro e vengono chiamati sul tipo stesso, piuttosto che su un'istanza del tipo. In poche parole stiamo parlando di un istruzione simile a quella dei costruttori, dei metodi statici e dei metodi di istanza, sebbene Rust li implementi in modo un po' diverso rispetto a linguaggi come C# o Java, *infatti non esiste un costruttore speciale predefinito come in altri linguaggi orientati agli oggetti*. Tuttavia, è comune definire un metodo associato, di solito chiamato *new*, che funge da costruttore per una struttura. Questo metodo associato non richiede un'istanza dell'oggetto e viene chiamato direttamente sul tipo:

```
struct Rettangolo {
    larghezza: u32,
    altezza: u32,
}

impl Rettangolo {
    // Metodo associato che funge da costruttore
    fn new(larghezza: u32, altezza: u32) -> Rettangolo {
        Rettangolo { larghezza, altezza }
    }
}

fn main() {
    let mio_rettangolo = Rettangolo::new(40, 60);
    println!("Larghezza: {}, Altezza: {}", mio_rettangolo.larghezza, mio_rettangolo.altezza);
}
```

In questo esempio, il metodo *new* è il costruttore che crea e restituisce una nuova istanza di *Rettangolo*. Questo è il modo idiomatico di costruire oggetti in Rust.

*Metodi di istanza*: operano su un'istanza specifica di un tipo e quindi prendono *self*

come primo parametro. Possono essere chiamati su un oggetto specifico di quel tipo e possono essere mutabili o immutabili, a seconda di come si definisce self, *&mut self* o *&self* rispettivamente:

```
impl Rettangolo {
    // Metodo di istanza che calcola l'area
    fn area(&self) -> u32 {
        self.larghezza * self.altezza
    }
}

fn main() {
    let mio_rettangolo = Rettangolo::new(40, 60);
    println!("L'area del rettangolo è: {}", mio_rettangolo.area());
}
```

In tal caso, *area* è un metodo di istanza che viene chiamato su *mio_rettangolo*, e utilizza i valori di larghezza e altezza associati a quella particolare istanza per calcolare l'area.
Quindi, mentre Rust non ha costruttori o metodi statici nel senso tradizionale, i metodi associati e i metodi di istanza forniscono funzionalità simili in un modo che si adatta al modello di proprietà e gestione della memoria.
Detto ciò, dobbiamo anche aggiungere che in questo linguaggio esistono concetti simili ai metodi astratti, asincroni, generici e di estensione, sebbene vengano implementati e utilizzati in modi specifici. Di seguito abbiamo una panoramica di ciascuno di questi concetti.

*Metodi Astratti e di Estensione*: Rust non ha metodi astratti nel senso tradizionale che troviamo in linguaggi come Java o C#, dove una classe può definire un metodo senza implementazione, costringendo le classi derivate a fornire la propria implementazione. Tuttavia, Rust permette la definizione di metodi senza implementazione nei trait. I *trait* sono simili alle interfacce e definiscono funzionalità che i tipi possono implementare. Un metodo definito in un trait senza

un corpo è, di fatto, un metodo astratto che deve essere implementato da qualsiasi tipo che voglia "implementare" quel trait:

```
trait Forma {
    fn area(&self) -> f64; // Metodo astratto
}

struct Cerchio {
    raggio: f64,
}

impl Forma for Cerchio {
    fn area(&self) -> f64 {
        std::f64::consts::PI * self.raggio * self.raggio
    }
}
```

Qui, *area* è un metodo astratto definito nel trait *Forma*, che viene poi implementato dalla struttura *Cerchio*.
Se vuoi aggiungere nuovi metodi a un tipo esistente, crei un *trait* che definisce questi metodi e poi lo implementi per il tipo in questione. Questo approccio consente di estendere i tipi senza modificarli direttamente, in modo simile ai "metodi di estensione" in C#:

```
trait Estensione {
    fn descrizione(&self) -> String;
}

impl Estensione for i32 {
    fn descrizione(&self) -> String {
        format!("Questo numero è {}", self)
    }
}

fn main() {
    let numero = 42;
    println!("{}", numero.descrizione());
}
```

Quindi, estendiamo il tipo i32 con un nuovo metodo *descrizione* utilizzando un *trait*.
La combinazione di metodi astratti tramite trait, metodi asincroni, generici e di estensione offre potenti strumenti per lo sviluppo di software.

*Metodi Asincroni*: Rust supporta la programmazione asincrona tramite la parola chiave *async*, che può essere utilizzata per definire funzioni e metodi asincroni. Un metodo asincrono restituisce un "futuro" (*Future*), che rappresenta un valore che sarà disponibile in un momento successivo. Questi metodi sono utili per gestire operazioni che richiedono tempo, come I/O, senza bloccare l'esecuzione del programma.

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

struct Downloader;

impl Downloader {
    async fn download(&self, url: &str) {
        println!("Inizio download da {}", url);
        sleep(Duration::from_secs(2)).await; // Simula un'attesa di 2 secondi
        println!("Download completato da {}", url);
    }
}

#[tokio::main]
async fn main() {
    let downloader = Downloader;
    downloader.download("https://example.com").await;
}
```

In questo esempio, il metodo *download* è asincrono, permettendo al programma di continuare a funzionare mentre aspetta il completamento dell'operazione simulata.

*Metodi Generici*: funzionano con qualsiasi tipo che soddisfi certi vincoli. I metodi generici sono utili per scrivere codice riutilizzabile che può operare su più tipi di

dati:

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

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

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

fn main() {
    let contenitore = Contenitore::nuovo(42);
    println!("Il valore è: {}", contenitore.ottenere());
}
```

Qui, *Contenitore* è una struttura generica e il metodo nuovo può accettare qualsiasi tipo T, rendendolo molto flessibile.

***Macro***

Le macro sono strumenti potenti che permettono di scrivere codice che ne genera altro durante la compilazione. Servono a ridurre le ripetizioni e a semplificare pattern complessi. Esistono due tipi principali di macro: quelle di procedura e quelle di dichiarazione che sono le più comuni, utilizzano il simbolo ! alla fine del loro nome, come la macro *println!* la quale stampa una stringa sullo schermo:

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

Le macro di procedura permettono di trasformare e generare codice durante la compilazione. A differenza delle precedenti, vengono definite tramite una funzione

che prende del codice come input e restituisce del codice come output. Sono utilizzate per manipolare sintassi complesse e vengono spesso applicate su funzioni, strutture o altri elementi di codice.
Le macro di procedura sono spesso utilizzate per implementare comportamenti come derive, attributi personalizzati o modificatori di funzioni. Un esempio comune è la macro *#[derive],* che automaticamente genera implementazioni di trait come *Debug* o *Clone* per una struttura (approfondiremo questi concetti più avanti):

```
#[derive(Debug)]
struct Persona {
    nome: String,
    età: u32,
}

fn main() {
    let p = Persona {
        nome: String::from("Alice"),
        età: 30,
    };
    println!("{:?}", p); // Stampa la struttura con i suoi valori
}
```

Le macro di procedura possono essere definite usando librerie come *proc-macro*, favorendo una manipolazione sofisticata del codice sorgente.
Permettono anche di creare istruzioni flessibili che possono accettare diversi tipi e numeri di argomenti. Le macro sono utilizzate in situazioni dove la flessibilità o la generazione automatica di codice sono importanti. Ad esempio, se vuoi scrivere una funzione che gestisca più tipi di dati o pattern comuni, le macro ti permettono di farlo senza dover scrivere manualmente ogni variante.
Puoi anche definire macro personalizzate:

```
macro_rules! saluta {
    ($name:expr) => {
        println!("Ciao, {}!", $name);
    };
}
```

```
fn main() {
    saluta!("Amico");
}
```

In questo caso, la macro *saluta!* genera codice che utilizza *println!* per salutare l'argomento passato.

Di seguito le principali macro di dichiarazione utilizzabili in Rust:

| Macro | Utilizzo | Esempio |
|---|---|---|
| println! | *Stampa un messaggio su console con formattazione* | println!("Ciao, {}!", "mondo"); // Stampa: Ciao, mondo! |
| format! | *Crea una stringa formattata senza stamparla* | let s = format!("Il numero è: {}", 42); // Restituisce: Il numero è: 42 |
| vec! | *Crea un Vec (vettore) con valori predefiniti* | let v = vec![1, 2, 3]; // Un vettore contenente 1, 2, 3 |
| include_str! | *Include il contenuto di un file come stringa* | let contenuto = include_str!("file.txt"); // Legge il contenuto di file.txt come stringa |
| dbg! | *Stampa a video il valore di un'espressione per debug* | let x = 2; dbg!(x * 2); // Stampa: [src/main.rs:10] 4 = 4 |
| panic! | *Termina l'esecuzione del programma con un messaggio* | panic!("Errore irreversibile"); // Genera un errore e termina il programma |
| assert! | *Verifica che una condizione sia vera* | assert!(1 + 1 == 2); // Passa, altrimenti il programma si arresta |
| assert_eq! | *Verifica che due valori siano uguali* | assert_eq!(3 * 3, 9); // Passa se entrambi i valori sono uguali, altrimenti genera un errore |
| todo! | *Placeholder per il codice non ancora implementato* | fn funzione() { todo!("Da implementare"); } // Genera un errore con messaggio |
| println! | *Stampa un messaggio su console con formattazione* | println!("Ciao, {}!", "mondo"); // Stampa: Ciao, mondo! |
| format! | *Crea una stringa formattata senza stamparla* | let s = format!("Il numero è: {}", 42); // Restituisce: Il numero è: 42 |
| vec! | *Crea un Vec (vettore) con valori predefiniti* | let v = vec![1, 2, 3]; // Un vettore contenente 1, 2, 3 |
| include_str! | *Include il contenuto di un file come stringa* | let contenuto = include_str!("file.txt"); // Legge il contenuto di file.txt come stringa |

### *Ownership e borrowing*

I concetti di ownership e borrowing sono fondamentali in Rust e sono alla base della gestione della memoria sicura e senza garbage collector, come già detto più volte, una delle caratteristiche più distintive del linguaggio.

L'ownership si riferisce al modo in cui si gestisce la memoria di un programma. In

Rust, ogni valore ha un unico proprietario, che è la variabile alla quale è assegnato. Quando una variabile esce dal suo scope, cioè quando il suo blocco di codice termina, la memoria occupata dal valore che possiede viene automaticamente liberata. Questo elimina la necessità di gestire manualmente la deallocazione della memoria, prevenendo bug come i "dangling pointers" o le fughe di memoria. Per esempio:

```
fn main() {
    let s = String::from("Ciao");
    println!("{}", s); // La variabile `s` possiede la stringa "Ciao"
} // Qui `s` esce dallo scope e la memoria viene rilasciata
```

Un aspetto critico dell'ownership è che, quando una variabile viene assegnata a un'altra, la proprietà del valore viene trasferita. Dopo il trasferimento, la variabile originale non può più essere utilizzata, evitando così accessi concorrenti non sicuri. Ad esempio:

```
fn main() {
    let s1 = String::from("Ciao");
    let s2 = s1; // `s1` trasferisce la proprietà a `s2`
    // println!("{}", s1); // Questo causerebbe un errore perché `s1` non è più valido
    println!("{}", s2); // `s2` è ora il proprietario della stringa
}
```

Il borrowing invece permette di "prendere in prestito" un valore senza trasferirne la proprietà. Si possono prendere in prestito valori in modo immutabile o mutabile. Il primo consente di leggere il valore senza modificarlo, mentre l'altro permette di farlo, ma con la regola che ci può essere solo un prestito mutabile attivo alla volta, per garantire la sicurezza dell'accesso ai dati.
Ecco un esempio di prestito immutabile:

```
fn stampa_lunghezza(s: &String) {
    println!("La lunghezza della stringa è: {}", s.len());
}
```

```
fn main() {
    let s1 = String::from("Ciao");
    stampa_lunghezza(&s1); // `s1` viene preso in prestito, ma non perde la proprietà
    println!("{}", s1); // `s1` può ancora essere usato qui
}
```

Ed ecco invece un esempio di prestito mutabile:

```
fn aggiungi_punto(s: &mut String) {
    s.push('.');
}

fn main() {
    let mut s1 = String::from("Ciao");
    aggiungi_punto(&mut s1); // `s1` viene preso in prestito mutabilmente
    println!("{}", s1); // `s1` è stato modificato e può essere usato di nuovo
}
```

Il sistema di ownership e borrowing di Rust permette una gestione della memoria estremamente efficiente e sicura, prevenendo una vasta gamma di bug legati alla memoria che sono comuni in altri linguaggi, senza la necessità di un garbage collector. Questa caratteristica è particolarmente importante quando si lavora con codice concorrente, dove la sicurezza e la gestione accurata delle risorse sono cruciali.

Una delle regole essenziali da tenere a mente è che non possiamo avere contemporaneamente un prestito mutabile e uno o più prestiti immutabili sullo stesso dato. Questo vincolo è progettato per evitare condizioni di competizione (*race conditions*) e garantire che i dati non vengano modificati mentre vengono letti da altre parti del codice:

```
fn modifica(s: &mut String) {
    s.push_str(", mondo!");
}

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

```
    // Prestito mutabile per modificare la stringa
    modifica(&mut saluto);

    // Una volta terminato il prestito mutabile, possiamo prendere un prestito immutabile
    let len = saluto.len(); // Prestito immutabile per leggere la lunghezza

    println!("Il saluto è: {} con lunghezza {}", saluto, len);
}
```

Il prestito mutabile a *modifica* permette di cambiare il valore della stringa. Conclusa questa operazione, possiamo tranquillamente prendere un prestito immutabile per calcolare e stampare la lunghezza della stringa.
Un errore comune potrebbe sorgere quando si tenta di violare questa regola:

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

    let r1 = &saluto; // Prestito immutabile
    let r2 = &saluto; // Un altro prestito immutabile
    let r3 = &mut saluto; // Errore: prestito mutabile mentre esistono prestiti immutabili

    println!("{}, {}, e {}", r1, r2, r3);
}
```

Questo codice genererà un errore perché abbiamo tentato di prendere un prestito mutabile *r3* mentre *r1* e *r2* detengono ancora prestiti immutabili. Rust non permette questo tipo di conflitto, prevenendo potenziali bug.
Un altro aspetto fondamentale da considerare è la durata del prestito, o *lifetime*. Infatti questi hanno una durata limitata, determinata dallo scope in cui esistono. Rust applica automaticamente la verifica delle durate dei prestiti per garantire che i riferimenti non diventino mai non validi. Questo viene chiamato "verifica del lifetime", che impedisce riferimenti pendenti a dati che sono stati deallocati.
Un semplice esempio di violazione delle durate potrebbe essere:

```
fn main() {
    let r;
    {
```

```
        let x = 5;
        r = &x; // Errore: `x` non esiste più quando `r` viene utilizzato
    }
    println!("r: {}", r); // Tentativo di utilizzo di un riferimento non valido
}
```

In questo caso, *x* viene deallocato quando esce dallo scope interno, quindi il riferimento *r* sarebbe non valido. Rust previene questo tipo di errore, garantendo che i riferimenti rimangano sempre sicuri.
Infine, un concetto strettamente legato all'ownership e al borrowing è lo *slicing*, che permette di prendere una vista di una parte di un dato senza trasferirne la proprietà. Ad esempio, si può creare uno *slice* su una stringa:

```
fn main() {
    let saluto = String::from("Ciao, mondo!");
    let ciao = &saluto[0..4]; // Slice immutabile sulla stringa
    println!("Slice: {}", ciao); // Stampa "Ciao"
}
```

Lo slice *ciao* non possiede i dati, ma li prende in prestito in modo immutabile, rendendo possibile operare su parti dei dati senza trasferire la proprietà.
Questi concetti di ownership e borrowing in Rust sono essenziali per scrivere codice sicuro e privo di errori legati alla memoria, fornendo al tempo stesso flessibilità ed efficienza. Il linguaggio richiede che gli sviluppatori comprendano e rispettino queste regole, ma in cambio offre un controllo preciso e potente sulla gestione della memoria e la sicurezza del codice.

### *I dati*

Ogni programma manipola diversi tipi di dati, e queste operazioni avvengono nella memoria dei dispositivi, siano essi computer, smartphone o altri sistemi. In Rust, come in molti altri linguaggi, i dati vengono rappresentati in byte, con ciascun byte composto da otto bit, che possono assumere valori da 0 a 255. Questi blocchi di dati possono essere aggregati in unità più grandi come megabyte (un milione di

byte) o gigabyte (un miliardo di byte), quantità di memoria oggi ampiamente disponibili nei dispositivi moderni.

Ma cosa rappresentano questi byte? Possono rappresentare numeri, testo, immagini o qualsiasi altra informazione digitale. Nei linguaggi di programmazione, i byte sono associati a specifici tipi di dati, come numeri interi, valori booleani o stringhe di testo. In Rust, è fondamentale dichiarare in modo esplicito i tipi di dati delle variabili per garantire la sicurezza e la prevedibilità del comportamento del programma. Per esempio, sono inclusi tipi di dati come i32 per numeri interi a 32 bit, f64 per numeri in virgola mobile a 64 bit, e bool per valori booleani.

Rust adotta un approccio diverso rispetto ad altri linguaggi in termini di mutabilità. Per impostazione predefinita, tutte le variabili sono immutabili, il che significa che una volta assegnato un valore a una variabile, non può essere modificato. Questo design favorisce la sicurezza e la prevedibilità, evitando effetti collaterali indesiderati. Se c'è il bisogno di modificare una variabile dopo la sua creazione, è necessario dichiararla come mutabile usando il modificatore *mut*. Ad esempio: *let mut x = 5;* consente di cambiare il valore di *x* successivamente nel programma.

La distinzione tra dati mutabili e immutabili è essenziale. Un esempio pratico può essere la gestione di strutture come le *String* e le *&str* (stringhe immutabili). Un oggetto di tipo *String* è mutabile, il che significa che possiamo aggiungere, rimuovere o modificare i caratteri che contiene. Al contrario, una stringa di tipo *&str* è immutabile, quindi una volta creata non può essere modificata.

Immagina di avere un robot come esempio: se il robot è mutabile, puoi cambiare le sue caratteristiche (colore, direzione, ecc.) senza doverne costruire uno nuovo. Questo è simile a una variabile dichiarata come mutabile. Se invece il robot fosse immutabile, ogni modifica richiederebbe la creazione di una nuova istanza del robot, proprio come accade con i tipi immutabili, dove una modifica crea un nuovo oggetto piuttosto che alterare quello esistente.

I tipi di dati immutabili sono più sicuri in contesti concorrenti, poiché non possono essere modificati una volta creati, il che facilita la prevenzione di problemi comuni

come le race condition. I tipi mutabili offrono maggiore flessibilità, ma richiedono una gestione attenta, soprattutto in ambienti multithreading. Rust semplifica questa gestione attraverso strumenti come *Mutex* e *Arc* per la gestione sicura della mutabilità in contesti concorrenti.
In generale possiamo affermare che i tipi di dati sono suddivisi in sei macrocategorie principali: tipi primitivi scalari, composti, di riferimento, di proprietà, speciali e tipi generici o specializzati, approfondiamoli di seguito:

***1) Tipi primitivi scalari*** suddivisi a loro volta in:

**- Numeri interi:** sono un tipo di dato primitivo utilizzato per rappresentare numeri senza parte decimale. La caratteristica fondamentale dei numeri interi in Rust è che il linguaggio offre una varietà di tipi di interi per gestire in modo efficiente diverse dimensioni di dati, a seconda delle necessità del programma. A differenza di altri linguaggi di programmazione, come C# o Java, Rust non ha direttamente tipi come *long, short, double* o *decimal* nel contesto dei numeri interi. Invece, suddivide i numeri interi in base alla lunghezza (cioè il numero di bit) e al fatto che siano o meno con segno.
Abbiamo quindi interi di diverse dimensioni: 8, 16, 32, 64, 128 bit e una dimensione che dipende dall'architettura della macchina (*isize e usize*). Per ogni dimensione, esistono due varianti: con segno (che possono rappresentare sia numeri positivi che negativi) e senza segno (che rappresentano solo numeri positivi). Gli interi con segno sono prefissati con la lettera i, mentre quelli senza segno con la lettera u. Ad esempio, i32 rappresenta un intero con segno a 32 bit, mentre u32 rappresenta un intero senza segno a 32 bit.
Ecco un esempio di come dichiarare e utilizzare vari numeri interi in Rust:

```
fn main() {
    let numero_piccolo: i8 = -128; // Un intero con segno a 8 bit, può variare da -128 a 127
    let numero_grande: u128 = 340282366920938463463374607431768211455; // Un intero senza segno
a 128 bit
```

```
    println!("Numero piccolo: {}", numero_piccolo);
    println!("Numero grande: {}", numero_grande);
}
```

*numero_piccolo* è un intero con segno a 8 bit, quindi può assumere valori da -128 a 127. D'altra parte, *numero_grande* è un intero senza segno a 128 bit, capace di rappresentare un intervallo di numeri molto più ampio, ma solo positivi.
Una delle peculiarità di Rust è che non permette automaticamente di eseguire operazioni tra tipi di interi di dimensioni diverse senza un'esplicita conversione. Questo evita molti errori comuni in altri linguaggi di programmazione dove le conversioni implicite tra tipi di interi possono portare a risultati inaspettati. Per esempio:

```
fn main() {
    let a: i32 = 10;
    let b: i64 = 20;

    // Rust non permette operazioni dirette tra `i32` e `i64` senza conversione:
    // let c = a + b; // Questo codice non compila

    // È necessario convertire uno dei due:
    let c = a as i64 + b;

    println!("Il risultato è: {}", c);
}
```

I tipi *long* e *short* non esistono come dati a sé stanti, ma i loro equivalenti possono essere trovati negli interi a 64 bit (i64, u64) e 16 bit (i16, u16). Allo stesso modo, *double* e *decimal* non fanno parte della categoria degli interi, ma rientrano rispettivamente nei tipi in virgola mobile (come f64) e nei tipi definiti dall'utente.
Un'altra caratteristica importante è il modo rigoroso in cui è gestito il comportamento di overflow degli interi. In modalità di debug, se un calcolo supera il valore massimo rappresentabile da un intero (ad esempio, sommare 1 a i8 quando è già a 127), il programma andrà in panico e terminerà, segnalando

l'errore. In modalità di rilascio (release), Rust applica il *wrapping* dei valori, riportando il numero al limite opposto (ad esempio, da 127 a -128 per i8):

```
fn main() {
    let mut x: i8 = 127;
    x += 1; // In modalità debug, questo causerà un panico
    println!("Valore di x: {}", x); // In modalità release, questo stampa -128
}
```

Per riassumere, i numeri interi coprono una gamma di dimensioni e segni, ma non ci sono tipi come *long, short, double* o *decimal* direttamente associati agli interi. I concetti simili sono rappresentati dai vari tipi di interi a bit, con un forte controllo sul comportamento per evitare errori comuni legati all'overflow e alla conversione implicita.

- **Numeri in virgola mobile:** sono utilizzati per rappresentare numeri reali, cioè numeri che possono avere una parte frazionaria oltre a una parte intera. Rust supporta due tipi principali di numeri in virgola mobile: f32 e f64, dove il numero dopo la f indica la precisione in bit. f32 rappresenta un numero in virgola mobile a 32 bit, mentre f64 rappresenta un numero in virgola mobile a 64 bit. La differenza principale tra questi due tipi risiede nella precisione e nell'intervallo di valori che possono rappresentare, con f64 che offre una precisione maggiore rispetto a f32.
I numeri in virgola mobile seguono lo standard IEEE 754, che è lo standard più comune per la loro rappresentazione nei linguaggi di programmazione. Questo significa che possono rappresentare numeri positivi, negativi, zero, valori molto piccoli e molto grandi, e persino valori speciali come NaN (*Not a Number*) e *infinity*. Vediamo un esempio:

```
fn main() {
    let x: f32 = 3.14; // Un numero in virgola mobile a 32 bit
    let y: f64 = 2.718281828459045; // Un numero in virgola mobile a 64 bit

    println!("Il valore di x è: {}", x);
    println!("Il valore di y è: {}", y);
```

```
}
```

In questo esempio, *x* è un numero in virgola mobile a 32 bit, che rappresenta il valore di π approssimato a due cifre decimali. D'altra parte, *y* è un numero in virgola mobile a 64 bit, che rappresenta il valore di e (la base dei logaritmi naturali) con una precisione molto maggiore. In generale, si preferisce utilizzare f64 per i calcoli in virgola mobile, poiché fornisce una maggiore precisione, ma f32 può essere utile quando si ha bisogno di risparmiare memoria e la precisione non è cruciale.
Ovviamente possiamo anche eseguire operazioni aritmetiche standard, come addizione, sottrazione, moltiplicazione e divisione:

```
fn main() {
    let a: f64 = 1.0;
    let b: f64 = 2.5;

    let somma = a + b;
    let differenza = a - b;
    let prodotto = a * b;
    let quoziente = a / b;

    println!("Somma: {}", somma);
    println!("Differenza: {}", differenza);
    println!("Prodotto: {}", prodotto);
    println!("Quoziente: {}", quoziente);
}
```

Le variabili *somma, differenza, prodotto* e *quoziente* vengono calcolate utilizzando operazioni aritmetiche tra due numeri in virgola mobile (*a* e *b*). Come in molti linguaggi di programmazione, anche in Rust la divisione tra due numeri in virgola mobile restituisce un altro numero in virgola mobile.
Come anticipato precedentemente, un aspetto interessante è la gestione dei valori speciali, come *NaN* e *infinity*. Questi possono emergere durante operazioni matematiche che non hanno un risultato definito, come la divisione di zero per zero o la radice quadrata di un numero negativo:

```
fn main() {
    let zero = 0.0;
    let infinito = 1.0 / zero;
    let nan = zero / zero;

    println!("Infinito: {}", infinito); // Infinito positivo
    println!("NaN: {}", nan); // Not a Number
}
```

Nel codice sopra, la divisione di un numero positivo per zero produce un valore *infinity*, mentre la divisione di zero per zero produce *NaN*. Questi risultati sono conformi allo standard IEEE 754 e vengono gestiti in modo sicuro e prevedibile.
Infine, è importante notare che i numeri in virgola mobile non sono adatti per tutte le situazioni, in particolare per le operazioni finanziarie, a causa delle limitazioni di precisione intrinseche nella loro rappresentazione binaria. Per situazioni che richiedono una precisione numerica esatta, come il calcolo di denaro, è preferibile utilizzare tipi di dati diversi, come interi o librerie specializzate che gestiscono la precisione decimale, come ad esempio *Rug* (https://crates.io/crates/rug/1.1.1).

- **Booleani**: questo tipo di dato, in poche parole, può essere vero (*True*) o falso (*False*). Nonostante a prima vista sembrino banali, in realtà sono molto utili per gestire il flusso del nostro codice, e lo scopriremo nel corso dei capitoli.
Per ora osserviamo un esempio, dove si desidera eseguire del codice solo se una certa condizione è soddisfatta, utilizzando una variabile booleana per rappresentare tale condizione. Rust supporta anche le classiche operazioni logiche come AND (&&), OR (||), e NOT (!), che permettono di combinare o invertire valori booleani:

```
fn main() {
    let is_rust_fun = true;
    let is_rust_hard = false;

    if is_rust_fun && !is_rust_hard {
```

```
        println!("Rust è divertente e non troppo difficile!");
    }

    let num = 10;
    let is_greater_than_five = num > 5;

    if is_greater_than_five {
        println!("Il numero è maggiore di cinque.");
    } else {
        println!("Il numero è cinque o meno.");
    }
}
```

Dichiariamo due variabili booleane, *is_rust_fun* e *is_rust_hard*. La prima è impostata a *true*, mentre la seconda a *false.* Nel blocco *if*, utilizziamo gli operatori && per combinare queste due condizioni, e *!* per negare il valore di *is_rust_hard*. Se la condizione risulta vera, viene stampato un messaggio. Successivamente, creiamo una variabile *is_greater_than_five* che verifica se il numero *num* è maggiore di 5, utilizzando un'espressione di confronto. Questa restituisce un valore booleano che viene poi utilizzato per determinare quale messaggio stampare.
I booleani sono strettamente tipizzati, il che significa che Rust non converte automaticamente altri tipi di dati (come gli interi) in booleani. Se si desidera confrontare un valore o una variabile con un'altra, bisogna esplicitamente usare un'espressione di confronto o un operatore logico.
Rust enfatizza la chiarezza e la sicurezza, e questo si riflette anche nel modo in cui gestisce i booleani. Ogni espressione booleana è semplice e chiara, evitando ambiguità o comportamenti imprevisti che possono verificarsi in altri linguaggi che hanno una tipizzazione meno rigorosa.

- **Caratteri (char)**: è utilizzato per rappresentare un singolo carattere Unicode. A differenza di molti altri linguaggi di programmazione, in cui un *char* è tipicamente un singolo byte e può contenere solo caratteri ASCII, in Rust un *char* è più performante: è a 32 bit e può rappresentare qualsiasi carattere Unicode, inclusi i

simboli, i caratteri speciali, e le emoji. Questo significa che può contenere caratteri provenienti da quasi tutte le lingue scritte, così come simboli e altri glifi. È dichiarato utilizzando le virgolette singole:

```
fn main() {
    let c = 'R';
    let emoji = '😊';

    println!("Il carattere è: {}", c);
    println!("L'emoji è: {}", emoji);
}
```

In questo esempio, *c* è una variabile di tipo *char* che contiene il carattere *R*, mentre *emoji* è un altro *char* che contiene, appunto, un'emoji. Quando esegui il programma, vedrai che Rust gestisce entrambi i tipi di caratteri senza problemi.
Un altro aspetto importante del tipo *char* è che, essendo un tipo a 32 bit, è in grado di rappresentare l'intera gamma di caratteri Unicode, il che lo rende molto più flessibile rispetto a un *char* di 8 bit (un byte), come quello che trovi in linguaggi come C o C++. Questo approccio garantisce che i programmi Rust possano lavorare con testo multilingue e simboli complessi in modo nativo.
Ecco un altro esempio che mostra l'utilizzo dei *char* in un contesto di confronto e iterazione:

```
fn main() {
    let lettera = 'a';

    if lettera == 'a' {
        println!("La lettera è una a.");
    }

    // Iterare su una stringa e stampare ogni char
    for c in "Rust".chars() {
        println!("{}", c);
    }
}
```

In questo esempio, il primo blocco *if* verifica se la variabile lettera contiene il carattere *a*. Se la condizione è vera, viene stampato un messaggio. Successivamente, viene utilizzato un ciclo for per iterare su una stringa, "Rust", convertendola in una sequenza di *char* con il metodo *.chars().* Ogni carattere viene quindi stampato su una riga separata.
Un aspetto cruciale da comprendere è che, anche se un *char* può rappresentare caratteri complessi come emoji, ognuno di essi è comunque solo un singolo valore Unicode. Le stringhe, che sono sequenze di *char*, vengono gestite separatamente e possono contenerne uno o più.
Questo tipo di dato è essenziale per la manipolazione accurata e sicura di testi internazionali e complessi, garantendo che i programmi possano trattare qualsiasi carattere Unicode in modo efficiente e corretto.

***2) Tipi composti:***

- **Array**: è un tipo composto che rappresenta una collezione di elementi dello stesso tipo, organizzati in un ordine specifico e con una lunghezza fissa. Gli array sono molto utili quando si ha bisogno di memorizzare una sequenza di valori che non cambia in dimensione durante l'esecuzione del programma. Poiché la dimensione di un array è nota a tempo di compilazione, il compilatore di Rust può garantire che l'accesso ai suoi elementi sia sicuro e privo di errori.
La sintassi per dichiarare un array è semplice, si utilizza una coppia di parentesi quadre [] per racchiudere i valori iniziali. Ad esempio, per dichiarare un array di cinque numeri interi, si potrebbe scrivere:

```
fn main() {
    let numeri = [1, 2, 3, 4, 5];
    println!("Il primo numero è: {}", numeri[0]);
    println!("Il secondo numero è: {}", numeri[1]);
}
```

In questo caso, abbiamo creato un array chiamato *numeri,* che contiene cinque elementi interi. Questi sono indicizzati a partire da 0, quindi *numeri[0]* accede al primo elemento (1), e *numeri[1]* accede al secondo elemento (2). Gli array hanno dimensione fissa, quindi una volta che è stato creato, non è possibile aggiungere o rimuovere elementi.
Un'altra caratteristica importante è la possibilità di specificare il tipo degli elementi e la lunghezza direttamente durante la dichiarazione. Ad esempio, per dichiarare un array di tre elementi di tipo i32 (interi a 32 bit), si può scrivere:

```
fn main() {
    let array_di_tre: [i32; 3] = [10, 20, 30];
    println!("L'array contiene: {:?}", array_di_tre);
}
```

Qui, [i32; 3] indica che l'array contiene tre elementi di tipo i32. La sintassi [10, 20, 30] inizializza l'array con i valori specificati.
È anche possibile inizializzare un array con lo stesso valore per tutti gli elementi, utilizzando una sintassi differente:

```
fn main() {
    let ripetuti = [0; 5]; // Crea un array con 5 zeri
    println!("Array con valori ripetuti: {:?}", ripetuti);
}
```

In questo caso, l'array *ripetuti* contiene cinque elementi, tutti inizializzati al valore 0. La sintassi [*valore; lunghezza*] è un modo comodo per creare insiemi con elementi ripetuti.
Gli array supportano anche la sicurezza del tipo e la gestione dei limiti. Se si tenta di accedere a un elemento al di fuori dei limiti della raccolta, Rust genererà un errore a tempo di esecuzione anziché causare un comportamento indefinito:

```
fn main() {
    let numeri = [1, 2, 3];
    // Questo causerà un panic a tempo di esecuzione
```

```
    println!("Elemento fuori dai limiti: {}", numeri[5]);
}
```

Poiché l'indice 5 è fuori dai limiti per un array di tre elementi, il programma fallirà con un messaggio di errore, evitando potenziali vulnerabilità o bug.
Gli array sono particolarmente utili in situazioni in cui è necessario lavorare con una collezione di elementi con una lunghezza fissa e ben definita. Tuttavia, se hai bisogno di una collezione di elementi che può variare in lunghezza, è più appropriato usare un tipo di dati come *Vec,* che offre maggiore flessibilità.
Di seguito mostriamo alcuni dei metodi principali in uso con gli array:

| Metodo | Utilizzo | Esempio |
|---|---|---|
| `len()` | *Riporta la lunghezza dell'array* | `let lunghezza = array.len();` |
| `is_empty()` | *Verifica se l'array è vuoto* | `let vuoto = array.is_empty();` |
| `get()` | *Restituisce un'opzione con l'elemento all'indice specificato* | `if let Some(valore) = array.get(2) { /* utilizzo valore */ }` |
| `first()` | *Restituisce il primo elemento come Option* | `if let Some(primo) = array.first() { /* utilizzo primo */ }` |
| `last()` | *Restituisce l'ultimo elemento come Option* | `if let Some(ultimo) = array.last() { /* utilizzo ultimo */ }` |
| `iter()` | *Restituisce un iteratore sugli elementi dell'array* | `for elem in array.iter() { /* utilizzo elem */ }` |
| `contains()` | *Verifica se un elemento è presente nell'array* | `let presente = array.contains(&valore);` |
| `reverse()` | *Inverte l'ordine degli elementi nell'array* | `array.reverse();` |
| `copy_from_slice()` | *Copia gli elementi da uno slice nell'array* | `array.copy_from_slice(&[1, 2, 3]);` |
| `split_at()` | *Divide l'array in due parti* | `let (prima_parte, seconda_parte) = array.split_at(2);` |

- **Slice**: rappresentano una vista su una sequenza continua di elementi in una collezione, come un array o un vettore. *A differenza degli array, che hanno una lunghezza fissa, permettono di lavorare con una porzione di dati senza dover possedere l'intera collezione*. Le *slice* sono utili quando si vuole passare parti di

una collezione a una funzione senza dover clonare i dati.
Una *slice* è sempre associata a una collezione sottostante, dalla quale eredita una parte degli elementi. Non possiede i dati, ma fa riferimento a essi, e la loro lunghezza è dinamica, ovvero può essere diversa ogni volta che si crea una nuova *slice*. Sono immutabili per impostazione predefinita, ma è possibile crearle mutabili.
La sintassi per dichiararla è semplice. Si utilizzano i due punti : per specificare l'intervallo degli elementi da includere:

```
fn main() {
    let numeri = [1, 2, 3, 4, 5];
    let parte = &numeri[1..4];
    println!("La slice contiene: {:?}", parte);
}
```

Qui, *parte* è una *slice* che fa riferimento agli elementi dal secondo al quarto dell'array *numeri*. L'output sarà 2, 3, 4. La sintassi *&numeri[1..4]* crea una *slice* che inizia dall'indice 1 (incluso) e termina all'indice 4 (escluso). In Rust sono rappresentate da *&[T],* dove T è il tipo degli elementi contenuti nella collezione.
Una delle caratteristiche importanti delle *slice* è che garantiscono la sicurezza dei limiti. Se si tenta di crearne una con un intervallo che supera quelli della collezione sottostante, Rust genererà un errore a tempo di esecuzione:

```
fn main() {
    let numeri = [1, 2, 3];
    // Questo causerà un panic a tempo di esecuzione
    let fuori_limiti = &numeri[1..5];
    println!("Questa slice non sarà mai stampata: {:?}", fuori_limiti);
}
```

Poiché l'intervallo specificato (1..5) supera la lunghezza dell'array *numeri*, il programma fallirà con un messaggio di errore.
Oltre alle *slice* su array, Rust permette di lavorare anche su stringhe. Sappiamo che queste sono rappresentate dal tipo *String*, mentre le *slice* di stringhe sono

rappresentate da *&str*. Anche qui, offrono una vista immutabile su una porzione di stringa, permettendo di lavorare con le sue parti senza doverla copiare:

```
fn main() {
    let s = String::from("Hello, Rust!");
    let hello = &s[0..5];
    let rust = &s[7..11];
    println!("La prima slice: {}", hello);
    println!("La seconda slice: {}", rust);
}
```

In questo caso, *hello* è una *slice* che contiene la sottostringa "Hello", mentre *rust* contiene la sottostringa "Rust". Le *slice* di stringhe sono ampiamente utilizzate grazie alla loro efficienza e sicurezza, permettendo di manipolare stringhe senza rischi di buffer overflow o accessi non validi.
Infine, è possibile crearne di mutabili utilizzando *&mut* al posto di *&*. Questo permette di modificare gli elementi della collezione attraverso la *slice*:

```
fn main() {
    let mut numeri = [1, 2, 3, 4, 5];
    let parte_mutabile = &mut numeri[2..4];
    parte_mutabile[0] = 10;
    println!("Array modificato: {:?}", numeri);
}
```

In questo esempio, *parte_mutabile* è una *slice* mutabile che modifica il terzo elemento dell'array *numeri* da 3 a 10. L'output sarà 1, 2, 10, 4, 5.
Essenzialmente, questo tipo di raccolta è fondamentale per la sua capacità di offrire accesso efficiente e sicuro a porzioni di dati, in pratica uno strumento potente per la manipolazione delle collezioni e delle stringhe.

| Metodo | Utilizzo | Esempio |
|---|---|---|
| `len()` | *Riporta la lunghezza della slice* | `let lunghezza = slice.len();` |
| `is_empty()` | *Verifica se la slice è vuota* | `let vuota = slice.is_empty();` |

| | | |
|---|---|---|
| `get()` | *Restituisce un'opzione con l'elemento all'indice specificato* | `if let Some(valore) = slice.get(2) { /* utilizzo valore */ }` |
| `first()` | *Restituisce il primo elemento come Option* | `if let Some(primo) = slice.first() { /* utilizzo primo */ }` |
| `last()` | *Restituisce l'ultimo elemento come Option* | `if let Some(ultimo) = slice.last() { /* utilizzo ultimo */ }` |
| `iter()` | *Restituisce un iteratore sugli elementi della slice* | `for elem in slice.iter() { /* utilizzo elem */ }` |
| `contains()` | *Verifica se un elemento è presente nella slice* | `let presente = slice.contains(&valore);` |
| `split_at()` | *Divide la slice in due parti* | `let (prima_parte, seconda_parte) = slice.split_at(2);` |
| `as_ptr()` | *Restituisce un puntatore grezzo all'inizio della slice* | `let ptr = slice.as_ptr();` |
| `clone_from_slice()` | *Copia il contenuto di un'altra slice in quella attuale* | `slice.clone_from_slice(&[1, 2, 3]);` |

- **Tupla**: è un tipo composto che consente di raggruppare un numero fisso di elementi di tipi diversi o uguali in un'unica entità. Le tuple sono utili quando si desidera restituire più valori da una funzione o quando è necessario lavorare con un piccolo insieme di valori correlati ma di tipi diversi. A differenza degli array, che contengono elementi dello stesso tipo, le tuple possono combinare elementi di tipi differenti. Sono definite utilizzando una coppia di parentesi tonde () e separando gli elementi con una virgola. Ogni tupla ha un tipo che è una combinazione dei tipi dei suoi elementi. Ad esempio, una tupla con un intero e una stringa avrà il tipo (i32, *&str*). Vediamo un esempio:

```
fn main() {
    let persona = ("Alice", 30, 1.75);
    println!("Nome: {}", persona.0);
    println!("Età: {}", persona.1);
    println!("Altezza: {}", persona.2);
}
```

La tupla *persona* contiene tre elementi: un nome di tipo *&str*, un'età di tipo i32 e un'altezza di tipo f64. Gli elementi della tupla possono essere accessibili utilizzando

la notazione con punto seguita dall'indice dell'elemento, dove l'indice inizia da zero. Quindi, *persona.0* si riferisce al primo elemento, *persona.1* al secondo, e così via. L'output di questo codice sarà:

```
Nome: Alice
Età: 30
Altezza: 1.75
```

Le tuple possono essere anche restituite dalle funzioni. Questo è utile quando una funzione deve restituire più valori:

```
fn main() {
    let risultato = calcola_area_perimetro(5.0, 3.0);
    println!("Area: {}", risultato.0);
    println!("Perimetro: {}", risultato.1);
}

fn calcola_area_perimetro(lunghezza: f64, larghezza: f64) -> (f64, f64) {
    let area = lunghezza * larghezza;
    let perimetro = 2.0 * (lunghezza + larghezza);
    (area, perimetro)
}
```

La funzione *calcola_area_perimetro* come si intuisce, calcola l'area e il perimetro di un rettangolo e restituisce una tupla contenente entrambi i valori. La funzione restituisce una tupla (f64, f64) dove il primo elemento rappresenta l'area e il secondo il perimetro. Quando si chiama la funzione, il risultato può essere scomposto e i valori possono essere utilizzati separatamente.
È anche possibile decomporre una tupla nei suoi elementi durante l'assegnazione, utilizzando una sintassi speciale:

```
fn main() {
    let (nome, età, altezza) = ("Bob", 25, 1.82);
    println!("Nome: {}", nome);
    println!("Età: {}", età);
    println!("Altezza: {}", altezza);
}
```

In questo caso, la tupla *("Bob", 25, 1.82)* viene decomposta nei suoi componenti, e ciascuno di essi viene assegnato a una variabile separata: *nome, età* e *altezza*. Essenzialmente, uno dei più grandi vantaggi delle tuple è che sono utilizzabili in una vasta gamma di contesti, dalle funzioni che devono restituire più valori alla gestione di dati correlati in modo efficiente e leggibile.

| Metodo | Utilizzo | Esempio |
|---|---|---|
| `tuple.0, tuple.1...` | *Accede agli elementi di una tupla tramite l'indice* | `let tuple = (10, "ciao", 3.5); let primo = tuple.0; // primo è 10` |
| `let (a, b, c) = tuple` | *Decompone la tupla in variabili* | `let tuple = (10, "ciao", 3.5); let (x, y, z) = tuple; // x è 10, y è "ciao", z è 3.5` |
| `std::mem::size_of_val` | *Restituisce la dimensione della tupla in byte* | `let tuple = (10, "ciao", 3.5); println!("{}", std::mem::size_of_val(&tuple)); // Stampa la dimensione in byte` |
| `std::cmp::PartialEq` | *Confronta due tuple per uguaglianza* | `let t1 = (1, 2, 3); let t2 = (1, 2, 3); assert_eq!(t1, t2); // Passa se t1 e t2 sono uguali` |
| `Debug formatting` | *Formattazione per il debug della tupla* | `let tuple = (10, "ciao", 3.5); println!("{:?}", tuple); // Stampa: (10, "ciao", 3.5)` |
| `tuple.0, tuple.1...` | *Accede agli elementi di una tupla tramite l'indice* | `let tuple = (10, "ciao", 3.5); let primo = tuple.0; // primo è 10` |
| `let (a, b, c) = tuple` | *Decompone la tupla in variabili* | `let tuple = (10, "ciao", 3.5); let (x, y, z) = tuple; // x è 10, y è "ciao", z è 3.5` |
| `std::mem::size_of_val` | *Restituisce la dimensione della tupla in byte* | `let tuple = (10, "ciao", 3.5); println!("{}", std::mem::size_of_val(&tuple)); // Stampa la dimensione in byte` |
| `std::cmp::PartialEq` | *Confronta due tuple per uguaglianza* | `let t1 = (1, 2, 3); let t2 = (1, 2, 3); assert_eq!(t1, t2); // Passa se t1 e t2 sono uguali` |
| `Debug formatting` | *Formattazione per il debug della tupla* | `let tuple = (10, "ciao", 3.5); println!("{:?}", tuple); // Stampa: (10, "ciao", 3.5)` |
| `tuple.0, tuple.1...` | *Accede agli elementi di una tupla tramite l'indice* | `let tuple = (10, "ciao", 3.5); let primo = tuple.0; // primo è 10` |
| `let (a, b, c) = tuple` | *Decompone la tupla in variabili* | `let tuple = (10, "ciao", 3.5); let (x, y, z) = tuple; // x è 10, y è "ciao", z è 3.5` |
| `std::mem::size_of_val` | *Restituisce la dimensione della tupla in byte* | `let tuple = (10, "ciao", 3.5); println!("{}", std::mem::size_of_val(&tuple)); // Stampa la dimensione in byte` |

- **Vettori**: il tipo *Vec* (abbreviazione di "vector") è un tipo composto dinamico che rappresenta una sequenza di elementi dello stesso tipo. A differenza degli array, che hanno una lunghezza fissa, i *Vec* possono crescere o ridursi in base alle necessità durante l'esecuzione del programma. Questa flessibilità li rende uno degli strumenti più utilizzati in Rust per la gestione di collezioni di dati.
Per creare un *Vec*, si può utilizzare la macro *vec![],* che permette di inizializzare un vettore con una lista di elementi:

```
fn main() {
    let mut numeri = vec![1, 2, 3, 4, 5];
    println!("Il primo numero è: {}", numeri[0]);
    println!("Il numero di elementi è: {}", numeri.len());
}
```

Il vettore *numeri* contiene una sequenza di interi. L'accesso ai singoli elementi avviene tramite gli indici, esattamente come negli array. La funzione *len()* restituisce il numero di elementi contenuti nel vettore.
Una caratteristica fondamentale dei *Vec* *è la capacità di mutare*, ovvero di aggiungere, rimuovere o modificare gli elementi. Questo richiede di dichiarare il *Vec* come mutabile utilizzando *mut,* così da poter effettuare operazioni come *push, pop,* e modifiche dirette agli elementi:

```
fn main() {
    let mut numeri = vec![1, 2, 3];
    numeri.push(4);
    numeri.push(5);
    println!("Dopo l'aggiunta: {:?}", numeri);

    numeri.pop();
    println!("Dopo la rimozione: {:?}", numeri);

    numeri[0] = 10;
    println!("Dopo la modifica: {:?}", numeri);
}
```

Come anticipato, usiamo *push* per aggiungere elementi alla fine del vettore e *pop*

per rimuoverne l'ultimo. La modifica di un elemento avviene accedendo direttamente all'indice desiderato. La macro *{:?}* nel *println!* è utilizzata per stampare il contenuto dell'intero vettore.
I *Vec* offrono anche una serie di metodi per operare su intere collezioni in modo efficiente. Per esempio, è possibile iterare sui suoi elementi usando un ciclo *for*:

```
fn main() {
    let numeri = vec![10, 20, 30];
    for numero in &numeri {
        println!("Il numero è: {}", numero);
    }
}
```

L'iterazione avviene qui tramite il riferimento al *Vec* (*&numeri*), che consente di accedere agli elementi senza spostarne la proprietà. In Rust, è importante fare attenzione alla gestione della proprietà e dei riferimenti per evitare problemi di memoria.
Il tipo *Vec* è altamente ottimizzato per le operazioni dinamiche e fornisce sicurezza nella gestione della memoria grazie al sistema di proprietà. Può essere ridimensionato automaticamente in memoria quando vengono aggiunti nuovi elementi, e offre funzionalità avanzate come la concatenazione, la trasformazione e la suddivisione delle collezioni.
Inoltre, grazie alla sua flessibilità e alle prestazioni, il *Vec* è ampiamente utilizzato per gestire liste di dati che necessitano di essere manipolate dinamicamente. Questo tipo rappresenta un equilibrio tra potenza e semplicità, rendendolo uno degli strumenti fondamentali nella programmazione in questo linguaggio.

| Metodo | Utilizzo | Esempio |
|---|---|---|
| `push()` | *Aggiunge un elemento alla fine del vettore* | `let mut v = vec![1, 2, 3]; v.push(4); // v è ora [1, 2, 3, 4]` |
| `pop()` | *Rimuove e restituisce l'ultimo elemento* | `let mut v = vec![1, 2, 3]; let x = v.pop(); // v è ora [1, 2], x è Some(3)` |
| `len()` | *Restituisce la lunghezza del vettore* | `let v = vec![1, 2, 3]; let lunghezza = v.len(); // lunghezza è 3` |

| | | |
|---|---|---|
| `is_empty()` | *Verifica se il vettore è vuoto* | `let v: Vec<i32> = Vec::new(); let vuoto = v.is_empty(); // vuoto è true` |
| `get()` | *Restituisce un riferimento opzionale all'elemento indicato* | `let v = vec![1, 2, 3]; let val = v.get(1); // val è Some(2)` |
| `remove()` | *Rimuove l'elemento all'indice specificato e lo restituisce* | `let mut v = vec![1, 2, 3]; let x = v.remove(1); // v è ora [1, 3], x è 2` |
| `insert()` | *Inserisce un elemento all'indice specificato* | `let mut v = vec![1, 3]; v.insert(1, 2); // v è ora [1, 2, 3]` |
| `clear()` | *Rimuove tutti gli elementi dal vettore* | `let mut v = vec![1, 2, 3]; v.clear(); // v è ora vuoto` |
| `contains()` | *Verifica se il vettore contiene un determinato elemento* | `let v = vec![1, 2, 3]; let trovato = v.contains(&2); // trovato è true` |
| `sort()` | *Ordina gli elementi del vettore in-place* | `let mut v = vec![3, 1, 2]; v.sort(); // v è ora [1, 2, 3]` |
| `push()` | *Aggiunge un elemento alla fine del vettore* | `let mut v = vec![1, 2, 3]; v.push(4); // v è ora [1, 2, 3, 4]` |
| `pop()` | *Rimuove e restituisce l'ultimo elemento* | `let mut v = vec![1, 2, 3]; let x = v.pop(); // v è ora [1, 2], x è Some(3)` |
| `len()` | *Restituisce la lunghezza del vettore* | `let v = vec![1, 2, 3]; let lunghezza = v.len(); // lunghezza è 3` |

Oltre a quelli appena visti, i tipi composti *HashMap, BTreeMap* e *HashSet* sono strutture dati avanzate che permettono di organizzare e gestire collezioni di dati in modo efficiente. Ognuno ha le sue caratteristiche e viene utilizzato in contesti specifici, offrendo differenti vantaggi in termini di velocità e ordine.

- **HashMap**: è una collezione chiave-valore non ordinata in cui ogni chiave è associata a un valore. Le chiavi in un *HashMap* devono essere uniche, e l'accesso ai valori avviene tramite queste chiavi. Viene implementato utilizzando una tabella hash, che rende l'accesso, l'inserimento e la rimozione molto efficienti.
Per utilizzare *HashMap,* è necessario importarlo dal modulo *std::collections*:

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

fn main() {
    let mut capitale = HashMap::new();
    capitale.insert("Italia", "Roma");
    capitale.insert("Francia", "Parigi");
    capitale.insert("Germania", "Berlino");
```

```
    println!("La capitale dell'Italia è: {}", capitale["Italia"]);
}
```

In questo esempio, creiamo un *HashMap* chiamato *capitale* per memorizzare le capitali dei paesi. Usiamo *insert* per aggiungere coppie chiave-valore, e poi ci accediamo usando la chiave. Poiché *HashMap* non mantiene l'ordine degli elementi, l'ordine di iterazione potrebbe variare.

- **BTreeMap**: è una collezione chiave-valore ordinata che memorizza le chiavi in un ordine specifico, implementato tramite un albero binario bilanciato. Questo tipo di mappa è utile quando l'ordine degli elementi è importante o quando è necessario un accesso sequenziale alle chiavi. Anche *BTreeMap* viene importato dal modulo *std::collections*:

```
use std::collections::BTreeMap;

fn main() {
    let mut capitale = BTreeMap::new();
    capitale.insert("Italia", "Roma");
    capitale.insert("Francia", "Parigi");
    capitale.insert("Germania", "Berlino");

    for (paese, cap) in &capitale {
        println!("La capitale di {} è: {}", paese, cap);
    }
}
```

Utilizziamo *BTreeMap* allo stesso modo di *HashMap*, ma qui le chiavi sono automaticamente ordinate alfabeticamente. L'iterazione su di esse segue quest'ordine, il che rende *BTreeMap* utile quando si vuole mantenere le chiavi in ordine durante l'operazione di ricerca.

- **HashSet**: è una collezione non ordinata di elementi unici. È simile a *HashMap*, ma non memorizza valori associati alle chiavi; piuttosto, ogni elemento in un *HashSet* è una chiave unica. Questo lo rende utile per verificare l'unicità degli

elementi o per effettuare operazioni come unione, intersezione e differenza tra insiemi. Lo importiamo dal modulo *std::collections*:

```
use std::collections::HashSet;

fn main() {
    let mut numeri = HashSet::new();
    numeri.insert(1);
    numeri.insert(2);
    numeri.insert(3);

    if numeri.contains(&2) {
        println!("Il numero 2 è presente nel set.");
    }

    numeri.remove(&2);

    if !numeri.contains(&2) {
        println!("Il numero 2 è stato rimosso dal set.");
    }
}
```

In questo esempio, *numeri* è un *HashSet* che contiene una serie di numeri interi. Usiamo *insert* per aggiungere elementi e *contains* per verificare se un elemento è presente. La rimozione di un elemento avviene con *remove.* Poiché *HashSet* non mantiene un ordine specifico, l'iterazione sugli elementi non segue un ordine particolare.

In pratica *HashMap* è ideale quando si cerca la massima efficienza nelle operazioni di inserimento e ricerca, ma non si ha bisogno di un ordine specifico. *BTreeMap* è utile quando è necessario mantenere le chiavi ordinate, ad esempio per un accesso sequenziale o per operazioni che beneficiano dell'ordinamento. *HashSet,* invece, è la scelta giusta quando si vogliono gestire collezioni di elementi unici senza bisogno di ordinarli o associarli a valori.

Queste strutture sono essenziali per la gestione efficiente delle collezioni di dati, e la scelta tra di esse dipende dalle esigenze specifiche dell'applicazione. La loro corretta applicazione permette di scrivere codice più pulito, efficiente e sicuro.

***3) Tipi di riferimento***:

- **Reference**: o riferimenti, ne abbiamo già parlato, ora approfondiamo il concetto. Sono un modo per accedere ai dati senza prenderne il possesso. Esistono due tipi principali di reference: immutabili e mutabili. Le prime sono indicate con &T, mentre quelle mutabili con *&mut T*. Questi meccanismi sono fondamentali per garantire la sicurezza della memoria e prevenire condizioni di *data race*, ovvero situazioni in cui più parti del programma accedono contemporaneamente e in modo non sicuro alla stessa variabile.
Una reference immutabile *&T* permette di leggere i dati ma non di modificarli. Quando la passi a una funzione o la utilizzi altrove, stai dicendo al compilatore che non modificherai quei dati attraverso quella reference. Rust ti permette di averne quante ne desideri su uno stesso dato, ma non puoi avere reference immutabili se ne esiste una mutabile attiva sullo stesso dato. Questo principio evita conflitti e garantisce la coerenza dei dati.
Un esempio di reference immutabile:

```
fn main() {
    let numero = 42;
    let reference_numero = &numero;

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

La variabile *numero* viene referenziata in modo immutabile con *&numero*, e, attraverso *reference_numero*, viene utilizzata per stampare il valore di *numero*. Ovviamente non è possibile modificarne il valore.
Le reference mutabili, *&mut T,* ti consentono di modificare i dati a cui fanno riferimento. Tuttavia, per mantenere la sicurezza, Rust impone che tu possa avere solo una reference mutabile attiva per un determinato dato alla volta, e nessuna immutabile può esistere simultaneamente a una reference mutabile per lo stesso

dato.
Ecco un esempio di reference mutabile:

```
fn main() {
    let mut numero = 42;
    let reference_mutabile = &mut numero;

    *reference_mutabile += 1;

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

In questo esempio, *numero* è dichiarata come *mut*, rendendola modificabile. Creiamo poi una reference mutabile *reference_mutabile* utilizzando *&mut numero*. Attraverso di essa, modifichiamo il valore di *numero* incrementandolo di uno. La sintassi **reference_mutabile* permette di dereferenziarla e accedere direttamente al dato a cui punta, consentendone la modifica. Infine, stampiamo il valore aggiornato.
Il principio di ownership e borrowing, che si applica a *&T* e *&mut T*, è cruciale per la gestione sicura della memoria e per prevenire errori comuni in altri linguaggi, come i *null pointer* e *le race condition*. Le reference immutabili ti danno la sicurezza di sapere che i dati non verranno cambiati, mentre le reference mutabili ti permettono di gestire la mutabilità in modo controllato, garantendo che non si verifichino modifiche non sincronizzate ai dati. Approfondiremo questi concetti nei prossimi capitoli.
In sintesi, Rust utilizza queste due tipologie di reference per garantire che i tuoi programmi siano sicuri e privi di errori comuni legati alla gestione della memoria, mantenendo al contempo un alto grado di efficienza.

- **Slice (&[T] e &str)**: sono tipi di riferimento che consentono di accedere a una porzione di dati più ampia senza prenderne possesso. Le *slice* sono essenziali per lavorare con sezioni di array o stringhe in modo efficiente e sicuro, evitando la

copia dei dati e riducendo l'overhead di memoria. Vediamo più in dettaglio cosa sono e come funzionano.
Una *slice &[T]* è una reference a una parte di un array o di una collezione. T rappresenta il tipo degli elementi contenuti nell'array. Quando crei una *slice*, non stai copiando gli elementi, ma stai creando una vista su una parte dell'array originale, con la garanzia che la *slice* non modificherà la raccolta se essa è immutabile. Questo è utile quando vuoi passare solo una parte di un array a una funzione, mantenendo la sicurezza e l'efficienza, vediamo:

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

    println!("La slice è: {:?}", slice_numeri);
}
```

Abbiamo un array *numeri* contenente cinque interi. Creiamo una slice *slice_numeri* che fa riferimento agli elementi da indice 1 a 3 dell'array originale (la notazione 1..4 indica gli indici 1, 2 e 3). La *slice* viene poi stampata, mostrando i valori [2, 3, 4]. Poiché *slice_numeri* è una reference immutabile, non è possibile modificarne i valori al suo interno.
Le *slice* sono anche molto comuni nel contesto delle stringhe. Una stringa slice (*&str*) è una reference immutabile a una sequenza di caratteri. Anche in questo caso, la stringa non possiede i dati a cui fa riferimento, ma consente di lavorare con una parte di essa in modo sicuro ed efficiente. Sono usate frequentemente quando passi delle stringhe a una funzione e non ne hai bisogno di una copia completa.
Vediamone un esempio su una stringa:

```
fn main() {
    let frase = String::from("Benvenuto in Rust!");
    let parola = &frase[0..9];

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

```
}
```

*String* contiene il testo "Benvenuto in Rust!". Creiamo una slice *parola* che fa riferimento ai primi 9 caratteri della stringa (gli indici vanno da 0 a 8), questa contiene la parola "Benvenuto" e viene stampata. Poiché *parola* è una *slice* immutabile, non possiamo modificarne il contenuto.
Una peculiarità delle stringhe slice (*&str*) è che possono fare riferimento sia a una stringa *String* sia a una letterale come "ciao". Queste sono già delle *slice*, quindi non devi fare nulla di speciale per crearne una da una stringa letterale.
Le *slice* sono importanti perché ti permettono di manipolare i dati in modo efficiente, accedendo solo alle parti che ti servono e mantenendo il controllo sulla mutabilità. Questo approccio evita sprechi di memoria e riduce il rischio di errori comuni legati alla gestione delle stringhe e degli array in altri linguaggi.

- **Raw pointer**: rappresentati da **const T* e **mut T*, sono puntatori "grezzi" che offrono un accesso diretto alla memoria, senza le garanzie di sicurezza tipiche delle reference regolari (*&T* e *&mut T*). Questi puntatori consentono un controllo a basso livello, ma richiedono molta attenzione, poiché non beneficiano delle protezioni del sistema di ownership e borrowing. I *raw pointer* sono utili principalmente in contesti dove è necessario interagire con codice non sicuro o con librerie esterne, come quelle scritte in C.
Un *raw pointer* immutabile, **const T,* punta a un valore di tipo T che non può essere modificato tramite quel puntatore. È simile a una reference immutabile &T, ma senza alcuna verifica da parte del compilatore su mutabilità, validità, o lifetime. Questo significa che è possibile avere raw pointer che puntano a dati non validi o non allineati correttamente, e il loro uso non è sicuro a meno di prendere precauzioni aggiuntive.
Un *raw pointer* mutabile, **mut T,* invece, consente di modificare il valore a cui punta. Come nel caso di **const T*, anche **mut T* non ha alcuna garanzia di sicurezza, il compilatore non impedirà operazioni pericolose come la

dereferenziazione di puntatori nulli o il doppio uso di reference mutabili:

```
fn main() {
    let mut x: i32 = 42;

    // Creazione di raw pointer
    let ptr_immutabile: *const i32 = &x;
    let ptr_mutabile: *mut i32 = &mut x;

    // Dereferenziazione dei raw pointer (richiede un blocco unsafe)
    unsafe {
        println!("Il valore di x tramite ptr_immutabile: {}", *ptr_immutabile);
        *ptr_mutabile += 1;
        println!("Il valore di x tramite ptr_mutabile: {}", *ptr_mutabile);
    }
}
```

In questo esempio, iniziamo con una variabile *x* di tipo i32. Creiamo un raw pointer immutabile *ptr_immutabile* e uno mutabile *ptr_mutabile*, puntando rispettivamente alla versione immutabile e mutabile di *x*. Poiché i *raw pointer* non garantiscono la sicurezza della memoria, tutte le operazioni che coinvolgono dereferenziazione (ovvero accedere al valore puntato) devono essere eseguite all'interno di un blocco *unsafe*. Qui, dereferenziamo *ptr_immutabile* per leggere il valore di *x* e poi dereferenziamo *ptr_mutabile* per incrementare il valore di x. È importante sottolineare che l'uso di blocchi *unsafe* è necessario per indicare chiaramente che stiamo eseguendo operazioni che il compilatore non può verificare come sicure.

Ovviamente i *raw pointer* devono essere utilizzati con molta cautela. Sono strumenti utili quando si deve interagire con codice esterno o quando si lavora a basso livello con la memoria, dove si ha un controllo esplicito sulla sua gestione, con rischi sulla stabilità. Quindi, il loro uso improprio può facilmente introdurre bug difficili da rilevare, come dereferenziazioni errate, condizioni di gara, o corruzione della memoria. Per questi motivi, Rust incoraggia l'uso di raw pointer solo quando non esistono alternative più sicure.

## ***4) Tipi di proprietà:***

- **String**: è una struttura dati che rappresenta una stringa di caratteri UTF-8 dinamica e modificabile. A differenza delle stringhe letterali (*&str*), che sono slice immutabili e hanno una lunghezza fissa nota a tempo di compilazione, le *String* sono allocate dinamicamente sull'heap, il che consente loro di crescere e ridursi in base alle necessità del programma. Questo le rende particolarmente utili quando si ha bisogno di costruire o modificare stringhe in modo dinamico.
La creazione di una *String* può avvenire in diversi modi. Un modo comune è quello di utilizzare il metodo *String::from* per convertire una stringa letterale in una *String*. Una volta creata, non può essere modificata, ad esempio aggiungendo nuovi caratteri o concatenando altre stringhe. Ecco un esempio:

```
fn main() {
    // Creazione di una String a partire da una stringa letterale
    let mut saluto = String::from("Ciao");

    // Aggiunta di un'altra stringa
    saluto.push_str(", mondo!");

    // Concatenazione con un'altra stringa
    let esclamazione = String::from(" Benvenuti in Rust!");
    saluto.push_str(&esclamazione);

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

Iniziamo creando una *String* chiamata *saluto* usando *String::from* e la inizializziamo con il valore "Ciao". Poi, aggiungiamo un'altra stringa ", mondo!" usando il metodo *push_str*, che permette di concatenare una stringa *slice (&str)* alla *String* esistente. Infine, concateniamo la String *esclamazione* a *saluto*, passando una slice di *esclamazione* (ottenuta con *&esclamazione*) al metodo *push_str.*

Un'altra operazione comune è la modifica individuale di caratteri nella *String*. Tuttavia, questo richiede attenzione, poiché Rust si assicura che le operazioni sulle stringhe rispettino la validità dell'encoding UTF-8. Ad esempio, non puoi semplicemente modificare un singolo byte in una *String* senza rischiare di corrompere la validità UTF-8 della stringa. Perciò il linguaggio fornisce anche il metodo *push* per aggiungere singoli caratteri a una *String*:

```
fn main() {
    let mut parola = String::from("Ru");

    // Aggiunta di un carattere alla String
    parola.push('s');
    parola.push('t');

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

La String *parola* inizia con "Ru" e aggiungiamo i caratteri 's' e 't' uno alla volta utilizzando *push*. Il risultato finale sarà "Rust".
Le String sono anche usate per costruire stringhe da altre variabili e tipi usando il *macro format!,* che è simile a *printf* in C o a *string interpolation* in altri linguaggi:

```
fn main() {
    let nome = "Alice";
    let saluto = format!("Ciao, {}!", nome);
    println!("{}", saluto);
}
```

In questo esempio, la *macro format!* crea una nuova String formattata inserendo il valore della variabile *nome* al suo interno. Questo approccio è molto flessibile e permette di costruire stringhe complesse in modo chiaro e sicuro.
La gestione delle stringhe è progettata per essere sicura e efficiente, rispettando l'encoding UTF-8 e offrendo flessibilità grazie alla gestione della memoria tramite il sistema di ownership. Tuttavia, lavorare con String richiede una buona comprensione delle differenze tra *String* e *&str*, i quali rappresentano due modi

diversi di gestire le stringhe. Il primo è un tipo allocato dinamicamente che permette modifiche e cresce in dimensione durante l'esecuzione del programma. Viene usato quando è necessario possedere e modificare una stringa. Ad esempio:

```
let mut s = String::from("Ciao");
s.push_str(", mondo!");
println!("{}", s);
```

*&str* è invece un riferimento immutabile a una stringa, tipicamente utilizzato per quelle letterali o per passare stringhe senza trasferirne la proprietà. Non può essere modificato:

```
let s: &str = "Ciao, mondo!";
println!("{}", s);
```

La differenza principale è che *String* è mutabile e possiede i dati, mentre *&str* è un riferimento a una stringa già esistente e immutabile.
Stabilito ciò passiamo ora alle operazioni con le stringhe, potenti e flessibili, ma anche un po' diverse da altri linguaggi a causa del sistema di gestione della memoria e del modello di proprietà. Vediamo come eseguirle.
Per la ricerca di sottostringhe, abbiamo il metodo *contains,* che permette di verificare se una determinata sottostringa è presente in una stringa. Ad esempio:

```
let frase = "Questo è un esempio.";
let contiene = frase.contains("esempio");
println!("La frase contiene 'esempio'? {}", contiene);
```

Se hai bisogno di trovare l'indice di una sottostringa, puoi usare *find*:

```
let posizione = frase.find("esempio");
println!("La posizione della parola 'esempio' è: {:?}", posizione);
```

La sostituzione di caratteri o sottostringhe può essere eseguita utilizzando il metodo *replace,* che sostituisce tutte le occorrenze di una determinata sottostringa con un'altra:

```
let frase_modificata = frase.replace("esempio", "test");
println!("{}", frase_modificata);
```

Per quanto riguarda la conversione di maiuscole e minuscole, esistono i metodi *to_uppercase* e *to_lowercase*. Questi restituiscono una nuova stringa con tutti i caratteri convertiti:

```
let minuscolo = "ciao".to_uppercase();
println!("{}", minuscolo);

let maiuscolo = "CIAO".to_lowercase();
println!("{}", maiuscolo);
```

La rimozione di spazi vuoti all'inizio e alla fine di una stringa può essere fatta con *trim*, che restituisce una stringa senza spazi iniziali e finali:

```
let stringa_con_spazi = "  troppo spazio  ";
let senza_spazi = stringa_con_spazi.trim();
println!("'{}'", senza_spazi);
```

Oltre a queste operazioni, ci sono altre istruzioni, ad esempio per ottenere la lunghezza di una stringa con *len*, o accedere ai singoli caratteri tramite *slicing*. Per la concatenazione di stringhe, puoi usare l'operatore + oppure il metodo *push_str*:

```
let parte1 = "Ciao, ";
let parte2 = "mondo!";
let saluto = parte1.to_string() + parte2;
println!("{}", saluto);

let mut frase = String::from("Ciao");
frase.push_str(", mondo!");
println!("{}", frase);
```

Le stringhe possono essere suddivise in sottostringhe utilizzando *slicing* o il metodo *split,* che le divide in base a un delimitatore e restituisce un iteratore:

```
let testo = "uno,due,tre";
```

```
for parola in testo.split(',') {
    println!("{}", parola);
}
```

Inoltre è possibile anche formattarle utilizzando il macro *format!,* che permette di costruire stringhe con un testo formattato in maniera simile a *println!,* ma senza stamparle direttamente:

```
let nome = "Mario";
let saluto = format!("Ciao, {}!", nome);
println!("{}", saluto);
```

Infine, la ripetizione di una stringa può essere realizzata con il metodo *repeat,* che ripete la stringa un numero specifico di volte:

```
let ripetuta = "ciao".repeat(3);
println!("{}", ripetuta);
```

Queste operazioni coprono la maggior parte delle manipolazioni che si possono eseguire con le stringhe, fornendo una base solida per costruire applicazioni complesse che richiedono una gestione avanzata del testo.

| Metodo | Utilizzo | Esempio |
|---|---|---|
| `is_empty()` | *Verifica se una stringa è vuota* | `let vuota = a.is_empty();` |
| `len()` | *Riporta la lunghezza della stringa in byte* | `let lunghezza = a.len();` |
| `substring()` | *Restituisce una sottostringa dalla stringa* | `let b = &a[4..];` |
| `find()` | *Fornisce l'indice di una sottostringa, se presente* | `if let Some(pos) = a.find("Ciao") { /* utilizzo pos */ }` |
| `replace()` | *Sostituisce tutte le occorrenze di una sottostringa con un'altra* | `let nuova = a.replace("vecchia", "nuova");` |
| `to_uppercase()` | *Converte tutti i caratteri in maiuscolo* | `let maiuscola = a.to_uppercase();` |
| `to_lowercase()` | *Converte tutti i caratteri in minuscolo* | `let minuscola = a.to_lowercase();` |
| `trim()` | *Rimuove gli spazi vuoti all'inizio e alla fine della stringa* | `let pulita = a.trim();` |
| `contains()` | *Verifica se la stringa contiene una sottostringa* | `let presente = a.contains("Z");` |
| `cmp()` | *Confronta due stringhe* | `let confronto = a.cmp(&b);` |

| | | |
|---|---|---|
| `join()` | *Unisce una serie di stringhe in una* | `let unita = vec!["a", "b", "c"].join("-");` |
| `starts_with()` | *Verifica se la stringa inizia con una determinata sottostringa* | `let inizia_con = a.starts_with("A");` |
| `ends_with()` | *Verifica se la stringa termina con una determinata sottostringa* | `let termina_con = a.ends_with("ok");` |

- **Struct**: sono uno dei principali costrutti per creare tipi di dati complessi e personalizzati. Permettono di raggruppare insieme variabili, chiamate campi, che possono avere differenti tipi di dati. Questo rende le *struct* molto utili per rappresentare concetti più articolati rispetto ai tipi primitivi.
La dichiarazione avviene definendo un nome e specificando i campi che compongono la *struct*, ciascuno con il proprio tipo. Per esempio, possiamo definirne una per rappresentare un punto in uno spazio bidimensionale:

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

fn main() {
    // Creazione di un'istanza della struct Punto
    let punto1 = Punto { x: 3.0, y: 4.0 };

    // Accesso ai campi della struct
    println!("Il punto si trova a ({}, {})", punto1.x, punto1.y);
}
```

*Punto* ha due campi: *x* e *y*, entrambi di tipo f64, che rappresentano le coordinate del punto nello spazio. Quando creiamo un'istanza di *Punto*, dobbiamo specificare un valore per ciascun campo, utilizzando la sintassi *Punto { x: 3.0, y: 4.0 }.* Per accedere ai campi di una struct, usiamo il punto (.) seguito dal nome del campo.
Per inizializzarle è anche possibile servirsi di una sintassi più concisa quando le variabili hanno lo stesso nome dei campi. Ad esempio:

```
fn main() {
```

```
    let x = 3.0;
    let y = 4.0;
    let punto2 = Punto { x, y }; // Utilizza direttamente i nomi delle variabili
    println!("Il punto si trova a ({}, {})", punto2.x, punto2.y);
}
```

Le struct in Rust possono essere rese mutabili, il che permette di modificare i campi dopo la creazione dell'istanza. Per fare questo, dobbiamo dichiarare la variabile come *mut*:

```
fn main() {
    let mut punto3 = Punto { x: 5.0, y: 6.0 };

    // Modifica dei campi della struct
    punto3.x = 7.0;
    punto3.y = 8.0;

    println!("Il punto modificato si trova a ({}, {})", punto3.x, punto3.y);
}
```

Oltre alle struct semplici, sono supportate anche le *tuple struct* e le *unit-like struct*. Le prime sono simili alle tuple, ma con un nome di tipo, mentre le altre sono struct senza campi, spesso usate per implementare specifici comportamenti tramite i *trait*. Osserviamo un esempio di *tuple struct*:

```
struct Colore(i32, i32, i32);

fn main() {
    let rosso = Colore(255, 0, 0);
    println!("Il rosso ha i valori RGB: ({}, {}, {})", rosso.0, rosso.1, rosso.2);
}
```

Qui, *Colore* è una *tuple struct* che ha tre campi di tipo i32 per rappresentare i valori RGB. I campi vengono accessi tramite indice, come nelle tuple normali.
Le struct sono fondamentali per creare tipi di dati che modellano concetti del mondo reale o specifiche entità di un programma. La loro potenza risiede nella capacità di essere combinate con altre caratteristiche, come i trait, per creare tipi

ricchi e comportamenti complessi, mantenendo al contempo la sicurezza della memoria e l'efficienza del linguaggio.

- **Enum**: sono un costrutto che consente di definire un tipo che può assumere uno tra diversi valori possibili, ciascuno dei quali può avere una struttura differente. Gli *enum* sono molto più potenti rispetto a quelli di altri linguaggi, poiché ogni variante di un *enum* può contenere dati aggiuntivi di qualsiasi tipo. Questo li rende estremamente flessibili per rappresentare concetti che hanno diverse varianti con informazioni associate.
Per dichiarare un *enum*, utilizziamo la parola chiave seguita dal nome e dall'elenco delle varianti racchiuse tra parentesi graffe. Ad esempio, possiamo definirlo per rappresentare un'opzione di pagamento, come questa:

```
enum MetodoPagamento {
    CartaCredito { numero: String, scadenza: String },
    BonificoBancario { iban: String },
    Contanti,
}

fn main() {
    let pagamento1 = MetodoPagamento::CartaCredito {
        numero: String::from("1234-5678-9012-3456"),
        scadenza: String::from("12/24"),
    };

    let pagamento2 = MetodoPagamento::BonificoBancario {
        iban: String::from("IT60X0542811101000000123456"),
    };

    let pagamento3 = MetodoPagamento::Contanti;

    match pagamento1 {
        MetodoPagamento::CartaCredito { numero, scadenza } => {
            println!("Pagamento con carta di credito. Numero: {}, Scadenza: {}", numero,
scadenza);
        }
        MetodoPagamento::BonificoBancario { iban } => {
```

```
            println!("Pagamento con bonifico bancario. IBAN: {}", iban);
        }
        MetodoPagamento::Contanti => {
            println!("Pagamento in contanti.");
        }
    }
}
```

*MetodoPagamento* è un *enum* che rappresenta tre possibili modalità di pagamento: con carta di credito, con bonifico bancario, o in contanti. Le prime due varianti, *CartaCredito* e *BonificoBancario*, includono dati aggiuntivi, come il numero della carta o l'IBAN. La variante *Contanti*, invece, non ha bisogno di alcun dato aggiuntivo. Utilizziamo il costrutto *match* per eseguire un'operazione diversa a seconda della variante dell'*enum*.

Un altro esempio comune di uso è la gestione di opzioni e risultati. Gli *enum* *Option* e *Result* sono ampiamente utilizzati nella standard library di Rust. *Option* rappresenta un valore che può essere presente o assente, mentre *Result* rappresenta un'operazione che può avere successo o fallire, restituendo un valore o un errore rispettivamente. Adesso vedremo un esempio dove viene utilizzato *Option*:

```
fn trova_elemento(lista: Vec<i32>, target: i32) -> Option<usize> {
    for (indice, &valore) in lista.iter().enumerate() {
        if valore == target {
            return Some(indice);
        }
    }
    None
}

fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    match trova_elemento(numeri, 3) {
        Some(indice) => println!("Elemento trovato all'indice: {}", indice),
        None => println!("Elemento non trovato"),
    }
}
```

Qui, la funzione *trova_elemento* cerca un numero in un vettore e restituisce un *Option<usize>,* dove *Some(indice)* indica che l'elemento è stato trovato all'indice specificato, mentre *None* stabilisce che non è stato trovato. Nel *main*, il valore restituito viene analizzato con *match* per gestire entrambe le possibilità.
Gli *enum* sono particolarmente potenti perché combinano la capacità di definire varianti con dati associati e la rigorosa gestione dei casi tramite il pattern matching. Questo permette di scrivere codice che è sia espressivo sia sicuro, minimizzando gli errori di runtime attraverso la verifica del compilatore. L'uso degli *enum* è quindi fondamentale per modellare scenari in cui un valore può appartenere a diverse categorie, ciascuna con comportamenti e dati specifici.

***5) Tipi speciali:***

- **Option**: è un tipo speciale che rappresenta un valore che può essere presente o assente. È particolarmente utile per gestire situazioni in cui una funzione o un'espressione può o non può restituire un valore valido. *Option* è definito come un *enum* con due varianti: *Some(T),* che contiene un valore di tipo T, e *None,* che rappresenta l'assenza di valore. Questo tipo aiuta a prevenire errori comuni nei programmi, come dereferenziare un puntatore nullo.
Per capire come funziona, consideriamo una funzione che cerca un elemento in un vettore e restituisce il suo indice, se presente. Se non viene trovato, la funzione restituisce *None*:

```
fn trova_elemento(lista: Vec<i32>, target: i32) -> Option<usize> {
    for (indice, &valore) in lista.iter().enumerate() {
        if valore == target {
            return Some(indice);
        }
    }
    None
}
```

```
fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    match trova_elemento(numeri, 3) {
        Some(indice) => println!("Elemento trovato all'indice: {}", indice),
        None => println!("Elemento non trovato"),
    }
}
```

Perciò la funzione *trova_elemento* cerca un numero nel vettore *lista*. Se viene trovato, la funzione restituisce *Some(indice),* dove *indice* è la posizione del numero all'interno del vettore. Se il numero non è presente, la funzione restituisce *None.* Nel blocco *main*, utilizziamo il costrutto *match* per verificare se il risultato è *Some* o *None,* e agire di conseguenza.
Un altro esempio utile è nell'accesso a elementi di una collezione come un vettore. Se proviamo con uno che ha l'indice fuori dai limiti del vettore, Rust restituisce un *Option*, proteggendoti da errori di accesso non sicuro:

```
fn main() {
    let numeri = vec![10, 20, 30];
    let valore = numeri.get(1);

    match valore {
        Some(v) => println!("Il valore all'indice 1 è: {}", v),
        None => println!("Indice fuori dai limiti!"),
    }

    let valore_fuori_limite = numeri.get(10);

    match valore_fuori_limite {
        Some(v) => println!("Il valore all'indice 10 è: {}", v),
        None => println!("Indice fuori dai limiti!"),
    }
}
```

In questo esempio, *get* è un metodo che restituisce un *Option<&T>.* Se l'indice è valido, restituisce *Some*, contenente un riferimento al valore. Se l'indice è fuori dai limiti, restituisce *None*. Questo approccio rende il codice più sicuro, prevenendo

accessi a elementi non esistenti.
L'uso di *Option* è una pratica comune e raccomandata per evitare di lavorare con valori nulli o assenti. In Rust, invece di utilizzare puntatori nulli come in altri linguaggi, si utilizza *Option*, forzando così il programmatore a gestire esplicitamente il caso in cui un valore possa non essere presente.
Il tipo *Option* può essere combinato con altre funzionalità, come il pattern matching, la catena di metodi (*unwrap, map, and_then*) e la propagazione degli errori, per gestire elegantemente la presenza o assenza di valori in una varietà di contesti.

- **Result**: anche questa tipologia è fondamentale per la gestione degli errori in modo sicuro ed esplicito. A differenza di altri linguaggi che utilizzano eccezioni per segnalare errori, Rust adotta un approccio basato sui tipi per gestire le situazioni in cui un'operazione può fallire. Questo design favorisce la scrittura di codice più robusto e affidabile, poiché obbliga il programmatore a considerare e gestire esplicitamente i possibili errori.
Il tipo Result è un *enum* generico definito nella libreria standard e ha la seguente struttura:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Qui, *T* rappresenta il tipo del valore di successo, mentre *E* rappresenta quello dell'errore. Quando un'operazione va a buon fine, viene restituito *Ok* contenente il valore risultante, mentre in caso di errore viene restituito *Err* con un valore che descrive l'errore.
Consideriamo un esempio pratico di utilizzo di *Result.* Immaginiamo di voler leggere il contenuto di un file. La funzione *read_to_string* della libreria standard restituisce un *Result<String, std::io::Error>,* indicando che l'operazione può avere

successo restituendo il contenuto del file come stringa, oppure fallire restituendo un errore di tipo *std::io::Error*:

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

fn leggi_file(percorso: &str) -> Result<String, io::Error> {
    let mut file = File::open(percorso)?;
    let mut contenuto = String::new();
    file.read_to_string(&mut contenuto)?;
    Ok(contenuto)
}

fn main() {
    match leggi_file("esempio.txt") {
        Ok(contenuto) => println!("Contenuto del file:\n{}", contenuto),
        Err(e) => println!("Errore nella lettura del file: {}", e),
    }
}
```

Pertanto, la funzione *leggi_file* tenta di aprire un file per leggerne il contenuto e l'operatore ? viene utilizzato per propagare eventuali errori: se un'operazione fallisce, la funzione restituisce immediatamente l'errore senza continuare l'esecuzione. Nel *main,* utilizziamo il costrutto *match* per gestire i due possibili risultati: se l'operazione ha successo, stampiamo il contenuto del file; altrimenti, stampiamo un messaggio di errore.
Un altro aspetto importante è la possibilità di concatenare operazioni che possono fallire utilizzando metodi come *and_then*, *map* e *unwrap_or*. Questi permettono di trasformare e combinare i risultati in modo fluido, facilitando la gestione degli errori senza dover scrivere molteplici blocchi *match*.
Ad esempio, supponiamo di avere una funzione che converte una stringa in un numero intero e un'altra funzione che calcola la radice quadrata di un numero. Entrambe le operazioni possono fallire, quindi utilizziamo *Result* per gestire i possibili errori:

```
fn stringa_a_numero(s: &str) -> Result<i32, std::num::ParseIntError> {
    s.parse::<i32>()
}

fn radice_quadrata(n: i32) -> Result<f64, String> {
    if n >= 0 {
        Ok((n as f64).sqrt())
    } else {
        Err(String::from("Non è possibile calcolare la radice quadrata di un numero negativo"))
    }
}

fn main() {
    let input = "25";
    let risultato = stringa_a_numero(input).and_then(radice_quadrata);

    match risultato {
        Ok(risultato) => println!("La radice quadrata di {} è {}", input, risultato),
        Err(e) => println!("Errore: {}", e),
    }
}
```

Pertanto, utilizziamo *and_then* per concatenare le due operazioni: prima convertiamo la stringa in un numero e, se ha successo, calcoliamo la radice quadrata del numero ottenuto. Se una delle operazioni fallisce, l'errore viene propagato e gestito nel blocco *match*.
Per semplificare ulteriormente la gestione degli errori, Rust offre il metodo *unwrap_or*, che consente di fornire un valore di default nel caso in cui l'operazione fallisca. Ad esempio:

```
fn main() {
    let input = "abc";
    let numero = stringa_a_numero(input).unwrap_or(0);
    println!("Il numero è {}", numero);
}
```

Perciò, se la conversione della stringa a numero fallisce (ad esempio, se la stringa non rappresenta un numero valido), la funzione restituisce 0 come valore di

default.
Un'altra funzionalità utile di *Result* è l'uso della macro ?, che semplifica la propagazione degli errori. Questa può essere utilizzata all'interno di funzioni che restituiscono un *Result,* permettendo di scrivere codice più leggibile senza dover gestire esplicitamente ogni possibile errore.
Consideriamo un esempio in cui leggiamo e parsiamo (convertiamo un input testuale in una rappresentazione più utile) un file contenente un numero intero:

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

fn leggi_e_parsa(percorso: &str) -> Result<i32, Box<dyn std::error::Error>> {
    let mut file = File::open(percorso)?;
    let mut contenuto = String::new();
    file.read_to_string(&mut contenuto)?;
    let numero: i32 = contenuto.trim().parse()?;
    Ok(numero)
}

fn main() {
    match leggi_e_parsa("numero.txt") {
        Ok(n) => println!("Il numero è {}", n),
        Err(e) => println!("Si è verificato un errore: {}", e),
    }
}
```

Osserviamo come la funzione *leggi_e_parsa* apre un file, ne legge il contenuto e tenta di *parsarlo* come un numero intero. Ogni operazione che può fallire utilizza l'operatore ? per propagare l'errore, rendendo il codice più conciso e leggibile. Il tipo di errore restituito è *Box<dyn std::error::Error>,* che permette di gestire diversi tipi in modo uniforme.
*Result* è strettamente integrato con il sistema di tipi, permettendo di combinare diverse operazioni che possono fallire in modo sicuro ed efficiente. Questo approccio garantisce che tutti i possibili errori vengano considerati e gestiti, riducendo la probabilità di bug e comportamenti imprevisti nel programma.

**- Unit** e **Never**: giocano ruoli fondamentali nel linguaggio, soprattutto per quanto riguarda il controllo del flusso e la coerenza dei tipi.
Il tipo *Unit,* rappresentato da (), è un tipo speciale che indica l'assenza di un valore significativo. È simile a *void* in linguaggi come C o Java, ma con alcune differenze importanti. È utilizzato come tipo di ritorno per funzioni che non restituiscono alcun valore utile. Quando una funzione non deve restituire nulla di significativo, restituisce (). Questo tipo viene anche utilizzato come valore per espressioni che non producono un risultato concreto, come *println!,* che serve solo a produrre un effetto collaterale, come stampare qualcosa a schermo:

```
fn saluta() {
    println!("Ciao!");
}

fn main() {
    let risultato = saluta();
    println!("La funzione saluta ha restituito: {:?}", risultato);
}
```

Ed ecco che la funzione *saluta* non ha un tipo di ritorno esplicitamente dichiarato, quindi Rust assume che sia (). Quando chiamiamo *saluta* all'interno della funzione *main,* la variabile *risultato* assume il valore (), che rappresenta il fatto che la funzione ha completato la sua esecuzione senza restituire un valore significativo.
Un altro caso d'uso per il tipo *Unit* è nelle espressioni che non producono un valore, come le dichiarazioni di variabili o le assegnazioni:

```
fn main() {
    let x = 5;
    let y = {
        x + 1
    };
    println!("x: {}, y: {}", x, y);
}
```

In questo esempio, l'assegnazione *let x = 5;* ha il tipo (), perché non restituisce un valore utile. Tuttavia, l'espressione { x + 1 } restituisce un valore (6 in questo caso), che viene assegnato a y.
Invece, il tipo *Never*, rappresentato da !, è un altro tipo speciale che indica che una funzione o un'espressione non termina mai con successo. Questo tipo è utilizzato per funzioni che non restituiscono mai un valore, perché terminano con un errore, entrano in un ciclo infinito o causano l'arresto del programma.
Un esempio tipico è la macro *panic!,* che interrompe immediatamente l'esecuzione del programma:

```
fn errore_fatale() -> ! {
    panic!("Qualcosa è andato terribilmente storto!");
}

fn main() {
    errore_fatale();
    println!("Questa linea non sarà mai eseguita.");
}
```

Qui, la funzione *errore_fatale* restituisce !, il che significa che non terminerà mai normalmente. Di conseguenza, qualsiasi codice scritto dopo la chiamata a *errore_fatale* nella funzione *main* non verrà mai eseguito.
Un altro uso interessante del tipo *Never* è nelle situazioni in cui un'espressione deve produrre un valore di un tipo specifico, ma il programma può terminare in modo non previsto. Ad esempio, in un *match*:

```
fn descrivi_numero(x: i32) -> &'static str {
    match x {
        1 => "uno",
        2 => "due",
        3 => "tre",
        _ => {
            println!("Numero non supportato");
            std::process::exit(1); // Termina il programma
        }
    }
```

```
}
```

In questo caso, il ramo _ del *match* chiama *std::process::exit(1),* che restituisce !. Rust sa che ! può essere considerato come qualsiasi tipo, quindi il compilatore non solleverà errori anche se il ramo _ non restituisce un *&'static str* come gli altri rami del match.
In sintesi, il tipo *Unit* (()) rappresenta l'assenza di un valore utile, mentre il tipo *Never* (!) rappresenta un'assenza totale di valore, poiché indica che un'espressione non termina mai normalmente. Questi tipi sono strumenti potenti nell'ambiente Rust, che consentono di esprimere in modo preciso concetti legati al flusso di controllo e alla sicurezza del programma.

***6) Tipi generici o specializzati:***

**Closure**: Funzioni anonime che possono catturare valori dall'ambiente circostante.
**Trait object**: usato per il polimorfismo a tempo di esecuzione.
**Funzioni**: Considerate di prima classe, possono essere passate come argomenti e restituite da altre funzioni.

Queste ultime tipologie, essendo estremamente importanti, verranno ampliamente trattate nei prossimi capitoli, precisamente nel terzo, dedicato interamente alle funzioni, quindi anche alle chiusure, e nel quarto, dove approfondiremo i trait.

***Main e Match***
Come visto in alcuni esempi precedenti, il contesto *main* rappresenta il punto d'ingresso principale di un programma. La funzione *main* è sempre richiesta quando si sviluppa un'applicazione eseguibile. Può restituire un tipo () (cioè nessun valore) o un *Result* per gestire errori. Ad esempio, una funzione di base può essere scritta così:

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

Può anche accettare argomenti dalla riga di comando utilizzando la libreria *std::env.* Il contesto di *main* è quello in cui si avviano le operazioni principali del programma e da cui si chiamano altre funzioni o blocchi di codice. Può anche usare strutture di controllo come *if, match* o cicli per gestire il flusso del programma.
Il costrutto *match* è una potente struttura di controllo simile a *switch* in altri linguaggi, ma più versatile. Permette di confrontare un valore con diverse condizioni e di eseguire codice in base alla corrispondenza. Funziona bene con tipi di dati come *enum*, ma può essere utilizzato anche per controllare valori di tipi primitivi:

```
fn main() {
    let numero = 3;
    match numero {
        1 => println!("Uno"),
        2 => println!("Due"),
        3 => println!("Tre"),
        _ => println!("Altro numero"),
    }
}
```

Perciò, *match* confronta il valore di numero con diversi possibili valori. Se numero è 1, 2 o 3, viene stampato un messaggio corrispondente. L'underscore _ rappresenta un "catch-all" per ogni valore non specificato.
*Main* e *match* vengono usati spesso insieme, la loro correlazione non è stretta in senso formale, ma in pratica, il secondo è spesso utilizzato all'interno della funzione *main* per gestire diversi casi logici. Ad esempio, in un programma in cui si vogliono gestire argomenti diversi in base alla loro corrispondenza, *match* può essere usato per gestire questa logica:

```
fn main() {
    let comando = "inizia";
```

```
    match comando {
        "inizia" => println!("Programma avviato"),
        "ferma" => println!("Programma fermato"),
        _ => println!("Comando non riconosciuto"),
    }
}
```

Nel codice, *match* permette di decidere l'azione da intraprendere in base al comando ricevuto. Rispetto a un semplice costrutto *if, match* è molto più potente perché permette di gestire più situazioni in modo chiaro e conciso, specialmente quando si ha a che fare con *enum* o altre strutture complesse.
Il costrutto *match* è estremamente versatile e viene spesso utilizzato per gestire una varietà di scenari complessi. Vediamo alcuni casi d'uso avanzati che mostrano la sua potenza e flessibilità:

1) *match con enum*: uno degli utilizzi più comuni di *match* è con i tipi *enum*. Questi sono ideali quando si ha a che fare con una serie di stati o valori discreti, e *match* consente di gestirli in modo elegante:

```
enum Stato {
    InCorso,
    Completato,
    Fallito,
}

fn main() {
    let stato_corrente = Stato::InCorso;

    match stato_corrente {
        Stato::InCorso => println!("L'operazione è in corso."),
        Stato::Completato => println!("L'operazione è completata."),
        Stato::Fallito => println!("L'operazione è fallita."),
    }
}
```

In questo caso, *match* gestisce i diversi valori del tipo *Stato*, eseguendo un'azione in base allo stato corrente. Questo approccio è molto più robusto rispetto a una

catena di *if* perché rende il codice più leggibile e facile da estendere.

2) *match con pattern matching su tuple*: consente di controllare più valori contemporaneamente:

```
fn main() {
    let coordinata = (10, -5);

    match coordinata {
        (0, 0) => println!("Punto all'origine."),
        (x, 0) => println!("Punto sull'asse X: x = {}", x),
        (0, y) => println!("Punto sull'asse Y: y = {}", y),
        (x, y) => println!("Punto fuori asse: x = {}, y = {}", x, y),
    }
}
```

Qui infatti *match* controlla la tuple coordinata e gestisce diversi scenari, come il punto all'origine, i punti sugli assi o i punti fuori asse. Il pattern matching permette anche di catturare i valori e utilizzarli nel blocco di codice corrispondente.

3) *match con range di valori*: è possibile utilizzarlo anche per confrontare un valore contro un range di essi:

```
fn main() {
    let punteggio = 85;

    match punteggio {
        0..=50 => println!("Insufficiente"),
        51..=70 => println!("Sufficiente"),
        71..=90 => println!("Buono"),
        91..=100 => println!("Eccellente"),
        _ => println!("Punteggio non valido"),
    }
}
```

In questo terzo caso, *match* utilizza *range (0..=50, 51..=70, ecc.)* per classificare un punteggio all'interno di determinate fasce. Questo tipo di pattern matching è

molto utile quando si vuole categorizzare valori numerici.

4) *destrutturazione di Option con match*: come sappiamo *Option* è un tipo che rappresenta un valore che può essere presente (*Some*) o assente (*None*). *match* è il modo ideale per gestire questo tipo di dati:

```
fn main() {
    let opzione: Option<i32> = Some(42);

    match opzione {
        Some(valore) => println!("Il valore è {}", valore),
        None => println!("Nessun valore presente"),
    }
}
```

*match* permette di gestire facilmente la possibilità che il valore sia presente o meno. Questa tecnica è fondamentale per gestire operazioni che potrebbero fallire, come la ricerca di elementi in un array o una funzione che potrebbe non restituire un risultato.

5) *binding di pattern all'interno di match*: un altro aspetto interessante di *match* è la possibilità di effettuare binding direttamente all'interno del pattern, cioè assegnare variabili locali a valori che vengono catturati durante il confronto:

```
fn main() {
    let valore = Some(10);

    match valore {
        Some(x) if x > 5 => println!("Valore maggiore di 5: {}", x),
        Some(x) => println!("Valore minore o uguale a 5: {}", x),
        None => println!("Nessun valore"),
    }
}
```

Non solo si cattura il valore all'interno di *Some(x),* ma si aggiunge anche una condizione (*if x > 5*) che affina ulteriormente il controllo. Questo è un modo

potente per gestire scenari complessi in modo conciso.

6) *utilizzo di match per il controllo del flusso*: un altro caso d'uso interessante di match è quello di gestire errori o successi con il tipo speciale *Result*. Questo è estremamente comune in Rust, dove la gestione esplicita degli errori è una pratica comune:

```
fn divisione(dividendo: i32, divisore: i32) -> Result<i32, String> {
    if divisore == 0 {
        Err(String::from("Errore: divisione per zero"))
    } else {
        Ok(dividendo / divisore)
    }
}

fn main() {
    let risultato = divisione(10, 2);

    match risultato {
        Ok(valore) => println!("Risultato: {}", valore),
        Err(messaggio) => println!("Errore: {}", messaggio),
    }
}
```

Abbiamo visto come *match* gestisce il risultato della divisione, verificando se è un *Ok* (successo) o un *Err* (fallimento) e agendo di conseguenza. Ne parleremo nella gestione delle eccezioni. Questo è un modello molto comune per la gestione degli errori, quindi, oltre a semplici confronti, *match* consente di gestire scenari complessi come pattern matching su *enum*, tuple, range di valori e molto altro, rendendolo centrale per lo sviluppo di applicazioni con questo linguaggio.

### *I cicli e la preferenza per gli iteratori*

Nella programmazione, oltre ai tipi di dati, abbiamo a disposizione diverse strutture logiche che ci consentono di gestire il flusso del nostro codice. Queste strutture sono note come controllo del flusso e ci permettono di eseguire

determinate azioni in modo condizionato o ripetitivo.
Tra queste strutture, i cicli di programmazione ci introducono al concetto di iterazione. Essenzialmente è il processo di esecuzione ripetuta di un insieme di istruzioni per un numero di volte solitamente prestabilito, al fine di ottenere un risultato utile. In pratica, si tratta di un'azione di ripetizione che viene effettuata finché non si raggiunge una condizione cosiddetta di uscita.
In questo contesto, una sequenza ripetuta di istruzioni è definita come loop, ciclo o iterazione. È particolarmente utile quando dobbiamo eseguire la stessa operazione più volte con valori differenti, come ad esempio quando vogliamo analizzare tutti gli elementi di una lista o quando dobbiamo eseguire una serie di calcoli iterativi.
Le iterazioni, o cicli, sono un elemento fondamentale della programmazione in qualsiasi linguaggio. Consentono di eseguire un blocco di codice in modo ripetuto, "scorrendo" un insieme di dati o eseguendo un'operazione un numero specifico di volte.
Le iterazioni sono essenziali per diverse ragioni, innanzitutto automatizzano compiti ripetitivi. Invece di scrivere lo stesso codice più volte per ogni elemento di un insieme di dati, le iterazioni permettono di automatizzare il processo, rendendo il codice più conciso e leggibile.
Le iterazioni possono migliorare l'efficienza del codice eseguendo un'operazione su ogni elemento in modo ordinato, senza sprechi di risorse, inoltre offrono anche la flessibilità di eseguire più azioni su elementi vari all'interno di un insieme di dati.
Sono quindi un metodo fondamentale per eseguire un'operazione su un gruppo di elementi in modo sequenziale. Consentono di "scorrere" un insieme di dati, eseguendo un'azione su ciascuno di essi.
Rust offre tre principali costrutti di loop: *loop, while* e *for,* ciascuno con caratteristiche specifiche che lo rendono adatto a diversi scenari. La gestione dei loop è simile a quella di altri linguaggi, ma con alcune differenze significative, soprattutto in termini di sicurezza della memoria e gestione delle risorse.
L'istruzione *loop* è il costrutto di base per creare un ciclo infinito. Questo tipo di

iterazione continuerà ad eseguire il codice al suo interno fino a quando non verrà interrotto esplicitamente con un'istruzione *break*. Ad esempio:

```
fn main() {
    let mut contatore = 0;
    loop {
        contatore += 1;
        if contatore == 5 {
            println!("Il contatore ha raggiunto 5, esco dal loop.");
            break;
        }
    }
}
```

In questo codice, il *loop* incrementa il valore di contatore fino a raggiungere 5, momento in cui l'istruzione *break* interrompe il ciclo. Ciò è utile quando sai che vuoi eseguire il codice ripetutamente ma non conosci in anticipo il numero di iterazioni.

Il ciclo *while* è simile a quello presente in altri linguaggi di programmazione. Esso continua a eseguire il blocco di codice finché la condizione specificata risulta vera. Se la condizione è falsa al primo controllo, il codice all'interno del ciclo non verrà mai eseguito:

```
fn main() {
    let mut numero = 3;
    while numero != 0 {
        println!("{}!", numero);
        numero -= 1;
    }
    println!("Lancio!");
}
```

Quindi, il ciclo *while* continua a eseguire il codice al suo interno decrementando numero fino a quando numero non diventa 0. Solo allora si interrompe e il programma continua con le istruzioni successive.

Il ciclo *for,* estremamente potente, differisce in qualche modo da quello degli altri

linguaggi. In Rust, è spesso utilizzato per iterare su una collezione di elementi, come una lista o un range numerico:

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    for elemento in a.iter() {
        println!("Il valore è: {}", elemento);
    }
}
```

In questo esempio, *for* scorre attraverso ogni elemento dell'array a e lo stampa. Questo costrutto è sicuro e previene molti errori comuni come il superamento degli indici (*out-of-bounds*) che possono verificarsi in altri linguaggi.
Un aspetto distintivo dei cicli è la gestione della memoria e della sicurezza del tipo. Mentre in altri linguaggi l'uso improprio di cicli potrebbe portare a errori di gestione della memoria o accessi a dati non validi, in Rust il compilatore garantisce che tali situazioni vengano intercettate durante la fase di compilazione, prevenendo potenziali bug.
Visti i classici cicli, *dobbiamo dire che tuttavia Rust predilige l'uso degli iteratori*, che sono dei costrutti che producono una sequenza di valori. Questo approccio è più funzionale e spesso porta a codice più conciso ed espressivo. Eliminando alcuni costrutti più soggetti a errori, come i cicli con contatori che possono facilmente andare fuori dai limiti, Rust promuove la scrittura di codice più sicuro.
Gli iteratori offrono un modo potente, flessibile e sicuro per eseguire operazioni ripetitive come iterare su elementi di un array, di un vettore o di qualsiasi altra struttura iterabile. Un iteratore è un oggetto che implementa il trait *Iterator,* il quale fornisce il metodo *next* per accedere al prossimo elemento in una sequenza.
L'uso degli iteratori permette di scrivere codice più conciso e espressivo, eliminando molti dei potenziali errori legati all'uso di indici e alla gestione manuale del ciclo. Inoltre, gli iteratori possono essere composti e trasformati attraverso una serie di metodi come *map, filter, collect*, e altri, che consentono di eseguire operazioni complesse in modo efficiente.

Per esempio, considera una situazione in cui vuoi sommare tutti i numeri in un array. Invece di usare un ciclo *for* tradizionale, puoi usare un iteratore:

```
fn main() {
    let numeri = [1, 2, 3, 4, 5];
    let somma: i32 = numeri.iter().sum();
    println!("La somma è: {}", somma);
}
```

Nel codice, l'array *numeri* viene trasformato in un iteratore con *iter(),* e poi viene utilizzato il metodo *sum* per sommare tutti gli elementi. Il risultato è un codice estremamente pulito e privo di gestione manuale degli indici, riducendo così il rischio di errori.
Gli iteratori possono anche essere usati per trasformare i dati. Supponiamo di voler raddoppiare ogni numero in un array:

```
fn main() {
    let numeri = [1, 2, 3, 4, 5];
    let doppiati: Vec<i32> = numeri.iter().map(|x| x * 2).collect();
    println!("I numeri raddoppiati sono: {:?}", doppiati);
}
```

Qui, *iter()* crea un iteratore sugli elementi dell'array. Il metodo *map* viene usato per applicare una funzione a ogni elemento (in questo caso, raddoppiandolo), e *collect* raccoglie i risultati in un nuovo *Vec*. Il risultato è un nuovo vettore contenente i numeri raddoppiati.
Un altro vantaggio degli iteratori è che sono "lazy", cioè le operazioni su di essi non vengono eseguite fino a quando non è necessario. Questo significa che puoi concatenare molte operazioni senza preoccuparsi delle performance, perché Rust ottimizza l'esecuzione finale.
In conclusione, mentre i cicli *for, while,* e *loop* sono utili in molti contesti, gli iteratori offrono un modo più potente e sicuro per lavorare con le collezioni di dati. Sono particolarmente utili quando si tratta di trasformare e filtrare dati, rendendo il codice più leggibile e meno soggetto a errori. La preferenza per gli iteratori

riflette la filosofia di Rust di fornire strumenti che bilanciano potenza e sicurezza, massimizzando l'efficienza del codice.

### *Distinzione tra iterabile e iteratore*

La differenza tra un iterabile e un iteratore è fondamentale per comprendere come il linguaggio gestisce la manipolazione e il processamento di collezioni di dati in modo sicuro ed efficiente.

Un iterabile è qualsiasi tipo di collezione che può essere attraversata o iterata, come un *array*, un *Vec*, o una *slice*. Tuttavia, l'iterabile di per sé non fornisce il meccanismo per attraversare i suoi elementi. Per farlo, deve essere trasformato in un iteratore. Un iteratore è un oggetto che implementa il trait *Iterator* e fornisce una serie di metodi per iterare sui valori uno alla volta, come *next*.

Rust preferisce che si adoperino gli iteratori rispetto ai loop tradizionali come *for* e *while* per diversi motivi. Uno di questi è la sicurezza. Infatti i primi astraggono il processo di iterazione in modo tale che errori comuni, come il superamento dei limiti di un array (*buffer overrun*), siano impossibili. Inoltre, gli iteratori sono più concisi e flessibili, permettendo di comporre operazioni complesse in modo leggibile e funzionale.

Per esempio, consideriamo un array di numeri interi e vediamo come possiamo sommare i valori pari:

```
let numeri = [1, 2, 3, 4, 5, 6];

// Uso di un iteratore per sommare i numeri pari
let somma_pari: i32 = numeri.iter()
    .filter(|&&x| x % 2 == 0)  // Filtra solo i numeri pari
    .sum();  // Somma i numeri filtrati

println!("La somma dei numeri pari è: {}", somma_pari);
```

Ed ecco che, *numeri.iter()* crea un iteratore che attraversa l'array *numeri*. Il metodo *filter* utilizza una funzione lambda per selezionare solo i numeri pari, e

sum aggiunge insieme i valori rimanenti. Questo approccio è più sicuro e meno soggetto a errori rispetto a un loop manuale, in quanto gestisce automaticamente i limiti e non richiede contatori o indici espliciti.
Un loop tradizionale *for* in Rust è in realtà una sintassi più semplice per servirsi di un iteratore. Ad esempio:

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

È equivalente a:

```
let mut iteratore = numeri.iter();
while let Some(numero) = iteratore.next() {
    println!("Numero: {}", numero);
}
```

In quest'ultimo esempio, *iteratore.next()* restituisce *Some(valore)* finché ci sono elementi, e *None* una volta terminati. Questo approccio funzionale e sicuro si allinea con la filosofia di Rust di fornire un controllo rigoroso e sicuro della memoria e del flusso di esecuzione, riducendo al minimo gli errori e migliorando la leggibilità del codice.
In sintesi, mentre sia gli iterabili che i loop sono strumenti utili, gli iteratori offrono una soluzione più potente, sicura ed espressiva per manipolare collezioni di dati.

### *Approfondiamo gli operatori*

Abbiamo già visto gli operatori utilizzati per il confronto tra due valori, adesso andremo a elencarne l'intero pacchetto, leggermente meno fornito rispetto ad altri linguaggi:

- *Operatori di confronto*: comparano due valori.
- *Operatori di assegnazione*: attribuiscono un valore alle variabili.
- *Operatori aritmetici*: servono per eseguire le classiche operazioni matematiche.
- *Operatori logici*: combinano istruzioni condizionali.

- *Operatori bit a bit*: servono a confrontare solamente i numeri binari.
- *Operatori di accesso ai membri* . (punto) per accedere ai campi di una struct o ai metodi di un oggetto.
- *Operatore ternario*: non esiste, ma l'istruzione *if* può essere usata in un contesto con le espressioni.

Di seguito una tabella con la maggior parte degli operatori appena descritti:

| **Operatori *aritmetici*** | **Utilizzo** | **Esempio** |
|---|---|---|
| `+, -, *, /` | *Operazioni aritmetiche di base* | `(a + 1) * (b - 2) / c` |
| `%` | *Operatore modulo* | `5 % 2 // restituisce 1` |
| **Operatori di *assegnazione*** | | |
| `=` | *Assegnazione di un valore* | `a = 2` |
| `+=, -=` | *Assegnazione combinata* | `a += 2, a -= 2` |
| `*=, /=` | *Assegnazione combinata* | `a *= 2, a /= 2` |
| **Operatori *logici*** | | |
| `&&` | *"and" logico* | `a == 1 && b < 5` |
| `\|\|` | *"or" logico* | `` `a == 1 `` |
| `!` | *"not" logico* | `! (a == 1 && b < 5)` |
| **Operatore *bit a bit*** | | |
| `&` | *AND bit a bit* | `x & y` |
| `\|` | *OR bit a bit* | `x \| y` |
| `^` | *XOR bit a bit* | `x ^ y` |
| `>>` | *Shift bit a destra* | `x >> 1` |
| `<<` | *Shift bit a sinistra* | `x << 2` |
| `&` | *AND bit a bit* | `x & y` |

Rust non supporta operatori come *is, is not,* e non ne offre uno ternario come *?:*. Tuttavia, è possibile usare *if* come espressione per ottenere un risultato simile. Vediamo adesso un esempio con vari operatori:

```
fn main() {
    // Operatori aritmetici
    let a = 10;
    let b = 3;

    let somma = a + b;  // Somma
```

```
let sottrazione = a - b;  // Sottrazione
let prodotto = a * b;  // Moltiplicazione
let divisione = a / b;  // Divisione intera
let modulo = a % b;  // Resto della divisione (modulo)

println!("Somma: {}", somma);  // Risultato: 13
println!("Sottrazione: {}", sottrazione);  // Risultato: 7
println!("Prodotto: {}", prodotto);  // Risultato: 30
println!("Divisione: {}", divisione);  // Risultato: 3
println!("Modulo: {}", modulo);  // Risultato: 1

// Operatori di assegnazione
let mut c = 5;
c += 3;  // c = c + 3
println!("Dopo c += 3, c è: {}", c);  // Risultato: 8

c *= 2;  // c = c * 2
println!("Dopo c *= 2, c è: {}", c);  // Risultato: 16

// Operatori logici
let d = true;
let e = false;

if d && !e {  // d è vero, e è falso
    println!("d è vero e e è falso.");  // Questo viene stampato
}

if d || e {  // Uno dei due è vero (d è vero)
    println!("Almeno uno è vero.");  // Questo viene stampato
}

// Operatore di confronto
let f = 10;
let g = 5;

if f > g {
    println!("f è maggiore di g.");  // Risultato: f è maggiore di g
}

if f == 10 {
    println!("f è uguale a 10.");  // Risultato: f è uguale a 10
}
```

```
    // Operatore di condizione con if espressionale (simile a ternario)
    let h = if f > g { "f è maggiore" } else { "g è maggiore o uguale" };
    println!("Risultato della condizione: {}", h);  // Risultato: f è maggiore
}
```

Vengono utilizzati operatori aritmetici (+, -, *, /, %) per eseguire operazioni matematiche di base tra due variabili intere. Poi si passa a quelli di assegnazione come += e *= per modificare una variabile esistente senza ripetere il nome della variabile. Gli operatori logici &&, || e ! servono per valutare condizioni booleane. Proseguiamo usando l'operatore di confronto per confrontare due valori (>, ==). Infine, viene mostrato l'uso dell'espressione *if* come condizione, simile all'operatore ternario in altri linguaggi.
Vediamo anche un esempio più chiaro riguardante gli operatori che ci permettono di eseguire diverse operazioni sui bit:

```
fn main() {
    // Dichiarazione delle variabili
    let a: u8 = 0b1100;  // 12 in binario
    let b: u8 = 0b1010;  // 10 in binario

    // Operatore AND bit a bit (&)
    let and_result = a & b;  // 1100 & 1010 = 1000 (8 decimale)
    println!("a & b = {:08b} (decimale: {})", and_result, and_result);  // Risultato: 00001000
(8)

    // Operatore OR bit a bit (|)
    let or_result = a | b;  // 1100 | 1010 = 1110 (14 decimale)
    println!("a | b = {:08b} (decimale: {})", or_result, or_result);  // Risultato: 00001110
(14)

    // Operatore XOR bit a bit (^)
    let xor_result = a ^ b;  // 1100 ^ 1010 = 0110 (6 decimale)
    println!("a ^ b = {:08b} (decimale: {})", xor_result, xor_result);  // Risultato: 00000110
(6)

    // Operatore NOT bit a bit (~)
    let not_result = !a;  // ~1100 = 0011 (3 decimale, considerando 8 bit)
    println!("~a = {:08b} (decimale: {})", not_result, not_result);  // Risultato: 11110011 (243
decimale in rappresentazione u8)
```

```
    // Shift a sinistra (<<)
    let shift_left_result = a << 2;  // 1100 << 2 = 110000 (48 decimale)
    println!("a << 2 = {:08b} (decimale: {})", shift_left_result, shift_left_result);   //
Risultato: 00110000 (48)

    // Shift a destra (>>)
    let shift_right_result = a >> 2;  // 1100 >> 2 = 0011 (3 decimale)
    println!("a >> 2 = {:08b} (decimale: {})", shift_right_result, shift_right_result);   //
Risultato: 00000011 (3)
}
```

Usiamo AND (&) che confronta i bit di *a* e *b*. Se entrambi sono 1, il risultato sarà 1, altrimenti 0. OR (|) confronta i bit di *a* e *b*. Se almeno uno dei due è 1, il risultato sarà 1. XOR (^) confronta i bit di *a* e *b.* Il risultato sarà 1 solo se uno dei due è 1 ma non entrambi. NOT (!) inverte tutti i bit di a. In un intero a 8 bit, 0b1100 diventa 0b11110011. Lo shift a sinistra (<<) sposta i bit di a verso sinistra di 2 posizioni, aggiungendo 0 nei bit di minor valore. Infine, lo shift a destra (>>) sposta i bit di a verso destra di 2 posizioni, eliminando i bit di minor valore.
In questo esempio, il formato binario viene stampato usando {:08b}, che formatta l'output in una stringa binaria di 8 bit.

### *I primi passi*

Dopo aver appreso alcune basi proviamo a vedere del codice semplice, anche per applicare in parte ciò che abbiamo visto finora. Partiamo con delle operazioni con le stringhe e vediamo come applicare i concetti base che abbiamo visto finora:

```
fn main() {
    // Operazioni con le stringhe
    let sport = "calcio";
    println!("Quest'inverno mi impegnerò nel: {}", sport);

    // Operazioni con gli insiemi
    let sport_set = ["ciclismo", "calcio", "fitness"];
    println!("Quest'anno mi allenerò con il: {}", sport_set.join(", "));
```

```
    // Operazioni aritmetiche
    let x = 8.0 + 2.0 - 2.0 * 10.0 / 2.0;
    println!("{}", x); // Risultato: 0.0

    // Somma e differenza
    let x = (8.0 + 2.0 - 2.0) * 10.0 / 2.0;
    println!("{}", x); // Risultato: 40.0

    // Elevamento a potenza
    println!("{}", 10.0_f64.powi(2)); // Dieci elevato alla seconda: 100.0

    // Altre operazioni aritmetiche
    let mut y = 7;
    y = y + y * 2;
    println!("{}", y); // Risultato: 21

    // Shortcut per l'addizione
    let mut z = 7;
    z += 2;
    println!("{}", z); // Risultato: 9

    // Vedremo successivamente come convertire i dati
}
```

Il codice esegue una serie di operazioni basilari, in cui gestisce le stringhe in modo esplicito e richiede l'uso di metodi specifici per concatenare o unire elementi di un array, come *join*. Le operazioni aritmetiche sono simili, ma l'elevamento a potenza utilizza il metodo *powi* per i numeri interi.
Adesso vedremo come gestire la conversione dei dati, sia implicita che esplicita, detta *casting*.

### *Cast e numeri*

Rust è un linguaggio fortemente tipizzato, quindi ogni variabile deve essere dichiarata con un tipo specifico, il quale non può cambiare durante l'esecuzione del programma. Questo sistema di tipizzazione rigorosa permette al compilatore di rilevare errori di tipo prima della creazione finale, garantendo un livello di sicurezza e affidabilità maggiore. In altre parole, non è possibile assegnare un

valore di un tipo a una variabile di un altro tipo senza esplicita conversione, evitando così una serie di potenziali bug e comportamenti imprevedibili.
Ci è capitato in qualche esempio precedente, infatti, quando ci si confronta con la scrittura di codice più complesso, spesso ci troviamo nella necessità di definire specificamente il tipo di alcune variabili. Ad esempio, potremmo dover convertire un numero in virgola mobile in un intero, o viceversa, per varie ragioni, come il calcolo o la semplice necessità. Questo processo è noto come "cast" (*typecasting*) o "conversione di tipo".
In molti linguaggi solitamente esistono due tipi di conversione, esplicita e implicita. Rust richiede che qualsiasi conversione di tipo venga effettuata in modo esplicito utilizzando funzioni standard o metodi appropriati. Ad esempio, per convertire un numero in virgola mobile in un intero, si utilizza il metodo *as*, ma bisogna essere consapevoli del fatto che potrebbe comportare una perdita di precisione:

```
let decimale: f64 = 3.14;
let intero: i32 = decimale as i32; // Convertirà 3.14 in 3
```

Qui, il cast viene eseguito con l'operatore *as*, che è esplicito e non comporta eccezioni di runtime come in C#. In Rust, la conversione di un tipo complesso, come da String a un numero, richiede l'uso di metodi specifici come *parse*:

```
let testo = "123";

let numero: i32 = testo.parse().expect("Non è un numero valido");
```

In questo caso, *parse* tenta di convertire la stringa in un numero, e se non riesce, restituisce un Result, gestibile per evitare crash del programma.
In Rust, non esiste un operatore per convertire oggetti in altri tipi tramite casting. Tuttavia, gestisce l'ereditarietà dei tipi tramite *traits*, e se un tipo ne implementa uno, può essere convertito o trattato come quel trait in modo sicuro e controllato.
A differenza di altri linguaggi, non si effettuano mai cast impliciti. Ad esempio, se si tenta di confrontare un intero con un numero in virgola mobile, si richiede una

conversione esplicita:

```
let numero1: i32 = 5;
let numero2: f64 = 5.0;

if numero1 as f64 > numero2 {
    println!("numero1 è maggiore di numero2");
}
```

Questo garantisce che ogni operazione avvenga in un contesto chiaro e previsto, riducendo al minimo il rischio di bug o comportamenti imprevisti.
Quindi, Rust richiede sempre casting espliciti e non permette conversioni implicite. Questo contribuisce a una maggiore sicurezza e affidabilità nel codice, in linea con la filosofia di prevenire errori comuni a livello di compilazione.
Il casting tra tipi numerici può essere essenziale per eseguire operazioni matematiche, poiché Rust impone regole rigorose sui tipi di dati. Ad esempio, se si tenta di sommare un i32 a un f64, il compilatore genererà un errore. Per risolvere questo problema, come già detto, è necessario eseguire il cast esplicito del tipo, utilizzando metodi come *as*. Questo diventa particolarmente utile quando si lavora con funzioni matematiche avanzate. Le operazioni matematiche, come il calcolo di potenze, radici quadrate o logaritmi, richiedono tipi specifici come f64. Ad esempio, per eseguire una radice quadrata su un intero, bisogna prima effettuare il cast del valore a f64 con *as f64*, altrimenti il metodo *sqrt()* non sarà disponibile. Allo stesso modo, operazioni come *sin()* o *cos()* richiedono angoli in radianti e lavorano solo con tipi a virgola mobile. Quindi, un buon uso del cast diventa cruciale per manipolare correttamente numeri interi e decimali, permettendo l'accesso a tutte le funzioni della libreria standard di Rust. L'interazione tra casting e operazioni matematiche rende il linguaggio versatile e performante, ma richiede attenzione nella gestione dei tipi.
Osserviamo alcuni metodi relativi alle operazioni aritmetiche:

| Metodi | Utilizzo | Esempio |
|---|---|---|
| abs | *Calcola il valore assoluto* | `-5.0_f64.abs()` |
| ceil | *Arrotonda per eccesso all'intero più vicino* | `4.2_f64.ceil()` |
| floor | *Arrotonda per difetto all'intero più vicino* | `4.8_f64.floor()` |
| round | *Arrotonda all'intero più vicino* | `4.5_f64.round()` |
| powf | *Calcola un numero elevato alla potenza* | `2.0_f64.powf(3.0)` |
| sqrt | *Calcola la radice quadrata di un numero* | `16.0_f64.sqrt()` |
| max | *Restituisce il maggiore di due numeri* | `f64::max(3.0, 7.0)` |
| min | *Restituisce il minore di due numeri* | `f64::min(3.0, 7.0)` |
| sin | *Calcola il seno di un angolo (in radianti)* | `(std::f64::consts::PI / 2.0).sin()` |
| cos | *Calcola il coseno di un angolo (in radianti)* | `0.0_f64.cos()` |
| tan | *Calcola la tangente di un angolo (in radianti)* | `(std::f64::consts::PI / 4.0).tan()` |
| ln | *Calcola il logaritmo naturale di un numero* | `2.71828_f64.ln()` |
| log10 | *Calcola il logaritmo in base 10 di un numero* | `1000.0_f64.log10()` |
| exp | *Calcola l'esponenziale di un numero* | `1.0_f64.exp()` |
| trunc | *Tronca la parte frazionaria di un numero* | `4.8_f64.trunc()` |

La gestione dei cast e delle conversioni tra tipi è strettamente legata alla precisione e alla sicurezza che il linguaggio garantisce. La necessità di cast espliciti, come da f64 a i32, evita errori silenziosi e garantisce che l'operazione sia consapevole della possibile perdita di dati o di precisione. Questo diventa particolarmente rilevante quando si eseguono operazioni matematiche complesse, che richiedono di lavorare con numeri di diverse tipologie, come interi e numeri in virgola mobile.

Nell'implementare operazioni matematiche avanzate, la libreria *std::f64* offre molte delle funzionalità già familiari da altri linguaggi, come l'elevamento a potenza o il calcolo della radice quadrata, consentendo un controllo preciso sui dati numerici. Come, ad esempio, il metodo *f64::sqrt()* che permette di calcolare la radice quadrata. Tuttavia, come visto nella tabella delle funzioni matematiche, è sempre necessario prestare attenzione alla gestione dei tipi, poiché molte

operazioni possono richiedere conversioni esplicite per mantenere la coerenza dei risultati.

La generazione di numeri casuali si integra naturalmente in questo contesto, soprattutto quando si ha la necessità di manipolare numeri casuali all'interno di un certo range o con determinate proprietà matematiche. In Rust, il crate *rand* fornisce strumenti per generare numeri casuali di diverso tipo, come interi, float o booleani, e può essere utilizzata in combinazione con le operazioni matematiche disponibili. Ad esempio, se si genera un numero casuale in virgola mobile con *rand::thread_rng().gen::<f64>(),* e si vuole applicare una trasformazione matematica, come l'elevamento a potenza o il calcolo del logaritmo, sarà possibile farlo utilizzando i metodi matematici di f64 con le dovute conversioni quando necessario.

Perciò, come detto, la generazione di numeri casuali è gestita dal crate *rand*, un'importante libreria per applicazioni come giochi, crittografia e simulazioni. Per utilizzarla, bisogna prima aggiungere il crate *rand* alle dipendenze del progetto e poi importarla nel proprio codice. La libreria offre diversi metodi per la generazione di numeri casuali. L'oggetto chiave è *rand::Rng*, che permette di generare numeri interi, decimali e booleani casuali.

Ad esempio, per generare un numero casuale intero tra 1 e 100 si può utilizzare *rand::thread_rng().gen_range(1..101).* Se si desidera un numero float tra 0.0 e 1.0, si può usare *rand::thread_rng().gen::<f64>().* La funzione *thread_rng()* genera un generatore casuale locale al thread in uso, assicurando che i valori siano diversi a ogni esecuzione. È possibile anche riempire array o vettori con valori casuali, utilizzando metodi come *fill().* La libreria fornisce anche funzioni avanzate, come la generazione di numeri casuali sicuri per la crittografia con *rand::rngs::OsRng.* Osserviamo un esempio:

```
use rand::Rng;

fn main() {
    let mut rng = rand::thread_rng();
```

```
    let random_number: i32 = rng.gen_range(1..101);
    let random_float: f64 = rng.gen();

    println!("Numero intero casuale tra 1 e 100: {}", random_number);
    println!("Numero float casuale tra 0.0 e 1.0: {}", random_float);
}
```

La sintassi *rand::thread_rng()* inizializza un nuovo generatore casuale per il thread corrente, che può essere utilizzato per produrre numeri casuali di diverso tipo. Come in altri linguaggi, Rust utilizza un seme basato sul tempo di sistema per garantire l'unicità delle sequenze casuali a ogni esecuzione.

### *Quiz & Esercizi*

1) Scrivi un programma che prenda in input due numeri e stampi la loro somma.
2) Scrivi un programma che determini se un numero è pari o dispari.
3) Scrivi un programma che stampi i primi 10 numeri naturali.
4) Scrivi un programma che calcoli la media di 5 numeri inseriti dall'utente.
5) Scrivi un programma che prenda una stringa in input e la stampi al contrario.
6) Scrivi un programma che calcoli il fattoriale di un numero.
7) Scrivi un programma che verifichi se una stringa è palindroma.
8) Scrivi un programma che stampi tutti i numeri primi da 1 a 100.
9) Scrivi un programma che calcoli la potenza di un numero base e un esponente forniti dall'utente.
10) Scrivi un programma che stampi i primi 10 numeri della serie di Fibonacci.
11) Scrivi un programma che determini se un anno è bisestile.
12) Scrivi un programma che calcoli l'area di un cerchio dato il raggio.
13) Scrivi un programma che stampi la tabellina del 5 fino a 10.
14) Scrivi un programma che conti il numero di vocali in una stringa.
15) Scrivi un programma che calcoli la somma di tutti i numeri pari da 1 a 100.
16) Scrivi un programma che trovi il numero massimo in un array di numeri interi.
17) Scrivi un programma che inverta l'ordine degli elementi in un array di interi.
18) Scrivi un programma che conti il numero di elementi positivi, negativi e zeri in un array.
19) Scrivi un programma che calcoli il perimetro e l'area di un rettangolo dati la base e l'altezza.
20) Scrivi un programma che trovi il secondo numero più grande in un insieme di numeri interi.
21) Qual è la differenza tra una costante e una variabile?
22) Qual è la sintassi per dichiarare una variabile?

23) Qual è la sintassi per dichiarare una costante?
24) Qual è il tipo di dato che rappresenta un insieme di valori univoci e immutabili?
25) Qual è il tipo di dato che rappresenta un insieme di valori modificabili e indicizzati tramite chiavi?
26) Qual è il tipo di dato che rappresenta un valore booleano?
27) Quale costrutto viene utilizzato per iterare su una sequenza di valori?
28) Come si può interrompere un ciclo?
29) Qual è l'operatore di assegnazione in Rust?
30) Cosa fa la funzione *string*?

*N.B. tutte le soluzioni si scaricano con il QRcode nell'ultima pagina.*

***Riassunto***

Abbiamo esplorato diversi aspetti basilari del linguaggio, iniziando dai cicli come *loop, while* e *for*, evidenziando come Rust tenda a preferire gli iteratori rispetto ai loop classici. Gli iteratori, rispetto ai cicli, offrono maggiore sicurezza e ottimizzazione nella gestione delle collezioni, poiché permettono un controllo esplicito delle operazioni svolte su di esse.

Successivamente abbiamo trattato il concetto di typecasting, spiegando che Rust, essendo fortemente tipizzato, richiede conversioni esplicite tra tipi, a differenza di linguaggi come C#. Abbiamo discusso della differenza tra iterabili e iteratori, spiegando che un iterabile è una collezione di dati su cui si può iterare, mentre un iteratore è l'oggetto che esegue l'iterazione effettiva.

In seguito, abbiamo esplorato il sistema di gestione delle stringhe, esaminando operazioni comuni come ricerca di sottostringhe, sostituzione di caratteri, conversione tra maiuscole e minuscole, rimozione di spazi e concatenazione. Abbiamo visto anche le differenze tra il tipo *String* e *&str*, dove String è un oggetto mutabile e *heap-allocated*, mentre *&str* è un riferimento a una stringa immutabile e *stack-allocated*.

Le macro sono state introdotte come un potente strumento per la generazione di codice durante la compilazione. Ne esistono due tipi principali: di dichiarazione e di procedura. Le ultime, in particolare, permettono la trasformazione del codice tramite funzioni.

Abbiamo anche esplorato operatori aritmetici, logici e bit a bit, con esempi di codice che mostra come utilizzarli in vari contesti. Infine, abbiamo trattato il concetto di *scope* e *lifetime*, spiegando come Rust gestisce la visibilità e la durata delle variabili, garantendo la sicurezza della memoria tramite il *borrow checker*. Il *lifetime,* in particolare, è fondamentale per assicurare che le reference siano valide solo finché lo sono anche i dati a cui fanno riferimento.

# # 2 - L’espressione in Rust

*Il concetto di espressione*
*Lavorare con variabili e dichiarazioni*
*File e directory*
*La gestione del tempo*

Dopo il corposo primo capitolo, dove abbiamo descritto dettagliatamente le varie tipologie di dati, e, probabilmente anche messo tanta carne al fuoco, andremo adesso ad approfondire uno degli elementi di base di ogni linguaggio di programmazione, cioè l'espressione, necessaria a descrivere una relazione logica o matematica.
Le espressioni sono una componente fondamentale della programmazione in Rust. Rappresentano costrutti che combinano valori, operatori e chiamate a funzioni per formare unità di codice che restituiscono un risultato. Possono essere semplici, come una singola variabile o un valore costante, oppure più complesse, comprendendo operazioni aritmetiche, logiche o di confronto. Queste espressioni possono essere utilizzate in diversi contesti: assegnate a variabili, passate come argomenti a funzioni, impiegate come condizioni nelle istruzioni di controllo come *if* o *match*, e molto altro ancora. In pratica, costituiscono una parte essenziale della sintassi di Rust e svolgono un ruolo cruciale nella scrittura di codice chiaro, conciso ed efficiente. Vediamo un esempio:

```
let somma = 5 + 3 * (7 - 2);
```

In questa espressione, ci sono diversi valori (5, 3, 7, 2) e operatori (+, *, -) che vengono combinati per calcolare il valore della variabile *somma*. Come detto,

possono essere anche più complesse, coinvolgendo chiamate a funzioni, operatori logici e altro ancora.
Oltre a ciò, seguono regole precise riguardanti l'ordine di valutazione degli operatori, note come "precedenza degli operatori", e la loro associatività, che indica in che direzione vengono valutati quello con la stessa precedenza.
Solitamente gli operatori hanno un ordine di precedenza definito, che determina la sequenza in cui vengono eseguite le operazioni all'interno di un'espressione. Ad esempio, gli operatori aritmetici come * e / hanno una precedenza più alta rispetto a quelli di somma e sottrazione (+ e -), esattamente come nelle regole aritmetiche di base. Di conseguenza, all'interno di un'espressione le operazioni di moltiplicazione e divisione vengono eseguite prima di quelle di somma e sottrazione.
Quando ci sono più operatori con la stessa precedenza all'interno di un'espressione, l'associatività degli operatori determina in che ordine vengono valutati. Ad esempio, gli operatori aritmetici come + e - sono associativi da sinistra a destra, il che significa che vengono valutati nell'ordine in cui compaiono nell'espressione. Quindi, in un'espressione come a + b - c, a + b viene valutato prima di – c:

```
let risultato = 5 + 3 * (7 - 2);
```

L'operazione tra parentesi (7 - 2) viene eseguita per prima, poiché le parentesi hanno la massima precedenza. Il risultato di (7 - 2) è 5. Successivamente, viene eseguita l'operazione di moltiplicazione 3 * 5, ottenendo 15. Infine, viene eseguita l'operazione di somma 5 + 15, ottenendo 20, che viene assegnato alla variabile *risultato*.
Le espressioni possono diventare più complesse quando coinvolgono più operatori e valori, e l'ordine di valutazione può influenzare il risultato finale. Vediamo alcuni esempi:

```
let risultato = (5 + 3) * (7 - 2);
```

Qui abbiamo due operazioni: una somma tra 5 e 3 e una sottrazione tra 7 e 2. L'ordine è determinato dalle parentesi, quindi prima vengono eseguite le operazioni all'interno delle parentesi e poi viene effettuata la moltiplicazione. Il risultato sarà 8 * 5, che è uguale a 40.

```
let condizione = (3 * 4 < 5 + 2) && (10 / 2 >= 4);
```

In questo caso, ci sono due operazioni di confronto, una moltiplicazione, una somma e una divisione. L'ordine di valutazione è determinato dagli operatori logici && (AND). Le operazioni all'interno delle parentesi vengono eseguite per prime e poi viene eseguita la valutazione degli operatori logici. Se entrambe le condizioni sono vere, il risultato sarà *true*, altrimenti sarà *false*.
Questi sono solo alcuni esempi di espressioni più complesse, in cui l'ordine di valutazione è cruciale per ottenere il risultato corretto. Le parentesi possono essere utilizzate per chiarire l'ordine e garantire che le operazioni vengano eseguite nel modo desiderato. È essenziale comprendere questi concetti per scrivere codice corretto e prevedibile.
Negli esempi precedenti, abbiamo utilizzato variabili rappresentanti numeri interi come oggetti su cui eseguire gli operatori. Tuttavia, le espressioni possono essere utilizzate in senso generale, mentre gli operatori possono essere applicati a vari tipi di dati, come ad esempio stringhe (testo) o valori booleani (*true* o *false*). Tuttavia, è importante notare che non tutte le tipologie di dati possono interagire tra loro; ad esempio, *non è possibile sommare una stringa con un numero*.
Osserviamo alcuni esempi che illustrano l'utilizzo delle espressioni con vari tipi di dati:

```
// Concatenazione di stringhe
let nome = "Alice";
let saluto = "Ciao, ".to_string() + nome + "!";
// Il risultato sarà "Ciao, Alice!"

// Utilizzo di operatori di confronto con valori booleani
```

```
let condizione = (3 < 5) && (10 >= 4);
// Il risultato sarà true, poiché entrambe le condizioni sono vere

// Utilizzo di espressioni condizionali con valori booleani
let x = 10;
let messaggio = if x > 5 { "x è maggiore di 5" } else { "x è minore o uguale a 5" };
// Il risultato sarà "x è maggiore di 5"
```

In pratica, le espressioni vengono utilizzate per combinare valori e operatori per eseguire operazioni specifiche. Nell'esempio della concatenazione di stringhe, l'operatore + viene utilizzato per unire diverse stringhe e creare un nuovo valore di tipo *String*. Nel seguente, gli operatori <, >, <=, >= vengono utilizzati per confrontare valori numerici e restituire valori booleani. Nel terzo, l'istruzione *if-else* viene utilizzata per valutare una condizione e restituire un valore diverso in base al risultato.
Vediamo ora un esempio più complesso, come un'espressione condizionale, che è una forma compatta per valutare una condizione e restituire un valore diverso in base a essa. Questo tipo di espressione è comunemente utilizzato per assegnare un valore a una variabile in base a una condizione:

```
let x = 10;
let risultato = if x > 5 { "x è maggiore di 5" } else { "x non è maggiore di 5" };
// Il risultato sarà "x è maggiore di 5"
```

Qui, l'espressione condizionale *if x > 5 { "x è maggiore di 5" } else { "x non è maggiore di 5" }* valuta la condizione x > 5. Se la condizione è vera, viene restituito il valore "x è maggiore di 5", altrimenti "x non è maggiore di 5". Questo valore viene quindi assegnato alla variabile risultato.

```
let temperatura = 25;
let stato = if temperatura > 30 { "caldo" } else { "fresco" };
// In questo caso, se la temperatura è superiore a 30 gradi, la variabile `stato` sarà assegnata
alla stringa "caldo", altrimenti verrà assegnata alla stringa "fresco".
```

Le espressioni condizionali sono utili quando si desidera assegnare un valore a una

variabile in base a una condizione semplice, senza dover scrivere un'intera istruzione *if-else.* Sono una forma compatta e leggibile per gestire logicamente i valori all'interno del codice. Queste espressioni descrivono il calcolo secondo cui il risultato dovrebbe essere uno o un altro valore a seconda di una condizione.
L'istruzione *if* in Rust può essere utilizzata in modo più completo per gestire condizioni complesse, eseguendo una serie di istruzioni in base al risultato di una condizione. Ecco un esempio di utilizzo:

```
let temperatura = 25;
if temperatura > 30 {
    println!("È caldo!");
}
```

Se la temperatura è superiore a 30 gradi, viene eseguita l'istruzione all'interno delle parentesi graffe, che in questo caso specifico è una chiamata alla funzione *println!* per stampare il messaggio "È caldo!".
Possiamo anche utilizzare l'istruzione *if-else* per gestire due casi diversi:

```
let temperatura = 25;
if temperatura > 30 {
    println!("È caldo!");
} else {
    println!("Non è così caldo!");
}
```

In questo secondo esempio, se la temperatura è superiore a 30 gradi, viene eseguita l'istruzione all'interno del blocco *if.* Altrimenti, viene eseguita l'istruzione all'interno del blocco *else*.
Le espressioni condizionali con le istruzioni *if* o *if-else* sono fondamentali per gestire situazioni in cui è necessario eseguire codice in base a una condizione specifica. Consentono di scrivere codice che si adatta dinamicamente alle variazioni nelle condizioni del programma.
Andiamo avanti. In Rust, come in altri linguaggi, esistono diversi tipi di numeri, tra cui gli interi (i32, i64, etc.), i numeri in virgola mobile (f32, f64), e numeri

complessi, anche se il linguaggio non ha un supporto nativo diretto per i numeri complessi. I numeri in virgola mobile (f64 per esempio) rappresentano valori decimali e possono quindi memorizzarne una frazione. Osserviamo un'espressione in cui vengono confrontati due numeri decimali, dove il primo è scritto nel classico formato (cioè 100.5), mentre l'altro è il risultato di un'elevazione a potenza:

```
let num1: f64 = 100.5;
let num2: f64 = 8.5_f64.powi(2); // Elevazione a potenza
let result: bool = num1 > num2; // true
println!("{}", result);
```

Perciò, utilizziamo f64, il tipo più comune per numeri in virgola mobile, e la funzione *powi()* per elevare 8.5 alla potenza di 2.

Per quanto riguarda i numeri complessi, non abbiamo un supporto nativo diretto come in altri linguaggi. Tuttavia, esistono *crate* esterni come *num-complex* che permettono di gestire numeri complessi. Un numero complesso è una coppia di numeri, dove il primo rappresenta la parte reale e il secondo la parte immaginaria. La parte immaginaria corrisponde alla radice quadrata di -1.

NB. I *crate* permettono di riutilizzare codice esterno e di organizzare le dipendenze in modo efficiente, fornendo un meccanismo standardizzato per includere e gestire librerie di terze parti o sviluppare codice in moduli separati e poi integrarli. Ne parleremo più avanti.

Ecco un esempio di come si potrebbero gestire numeri complessi utilizzando il crate *num-complex*:

```
use num_complex::Complex;

fn main() {
    let complex_number = Complex::new(3.0, 2.0);
    let result = complex_number.powf(2.0);
    let comparison = result.norm() < 20.0; // true
    println!("{}", comparison);
}
```

Quindi *num-complex* viene utilizzato per creare e manipolare numeri complessi. La funzione *powf()* eleva il numero complesso alla potenza indicata, mentre *norm()*

restituisce la magnitudine del numero complesso.
Oltre a queste operazioni, possiamo anche eseguire confronti concatenati. Consideriamo un esempio in cui si combinano una condizione booleana e un confronto numerico:

```
let y: bool = true; // esempio di valore, può essere qualsiasi condizione
let x: f64 = 3.0;  // esempio di valore, può essere qualsiasi numero
let comparison = y && (x < 5.0);
println!("{}", comparison);
```

Questa espressione può essere riscritta in modo più compatto:

```
let chain_comparison = (y == true) && (x < 5.0);
println!("{}", chain_comparison);
```

Entrambe le espressioni sono equivalenti, con la differenza che la seconda è più esplicita riguardo alla condizione *y == true*, anche se non è strettamente necessario in Rust, poiché *y* è già un valore booleano.
Il linguaggio supporta anche la concatenazione di confronti più complessi:

```
let a = 5;
let b = 10;
let c = 15;
let chained = (a < b) && (b < c);
println!("{}", chained); // true
```

In questo esempio, i confronti vengono combinati in una singola espressione booleana, utile per verificare una sequenza di condizioni. Questo tipo di notazione è utile per mantenere il codice leggibile e conciso.

### *Gli identificatori*

Gli identificatori nelle espressioni sono nomi utilizzati per identificare variabili, costanti, funzioni, strutture, enumerazioni, moduli e altri elementi del codice. Devono seguire determinate regole e convenzioni per essere validi e comprensibili nel contesto del linguaggio di programmazione. In Rust, esiste una convenzione

standard per la loro denominazione, quelle comuni includono:

*snake_case* per i nomi di variabili e funzioni (ad esempio *nome_variabile, calcola_valore*).
*PascalCase* per i nomi di strutture, enumerazioni e tipi (ad esempio *NomeStruttura, TipoDato*).
*SCREAMING_SNAKE_CASE* per costanti e variabili statiche (ad esempio *VALORE_COSTANTE*).

Gli identificatori devono iniziare con una lettera o un carattere di sottolineatura (_) e possono contenere lettere, cifre e caratteri di sottolineatura. Non possono iniziare con una cifra, non possono contenere spazi o caratteri speciali, e sono sensibili alle maiuscole e minuscole, il che significa che *nome_variabile* e *NomeVariabile* sono considerati identificatori diversi.
È importante notare che, per evitare confusione, gli identificatori non possono essere parole chiave riservate del linguaggio. Ad esempio, non è possibile utilizzare *let* o *fn*. Inoltre, devono essere univoci nel contesto in cui vengono utilizzati. Non è possibile utilizzare lo stesso identificatore per diverse variabili nello stesso blocco di codice:

```
let valore = 10;
let messaggio = "Il valore è:";
println!("{} {}", messaggio, valore);
```

In questo esempio, valore e messaggio sono identificatori utilizzati per definire rispettivamente una variabile di tipo intero e una variabile di tipo stringa. Gli identificatori vengono poi utilizzati per fare riferimento a queste variabili all'interno dell'espressione.
Oltre alle caratteristiche di base, esistono altri aspetti importanti da considerare quando si lavora con gli identificatori nelle espressioni:

- *Nomi significativi*: gli identificatori dovrebbero essere scelti in modo da essere significativi e descrittivi del loro scopo nel contesto del programma. Questo aiuta a rendere il codice più leggibile e comprensibile per altri sviluppatori che leggono il codice.

- *Portata (scope) degli identificatori*: determina dove possono essere utilizzati nel codice. Le variabili

definite all'interno di un blocco ({}) sono accessibili solo al suo interno, mentre le variabili definite all'esterno, come in un modulo o una funzione, possono essere accessibili in tutto il modulo o la funzione, a seconda della visibilità.

- *Identificatori globali e locali*: i primi, come le costanti o variabili statiche, sono definiti fuori da qualsiasi blocco di codice e sono accessibili da tutto il programma. Gli identificatori locali, definiti all'interno di un blocco di codice, sono accessibili solo all'interno di quel blocco.

- *Ridefinizione degli identificatori*: è possibile ridefinire un identificatore all'interno di un blocco di codice (shadowing), ma questo può causare confusione nel codice. Rust permette lo *shadowing* intenzionale, che può essere utile per mutare una variabile non mutabile, ma è buona pratica evitare di ridefinire gli identificatori nella stessa portata, a meno che non ci sia una chiara ragione.
Esempio:

```
let x = 5;
let x = x + 1; // 'x' viene ridefinito
println!("{}", x); // Stampa '6'
```

- *Commenti sugli identificatori*: è consigliabile aggiungere commenti significativi agli identificatori per spiegare il loro scopo e la loro utilità nel codice. Questo aiuta a facilitare la comprensione del codice da parte di altri sviluppatori e facilita la manutenzione del codice nel tempo.

In sintesi, gli identificatori sono elementi fondamentali nelle espressioni e devono essere scelti con cura e significato per garantire la leggibilità, la manutenibilità e la comprensibilità del codice. Considerare la portata, la pratica dello *shadowing*, e l'aggiunta di commenti significativi sono pratiche consigliate quando si lavora in questo contesto in Rust.
Osserviamo un esempio di codice che illustra le caratteristiche appena discusse:

```
// Definizione di una struttura per rappresentare un contatore semplice
struct Contatore {
    // Campo privato, accessibile solo all'interno della struttura
    valore_contatore: i32,
}

impl Contatore {
    // Costruttore della struttura per inizializzare il contatore
```

```
    fn new() -> Contatore {
        // Inizializzazione del valore del contatore a zero
        Contatore { valore_contatore: 0 }
    }

    // Metodo pubblico per incrementare il contatore
    fn incrementa(&mut self) {
        self.valore_contatore += 1; // Incrementa il valore del contatore di uno
    }

    // Metodo pubblico per ottenere il valore attuale del contatore
    fn ottieni_valore(&self) -> i32 {
        self.valore_contatore // Restituisce il valore attuale del contatore
    }
}

fn main() {
    // Creazione di un'istanza della struttura Contatore
    let mut mio_contatore = Contatore::new();

    // Incremento del contatore
    mio_contatore.incrementa();
    mio_contatore.incrementa();

    // Ottenimento del valore attuale del contatore
    let valore_attuale = mio_contatore.ottieni_valore();

    // Visualizzazione del valore attuale del contatore
    println!("Il valore attuale del contatore è: {}", valore_attuale);

    // Esempio di variabile locale
    let somma = valore_attuale + 10; // Somma del valore del contatore con 10

    // Visualizzazione della somma
    println!("La somma del valore del contatore con 10 è: {}", somma);
}
```

Gli identificatori utilizzati nel codice sono descrittivi e riflettono chiaramente il loro scopo: *Contatore* è la struttura che rappresenta un contatore, *valore_contatore* è il campo privato che ne contiene il valore, accessibile solo all'interno della struttura. *new*, *incrementa* e *ottieni_valore* sono metodi associati alla struttura,

rispettivamente utilizzati per creare un nuovo contatore, incrementarlo e ottenerne il valore attuale. Il campo *valore_contatore* è privato per convenzione, in Rust, tutti i campi di una struttura sono privati per default. I metodi *incrementa* e *ottieni_valore* sono pubblici all'interno dell'implementazione (*impl*) della struttura *Contatore*, quindi accessibili tramite le istanze della struttura.
Sappiamo già che Rust non utilizza la parola chiave *public* per la visibilità, ma la visibilità predefinita dei campi è privata, mentre i metodi associati possono essere richiamati tramite le istanze della struttura.
La variabile *mio_contatore* è dichiarata mutabile (*mut*) poiché i metodi che modificano lo stato interno della struttura (*incrementa*) richiedono un riferimento mutabile a *self.* Al contrario, *ottieni_valore* utilizza un riferimento immutabile poiché non modifica lo stato.
Le variabili *valore_attuale* e *somma* sono dichiarate localmente all'interno della funzione *main*, rendendole accessibili solo al suo interno. Non vi è alcuna ridefinizione degli identificatori all'interno della stessa portata, mantenendo il codice chiaro e privo di ambiguità.
Infine, sono presenti commenti che spiegano il significato e l'uso degli identificatori, rendendo il codice più facile da comprendere e manutenere.
Questo esempio dimostra come in Rust gli identificatori possano essere utilizzati in modo efficace per creare un programma chiaro, leggibile e conforme alle convenzioni del linguaggio. Inoltre, l'uso di strutture, metodi e visibilità consente di organizzare il codice in modo modulare e sicuro.

### *Espressioni condizionali e strutture di controllo*

Le espressioni condizionali e le strutture di controllo sono strumenti essenziali per gestire il flusso del programma. Un elemento chiave è che in Rust quasi tutto è un'espressione, il che significa che non solo il flusso di controllo può decidere il comportamento del programma, ma può anche restituire un valore.
L'istruzione *if,* ad esempio, è usata per eseguire blocchi di codice in base a

condizioni specifiche. Tuttavia, poiché è un'espressione, può essere utilizzato per restituire valori, rendendo il codice più conciso ed espressivo. Consideriamo questo esempio:

```
fn main() {
    let numero = 5;
    let risultato = if numero % 2 == 0 {
        "pari"
    } else {
        "dispari"
    };
    println!("Il numero è {}", risultato);
}
```

In questo caso, l'espressione *if* restituisce una stringa in base al valore di numero. Se numero è divisibile per 2, restituisce "pari", altrimenti restituisce "dispari". Il valore risultante viene assegnato alla variabile risultato e stampato. Questo mostra come *if* possa essere utilizzato non solo per controllare il flusso, ma anche per costruire valori in modo conciso.

Un altro aspetto importante è che, a differenza di molti altri linguaggi, Rust non richiede le parentesi attorno alla condizione dell'*if*, ma obbliga a usare le parentesi graffe per delimitare i blocchi di codice. Questa scelta rende più chiara e sicura la struttura del programma, evitando ambiguità.

Oltre all'*if*, esiste la struttura *match*, di cui abbiamo già discusso nel precedente capitolo e che è molto più potente dello *switch* di altri linguaggi. Permette di confrontare un valore con diversi pattern, eseguendo il codice associato al primo che corrisponde. È particolarmente utile quando si ha a che fare con enumerazioni o con tipi come *Option* e *Result*. Ad esempio, supponiamo di voler gestire i diversi casi di un *Option*:

```
fn main() {
    let valore: Option<i32> = Some(10);

    let risultato = match valore {
```

```
        Some(v) => v + 1,
        None => 0,
    };

    println!("Il risultato è {}", risultato);
}
```

Qui, *match* viene utilizzato per gestire un'*Option* che può contenere un valore (*Some(v)*) o essere vuota (*None*). Nel primo caso, viene incrementata di 1, altrimenti viene restituito 0. Il sistema di tipi garantisce che tutti i casi vengano coperti, rendendo il codice robusto e sicuro.
Inoltre, Rust supporta il cosiddetto "pattern matching" all'interno delle espressioni *match*, il che significa che è possibile controllare condizioni molto complesse e destrutturare valori direttamente. Ad esempio, si può usare *match* con tuple per distruggere i valori e fare verifiche multiple:

```
fn main() {
    let coppia = (2, 3);

    match coppia {
        (x, y) if x == y => println!("I numeri sono uguali."),
        (x, y) => println!("I numeri sono diversi: {} e {}", x, y),
    }
}
```

Qui, *match* è usato per confrontare i due elementi della tuple. Se i numeri sono uguali, stampa un messaggio; altrimenti, mostra i numeri diversi. L'uso di *if* all'interno di match per aggiungere condizioni ai pattern offre ulteriore flessibilità.
Per quanto riguarda i cicli, Rust offre diverse opzioni. Il *loop* è il più semplice e ripete il blocco di codice indefinitamente finché non si usa una condizione di uscita esplicita, come *break*. Ad esempio:

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

    loop {
```

```
        count += 1;
        if count == 5 {
            break;
        }
    }

    println!("Il conteggio è {}", count);
}
```

In questo caso, il ciclo si interrompe solo quando *count* raggiunge 5. Anche qui, l'uso di *loop* mostra come Rust preferisca meccanismi espliciti per il controllo del flusso rispetto a costrutti impliciti o non controllati.
Oltre a loop, è supportato anche *while*, che continua a eseguire il blocco di codice finché una condizione è vera. Questo è simile ad altri linguaggi:

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

    while numero < 5 {
        println!("Numero: {}", numero);
        numero += 1;
    }
}
```

Il ciclo esegue il blocco finché numero è inferiore a 5, incrementando numero a ogni iterazione.
Infine, c'è il ciclo *for*, che è utilizzato per iterare su una collezione o un intervallo. Questo è il ciclo più sicuro e idiomatico in Rust, poiché evita i problemi legati agli indici e gestisce automaticamente il fine della collezione. Consideriamo questo esempio in cui si itera su un intervallo di numeri:

```
fn main() {
    for numero in 1..5 {
        println!("Numero: {}", numero);
    }
}
```

Qui, *for* itera sui numeri da 1 a 4 (l'intervallo 1..5 è esclusivo, quindi non include 5). for può anche essere usato per iterare su collezioni come vettori, con l'uso di pattern matching o iteratori per elaborare i valori.

### *Espressioni lambda o chiusure*

Le espressioni lambda in Rust, spesso chiamate "chiusure" (*closures*), sono funzioni anonime che possono catturare e utilizzare variabili dal loro contesto circostante. Queste chiusure sono potenti e flessibili, consentendo di scrivere codice conciso e modulare. A differenza delle funzioni tradizionali, le chiusure possono catturare variabili dal loro ambiente in tre modi: per valore, per riferimento immutabile, o per riferimento mutabile. Non vi preoccupate se questi concetti al momento potrebbero sembrare poco chiari, ne parleremo approfonditamente nel prossimo capitolo dedicato proprio alle funzioni.

Una chiusura viene definita utilizzando la sintassi |*parametri*| *espressione*. Ad esempio, supponiamo di avere una lista di numeri interi e vogliamo filtrare quelli maggiori di un certo valore. Possiamo usare una chiusura per eseguire questo filtro:

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

    let numeri_filtrati: Vec<i32> = numeri.into_iter()
        .filter(|&x| x > soglia)
        .collect();

    println!("{:?}", numeri_filtrati); // Output: [4, 5]
}
```

La chiusura |*&x*| *x* > *soglia* filtra i numeri maggiori di *soglia*. Questa variabile viene catturata per valore, poiché non è mutabile ed è semplicemente letta all'interno dell'espressione.

Le chiusure possono anche essere utilizzate per operazioni più complesse, come

modificare gli elementi di una lista. Immaginiamo di volerne moltiplicare ogni elemento per un fattore specifico:

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

    numeri.iter_mut().for_each(|x| *x *= fattore);

    println!("{:?}", numeri); // Output: [2, 4, 6, 8, 10]
}
```

In questo caso, la chiusura *|x| *x *= fattore* cattura *fattore* per valore e modifica direttamente gli elementi della lista usando un riferimento mutabile a ciascun elemento.

Un aspetto importante delle chiusure è la loro capacità di essere utilizzate come argomenti per altre funzioni o essere restituite da funzioni. Ad esempio, possiamo definire una funzione che restituisce una chiusura:

```
fn crea_incrementatore(incremento: i32) -> impl Fn(i32) -> i32 {
    move |x| x + incremento
}

fn main() {
    let incrementa_di_due = crea_incrementatore(2);
    let risultato = incrementa_di_due(5);
    println!("{}", risultato); // Output: 7
}
```

Qui, la funzione *crea_incrementatore* restituisce una chiusura che incrementa un numero di una quantità specificata. La parola chiave *move* forza la chiusura a catturare incremento per *valore*, trasferendo la proprietà all'interno della chiusura. Questo è utile quando si vuole garantire che la variabile catturata non cambi al di fuori della chiusura o quando la chiusura deve essere utilizzata in un contesto asincrono o concorrente.

Un altro esempio interessante è l'uso delle chiusure con funzioni di ordine

superiore come *map, filter,* e *fold*, che permettono di eseguire operazioni complesse in modo dichiarativo e conciso. Ad esempio, sommare tutti i numeri di una lista:

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

    let somma: i32 = numeri.iter().fold(0, |acc, &x| acc + x);

    println!("{}", somma); // Output: 15
}
```

La chiusura |*acc, &x*| *acc + x* somma ogni elemento della lista al valore accumulato *acc*, partendo da 0. Questo esempio mostra come le chiusure possono essere utilizzate in modo efficiente per ridurre una sequenza di valori a un singolo risultato.
Abbiamo dimostrato che le chiusure sono potenti strumenti le quali permettono di catturare il contesto in cui sono definite, consentendo una programmazione funzionale e concisa, particolarmente utile quando si lavora con iterazioni, trasformazioni di dati, o operazioni su collezioni. La capacità di catturare variabili dal contesto circostante le rende molto flessibili e adattabili a una vasta gamma di situazioni, mantenendo al contempo il rigore e la sicurezza del sistema di proprietà del linguaggio.

### *La gestione delle eccezioni*

In Rust, la gestione delle eccezioni è affrontata in modo diverso rispetto a molti altri linguaggi di programmazione come C# o Java, che utilizzano meccanismi di *try-catch* per intercettare e gestire errori. Qui non c'è un sistema di gestione delle eccezioni basato su questo costrutto, esiste invece un approccio orientato al risultato e alla gestione esplicita degli errori. Questa metodologia è progettata per promuovere la sicurezza e la prevedibilità del codice, riducendo il rischio di errori non gestiti.

Quindi, gli errori vengono gestiti principalmente attraverso il tipo di dato *Result<T, E>,* che rappresenta un'operazione che può avere successo e restituire un valore di tipo T, o fallire e restituire un errore di tipo E. Questa gestione esplicita obbliga il programmatore a considerare ogni possibile esito di un'operazione, migliorando la robustezza del codice.
Ad esempio, supponiamo di voler leggere il contenuto di un file. La funzione *std::fs::read_to_string* restituisce un *Result<String, std::io::Error>,* dove *String* è il tipo di dato atteso in caso di successo, e *std::io::Error* rappresenta il tipo di errore in caso di fallimento. Vediamo come questo può essere gestito:

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

fn leggi_file(nome_file: &str) -> Result<String, io::Error> {
    let mut file = File::open(nome_file)?;
    let mut contenuto = String::new();
    file.read_to_string(&mut contenuto)?;
    Ok(contenuto)
}

fn main() {
    match leggi_file("testo.txt") {
        Ok(contenuto) => println!("Contenuto del file:\n{}", contenuto),
        Err(errore) => eprintln!("Errore nella lettura del file: {}", errore),
    }
}
```

La funzione *leggi_file* cerca di aprire un file e leggerne il contenuto. Se l'operazione ha successo, restituisce il contenuto come *String* incapsulato in un *Ok*. In caso di errore, come l'assenza del file, il programma restituisce un *Err* contenente la problematica. La gestione avviene utilizzando un *match,* che consente di distinguere tra i due casi e di gestirli di conseguenza.
Il simbolo ? utilizzato dopo l'invocazione di *File::open* e *read_to_string* è una sintassi concisa per propagare l'errore al chiamante. Se la funzione chiamata restituisce un *Err,* l'errore viene immediatamente restituito dalla funzione che

utilizza ?, senza bisogno di scrivere esplicitamente un blocco *match*.
Spieghiamo bene cosa è un *Ok*. Parliamo di una variante dell'enum *Result,* che viene utilizzata per rappresentare operazioni che possono riuscire o fallire. Quando un'operazione va a buon fine, viene restituito Ok, insieme al valore risultante dall'operazione. Se invece l'operazione fallisce, viene restituito *Err*, che contiene un'informazione sull'errore.
Il tipo *Result<T, E>* ha due varianti: *Ok(T)* per rappresentare un successo, dove T è il tipo del risultato desiderato, e *Err(E)* per rappresentare un fallimento, dove E è il tipo dell'errore. Usando Ok, puoi indicare che l'operazione ha prodotto un risultato valido. Ad esempio, supponiamo di voler eseguire una divisione che potrebbe fallire se si tenta di dividere per zero. Possiamo usare *Result* per gestire questo scenario:

```
fn dividi(a: i32, b: i32) -> Result<i32, String> {
    if b == 0 {
        Err(String::from("Errore: divisione per zero"))
    } else {
        Ok(a / b)
    }
}

fn main() {
    match dividi(10, 2) {
        Ok(valore) => println!("Il risultato è {}", valore),
        Err(e) => println!("{}", e),
    }

    match dividi(10, 0) {
        Ok(valore) => println!("Il risultato è {}", valore),
        Err(e) => println!("{}", e),
    }
}
```

Nel programma, *dividi* restituisce *Ok* quando la divisione ha successo e *Err* quando si verifica un errore (come la divisione per zero). Il *match* è utilizzato per controllare il valore restituito: se è *Ok,* viene stampato il risultato; se è *Err*, viene

gestito l'errore.
Rust offre anche un tipo *Option<T>* per gestire i casi in cui un valore potrebbe essere presente o meno, senza dover ricorrere ad errori veri e propri. Tuttavia, per le situazioni in cui c'è bisogno di gestire errori veri e propri, il tipo *Result* è la scelta appropriata.
Se vogliamo gestire un'operazione che può fallire senza dover immediatamente propagare l'errore, possiamo utilizzare *match, unwrap* o *expect*. Gli ultimi due vengono utilizzati per estrarre il valore da un *Result* o un *Option*, causando un *panic* (cioè un'uscita immediata del programma) se il risultato è un errore o *None*. *expect* è preferibile a *unwrap* poiché permette di specificare un messaggio di errore personalizzato:

```
fn main() {
    let risultato = leggi_file("inesistente.txt").expect("Impossibile leggere il file");
    println!("Contenuto del file:\n{}", risultato);
}
```

In questo caso, se il file non esiste, il programma andrà in *panic* e stamperà il messaggio specificato in *expect*, fornendo un feedback immediato e chiaro su cosa sia andato storto.
In Rust, i *panic* sono considerati un meccanismo per gestire errori irreversibili, che dovrebbero essere utilizzati solo in situazioni in cui non è possibile recuperare l'esecuzione in modo sicuro. Il linguaggio promuove la gestione degli errori tramite *Result* per garantire che i programmi siano resilienti e che gli errori siano gestiti in modo prevedibile e sicuro.
Questo approccio alla gestione delle eccezioni, che evita il tradizionale *try-catch* a favore di un controllo esplicito tramite *Result,* incoraggia uno stile di programmazione che rende gli errori evidenti e obbliga a gestirli, portando a un codice più sicuro e robusto.
Continuando a esplorare la gestione degli errori, è importante comprendere come questa si integri con la filosofia generale del linguaggio, che è incentrata sulla

sicurezza e sul controllo esplicito delle risorse. Mentre in altri è comune lasciare che eccezioni non gestite si propaghino attraverso il *call stack*, Rust obbliga i programmatori a trattare gli errori in modo esplicito. Questa strategia evita che vengano ignorati accidentalmente e riduce la probabilità di bug difficili da individuare.

Un aspetto chiave è la distinzione tra errori recuperabili e non recuperabili. I primi sono rappresentati dal tipo *Result,* come discusso, mentre gli errori non recuperabili sono gestiti tramite il panico (*panic!*), che forza il programma a terminare. Quindi si utilizza quando si incontra un problema che non può essere risolto o ignorato, come accedere a un indice fuori dai limiti di un array. Anche se *panic!* può sembrare simile al lancio di un'eccezione in altri linguaggi, in Rust il suo uso è limitato a situazioni in cui l'unica risposta sensata è terminare l'esecuzione.

Per evitare l'uso eccessivo di *panic!,* che potrebbe portare a un codice meno robusto, Rust incoraggia l'uso di *Result* per gestire gli errori recuperabili. Ad esempio, consideriamo un'operazione che potrebbe fallire, come il *parsing* di un numero da una stringa. Il metodo *parse* restituisce un *Result,* che ci obbliga a considerare sia il caso di successo sia quello di fallimento:

```
fn main() {
    let numero: Result<i32, _> = "42".parse();
    match numero {
        Ok(n) => println!("Numero parsed correttamente: {}", n),
        Err(e) => println!("Errore nel parsing: {}", e),
    }
}
```

In questo esempio, la stringa "42" viene convertita in un numero intero utilizzando *parse*. Se la conversione ha successo, *Ok* contiene il valore; in caso contrario, *Err* contiene l'errore, e possiamo gestirlo nel blocco *match*. Questa forma di gestione esplicita ci dà il controllo totale su cosa fare in caso di errore.

Un'altra caratteristica interessante è la possibilità di "collegare" più operazioni che possono fallire utilizzando il simbolo ?. Questa sintassi rende il codice più conciso e

leggibile, propagando l'errore automaticamente se una delle operazioni fallisce. Consideriamo il seguente esempio in cui tentiamo di aprire un file e leggerne il contenuto:

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

fn leggi_contenuto(nome_file: &str) -> Result<String, io::Error> {
    let mut file = File::open(nome_file)?;
    let mut contenuto = String::new();
    file.read_to_string(&mut contenuto)?;
    Ok(contenuto)
}

fn main() {
    match leggi_contenuto("esempio.txt") {
        Ok(contenuto) => println!("Contenuto del file:\n{}", contenuto),
        Err(e) => eprintln!("Errore: {}", e),
    }
}
```

Perciò il simbolo ? semplifica il controllo degli errori. Se *File::open* o *read_to_string* falliscono, l'errore viene propagato automaticamente verso l'alto, e la funzione *leggi_contenuto* restituirà immediatamente un *Err.* Questo evita la necessità di scrivere ripetutamente blocchi *match* per ogni singola operazione.
Rust supporta anche l'uso di *map* e *and_then* per gestire in modo funzionale le operazioni su *Result*. Questi metodi permettono di applicare funzioni ai valori contenuti in *Ok* o di concatenare più operazioni che possono fallire. Per esempio, se volessimo convertire il contenuto del file in maiuscolo solo se la lettura ha successo, potremmo usare *map*:

```
fn leggi_e_converti(nome_file: &str) -> Result<String, io::Error> {
    let contenuto = leggi_contenuto(nome_file)?;
    Ok(contenuto.to_uppercase())
}
```

Qui, *to_uppercase* viene applicato solo se *leggi_contenuto* restituisce un *Ok*. In

caso di errore, l'errore viene propagato come al solito.
In alcuni casi, potrebbe essere utile convertire un *Result* in un *Option* per gestire situazioni in cui il fallimento non è considerato critico. Questo può essere fatto utilizzando il metodo *ok()* su un *Result*, che converte un *Ok* in *Some* e un *Err* in *None*. Questo approccio è utile quando non ci interessa l'errore specifico e vogliamo semplicemente sapere se un'operazione è riuscita o meno.
Un altro strumento per la gestione degli errori è la macro *panic!,* utilizzata per indicare che qualcosa è andato irrimediabilmente storto e il programma deve terminare. Sebbene il suo uso dovrebbe essere limitato a casi eccezionali, è importante sapere come e quando usarlo. Ad esempio, potremmo voler far fallire un programma se una condizione critica non è soddisfatta:

```
fn dividi(a: i32, b: i32) -> i32 {
    if b == 0 {
        panic!("Divisione per zero!");
    }
    a / b
}

fn main() {
    let risultato = dividi(10, 0);
    println!("Risultato: {}", risultato);
}
```

Perciò, se *b* è zero, il programma va in *panic* e si arresta immediatamente, indicando che è stato commesso un errore grave e non recuperabile. Anche se Rust evita di ricorrere frequentemente a *panic!,* ci sono situazioni in cui è l'unico modo di procedere.
L'approccio alla gestione degli errori, attraverso *Result, Option* e *panic!,* offre un equilibrio tra sicurezza e flessibilità. Richiede che i programmatori affrontino esplicitamente ogni possibile errore, promuovendo la scrittura di codice più robusto e affidabile. Questo non solo migliora la qualità del software, ma anche la sua manutenibilità, rendendo più semplice individuare e correggere eventuali problemi.

| Tipo | Utilizzo | Esempio |
|---|---|---|
| `Result (Ok/Err)` | *Utilizzato per rappresentare il risultato di un'operazione che può avere successo (Ok) o fallire (Err).* | `fn dividi(a: i32, b: i32) -> Result<i32, String> { if b == 0 { Err(String::from("Divisione per zero")) } else { Ok(a / b) } }` |
| `Option (Some/None)` | *Usato per indicare la presenza (Some) o l'assenza (None) di un valore.* | ```` ```fn trova_valore(valori: &[i32], target: i32) -> Option<usize> { valori.iter().position( ```` |
| `panic!` | *Utilizzato per errori irrecuperabili che terminano il programma immediatamente.* | `fn errore_fatale() { panic!("Errore fatale!") }` |
| `unwrap()/expect()` su Result o Option | *Metodo utilizzato per ottenere il valore se è presente, causando un panic in caso di errore (Err) o assenza di valore (None).* | `let risultato = dividi(10, 0).unwrap(); // Panica poiché la divisione per zero fallisce` |
| `? (Propagazione degli errori)` | *Usato per propagare errori verso il chiamante senza interrompere il flusso del programma.* | `fn leggi_file(path: &str) -> Result<String, io::Error> { let contenuto = fs::read_to_string(path)?; Ok(contenuto) }` |
| `Box<dyn std::error::Error>` | *Usato per rappresentare errori generici che implementano il trait std::error::Error.* | `fn errore_generico() -> Result<(), Box<dyn std::error::Error>> { Err("Errore generico".into()) }` |

### *La gestione di file e directory*

La gestione di file e directory viene realizzata attraverso il modulo standard *std::fs*, che fornisce tutte le funzionalità necessarie per lavorare con il filesystem, come la lettura e la scrittura di file, la creazione di directory, e la manipolazione di percorsi. A differenza di altri linguaggi, in Rust l'uso delle espressioni è centrale anche in questo contesto, perché molte operazioni restituiscono un risultato che può essere gestito tramite *Result,* permettendo così di integrare in modo fluido il controllo degli errori nel flusso del programma.

Per esempio, per leggere il contenuto di un file, si utilizza il metodo *fs::read_to_string*, che restituisce un *Result<String, io::Error>.* Questo significa che l'operazione può avere successo, restituendo il contenuto del file (una *String*), o fallire, restituendo un errore di tipo *io::Error*. L'utilizzo di espressioni come il pattern matching o l'operatore ? permette di gestire questi casi senza dover scrivere esplicitamente strutture di controllo come *if-else*.

Ecco un esempio di lettura di un file:

```
use std::fs;
use std::io;

fn leggi_contenuto_file(percorso: &str) -> Result<String, io::Error> {
    let contenuto = fs::read_to_string(percorso)?;
    Ok(contenuto)
}

fn main() {
    match leggi_contenuto_file("file.txt") {
        Ok(contenuto) => println!("Il contenuto del file è:\n{}", contenuto),
        Err(errore) => println!("Si è verificato un errore: {}", errore),
    }
}
```

Nel codice, l'espressione *fs::read_to_string(percorso)* restituisce un *Result*, che viene automaticamente propagato grazie all'operatore ?. Se l'operazione riesce, il contenuto del file viene restituito all'interno di un *Ok*. Se fallisce, il programma gestisce l'errore tramite l'*Err.*

Per la scrittura su un file, il modulo *fs* fornisce la funzione *write*, che accetta un percorso e una stringa da scrivere. Anche questa funzione restituisce un *Result*, il che significa che possiamo inserirla in una catena di espressioni come visto in precedenza:

```
use std::fs;
use std::io;

fn scrivi_su_file(percorso: &str, contenuto: &str) -> Result<(), io::Error> {
    fs::write(percorso, contenuto)?;
    Ok(())
}

fn main() {
    if let Err(errore) = scrivi_su_file("file.txt", "Ciao, mondo!") {
        println!("Errore durante la scrittura su file: {}", errore);
    } else {
        println!("Scrittura avvenuta con successo!");
```

```
    }
}
```

Anche in questo caso, *fs::write* è un'espressione che restituisce un *Result.* Se l'operazione ha successo, restituisce *Ok(()),* altrimenti *Err(io::Error).* La gestione dell'errore viene fatta in modo esplicito con il costrutto *if let*, una forma di espressione condizionale che permette di trattare il valore del *Result.*
La creazione e la gestione delle directory seguono un pattern simile. Per crearne una si può usare *fs::create_dir* o *fs::create_dir_all,* che crea anche tutte le directory intermedie necessarie. Anche qui, le funzioni restituiscono un *Result*, e si possono gestire gli errori nello stesso modo:

```
use std::fs;
use std::io;

fn crea_cartella(percorso: &str) -> Result<(), io::Error> {
    fs::create_dir(percorso)?;
    Ok(())
}

fn main() {
    match crea_cartella("nuova_cartella") {
        Ok(_) => println!("Cartella creata con successo!"),
        Err(errore) => println!("Errore durante la creazione della cartella: {}", errore),
    }
}
```

L'uso di espressioni è particolarmente utile per evitare la gestione manuale degli errori a ogni passo. Grazie alla loro propagazione con ?, possiamo scrivere codice conciso e sicuro. In molti casi, il flusso logico del programma può essere interamente basato su espressioni che restano "pure", senza bisogno di interrompere il flusso con costrutti condizionali troppo espliciti.
Rust incoraggia l'uso di espressioni non solo per il controllo del flusso ma anche per la gestione sicura delle risorse, come file e directory. In questo modo, possiamo garantire che le operazioni vengano sempre completate con successo o

falliscano gestendo esplicitamente gli errori, mantenendo la chiarezza e la leggibilità del codice.

### *La gestione del tempo*

La gestione del tempo è gestita principalmente attraverso il modulo *std::time*, che offre tipi e funzioni per misurare il tempo e lavorare con durate. Due tipi fondamentali sono *Instant*, utilizzato per misurare il tempo relativo, e *SystemTime,* che rappresenta un punto nel tempo basato sul sistema. Le operazioni legate al tempo, come calcolare intervalli o aspettare durate, si integrano facilmente nel paradigma delle espressioni di Rust, poiché spesso restituiscono un *Result* o possono essere utilizzate in catene di operazioni espressive.
Un esempio semplice di utilizzo di *Instant* è misurare quanto tempo impiega una sezione di codice a completarsi. Questo si può fare catturando un'istanza di *Instant* prima e dopo l'esecuzione, e poi sottraendo i due valori per ottenere una *Duration,* la quale rappresenta la differenza di tempo tra i due istanti:

```
use std::time::{Instant, Duration};

fn esegui_operazione_lunga() {
    // Simuliamo un'operazione lunga
    std::thread::sleep(Duration::from_secs(2));
}

fn main() {
    let inizio = Instant::now();
    esegui_operazione_lunga();
    let durata = inizio.elapsed();

    println!("L'operazione ha impiegato: {:?}", durata);
}
```

*Instant::now()* restituisce un'espressione che cattura il tempo corrente. Dopo l'esecuzione dell'operazione, *inizio.elapsed()* è un'altra espressione che calcola il tempo trascorso, restituendo una *Duration*. Questo valore può essere utilizzato per

vari scopi, come la registrazione o il controllo delle prestazioni, il tutto senza dover gestire manualmente strutture di controllo complesse.
Quando si lavora con *SystemTime*, che rappresenta un orologio relativo al sistema (data e ora corrente), possiamo eseguire operazioni simili per ottenere *timestamp* o verificare differenze rispetto a un momento specifico. Tuttavia, *SystemTime* può fallire (ad esempio, se si chiede un tempo precedente all'inizio dell'*epoch*), e questo si riflette tramite un *Result:*

```
use std::time::{SystemTime, UNIX_EPOCH};

fn main() {
    match SystemTime::now().duration_since(UNIX_EPOCH) {
        Ok(durata) => println!("Tempo passato dall'epoch: {} secondi", durata.as_secs()),
        Err(e) => println!("Errore nel calcolare il tempo: {:?}", e),
    }
}
```

In questo caso, *SystemTime::now().duration_since(UNIX_EPOCH)* restituisce un *Result<Duration, SystemTimeError>,* quindi possiamo gestire facilmente il possibile fallimento con un'espressione *match*. Se l'operazione ha successo, *Ok(durata)* ci permette di accedere alla durata passata dall'*epoch*, altrimenti possiamo catturare l'errore con *Err(e).*
Rust permette anche di manipolare durate temporali con il tipo *Duration,* che rappresenta un intervallo di tempo. Può essere utilizzato, per esempio, per definire quanto tempo attendere in un thread o per fare confronti tra intervalli di tempo. Le espressioni basate su *Duration* si integrano naturalmente con le funzioni che richiedono attese, come il metodo *sleep* dei thread, senza necessità di introdurre logiche complesse:

```
use std::time::Duration;
use std::thread::sleep;

fn main() {
    let attesa = Duration::from_secs(3);
```

```
    println!("Attendo per {} secondi...", attesa.as_secs());
    sleep(attesa);
    println!("Attesa completata!");
}
```

Nell'esempio, *Duration::from_secs(3)* è un'espressione che restituisce una durata di 3 secondi. Questa durata viene passata direttamente alla funzione *sleep*, che sospende l'esecuzione del programma per il tempo specificato. La gestione delle durate come espressioni rende il codice leggibile e modulare, in quanto le durate possono essere modificate o calcolate dinamicamente all'interno di espressioni più complesse.

La gestione del tempo con il supporto di tipi robusti come *Instant, SystemTime,* e *Duration,* si inserisce perfettamente nel contesto delle espressioni. Funzioni che restituiscono valori relativi al tempo, integrate con il sistema di *Result* e gestione degli errori, consentono un flusso di codice chiaro e sicuro. Inoltre, la possibilità di trattare il tempo come una risorsa misurabile e manipolabile tramite espressioni è ideale per applicazioni ad alte prestazioni o per la gestione di eventi temporali critici.

### *Quiz & Esercizi*

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

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

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

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

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

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

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

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

9) Scrivere un identificatore valido e uno non valido.

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

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

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.

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.

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

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.

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

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

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

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

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.

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

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

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

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

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

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

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

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.

***Riassunto***

Nel capitolo, abbiamo esplorato diversi aspetti del linguaggio, approfondendo il concetto di gestione delle eccezioni e le espressioni condizionali. A differenza di altri, Rust non ha un sistema classico di gestione delle eccezioni basato su blocchi *try-catch*, ma si affida invece al tipo *Result* per indicare il successo o il fallimento di un'operazione. Il tipo contiene due varianti: *Ok*, che rappresenta un'operazione riuscita, e *Err*, che indica un fallimento. Questo sistema obbliga i programmatori a gestire esplicitamente ogni possibile errore, migliorando la robustezza del codice. Abbiamo anche discusso delle espressioni condizionali e delle strutture di controllo, in cui *if, match* e *loop* giocano un

ruolo centrale nel flusso logico del programma, integrandosi perfettamente con la filosofia del linguaggio di favorire l'immutabilità e il controllo esplicito sui flussi.
Successivamente abbiamo trattato la gestione di file e directory, un ambito in cui si utilizza funzioni del modulo *std::fs* per leggere, scrivere e manipolare file e cartelle. Anche qui, Rust si affida al sistema di gestione degli errori attraverso *Result*, e ogni operazione potenzialmente fallibile, come l'apertura di un file, restituisce un *Result* che può essere controllato per garantire la correttezza del programma. Allo stesso tempo, abbiamo integrato queste operazioni nel contesto delle espressioni, mostrando come i risultati delle operazioni sui file possono essere gestiti e concatenati attraverso espressioni condizionali e altre strutture di controllo.
Infine, abbiamo discusso della gestione del tempo utilizzando il modulo *std::time*, con tipi come *Instant* e *SystemTime* che consentono di misurare intervalli di tempo e lavorare con *timestamp* relativi al sistema. Questi tipi, insieme a *Duration* utile a rappresentare durate temporali, offrono un'ampia gamma di funzionalità per manipolare il tempo e misurare la performance del codice. Anche qui, abbiamo visto come le operazioni temporali siano gestite tramite espressioni che restituiscono *Result*, mantenendo coerenza con l'approccio del linguaggio alla gestione degli errori e del flusso.
Attraverso tutti questi esempi, è emerso come Rust incentivi una programmazione sicura e robusta, dove il controllo esplicito delle condizioni e delle risorse è integrato direttamente nel design del linguaggio e delle sue espressioni.

# # 3 - Le funzioni

*L'ambito delle funzioni*
*Le funzioni generiche*
*Le chiusure*
*I callback*

In tutti i linguaggi di programmazione, la funzione ha il duplice vantaggio di ridurre la quantità di codice necessario all'esecuzione del programma e anche di rendere riutilizzabile una porzione di listato. Essenzialmente, in una funzione si assegna una porzione di codice con i suoi parametri a un nome che servirà a richiamarla al bisogno. Le funzioni possono comportarsi in modo simile alla matematica: calcolano un numero da uno o più numeri dati, anche se, generalmente creano un nuovo oggetto eseguendo istruzioni da determinati oggetti software.
Le funzioni rappresentano uno dei pilastri fondamentali di Rust, in quanto consentono di organizzare il codice in blocchi riutilizzabili e modulari. Qui, come in molti altri linguaggi di programmazione, permettono di incapsulare logica e comportamento, rendendo il codice più leggibile, manutenibile e testabile.
La funzione è definita tramite la parola chiave *fn,* seguita dal suo nome, dalla lista dei parametri racchiusa tra parentesi tonde, e dal tipo di ritorno, se presente. Può accettare zero o più parametri, ciascuno dei quali deve essere annotato con il tipo. Rust è un linguaggio a tipizzazione statica, il che significa che *il tipo dei parametri e del valore di ritorno deve essere specificato chiaramente*, evitando ambiguità.
Il corpo della funzione è racchiuso tra parentesi graffe. Al suo interno, è possibile scrivere il codice che implementa la logica desiderata. Il linguaggio enfatizza il

concetto di espressioni, dove la maggior parte delle costruzioni del linguaggio restituisce un valore. Questo si riflette anche nelle funzioni, le quali, se devono restituire un valore, possono farlo sia tramite la parola chiave *return*, seguita dal valore da restituire, sia semplicemente con l'ultima espressione del blocco senza il punto e virgola. Nel secondo caso, il valore dell'ultima espressione viene implicitamente considerato come il valore di ritorno della funzione.
Il linguaggio detiene un sistema di proprietà (ownership) rigoroso, il quale richiede che i programmatori siano molto attenti nella gestione della memoria e dei dati passati alle funzioni. A seconda del modo in cui un parametro viene passato a una funzione (per valore, per riferimento o per riferimento mutabile), cambiano le regole di ownership e borrowing, influenzando la sicurezza del codice e le prestazioni.
Un altro aspetto cruciale è il concetto di *lifetime*. Questo termine si riferisce al periodo di validità di un riferimento e viene usato dal compilatore per garantire che questi non diventino mai *dangling*, ovvero che non puntino mai a memoria non valida. Nelle funzioni che accettano riferimenti come parametri, è possibile e talvolta necessario annotare esplicitamente i lifetime per indicare al compilatore come i diversi riferimenti si relazionano tra loro in termini di durata.
Rust supporta anche la sovrapposizione mediante la funzione *main,* che funge da punto di ingresso per i programmi eseguibili. Sebbene sia tecnicamente una funzione come le altre, essa viene trattata in modo speciale dal compilatore, in quanto rappresenta il punto in cui l'esecuzione del programma inizia. La sua firma è fissa e non accetta parametri né restituisce valori. Tuttavia, esistono varianti che permettono di gestire parametri da linea di comando e di restituire codici di uscita, garantendo una maggiore flessibilità nell'implementazione.
Infine, il linguaggio permette la definizione di funzioni anonime, note come *closure*, che possono catturare l'ambiente in cui sono definite. Le *closure* o chiusure offrono una sintassi concisa per esprimere *funzioni inline* e vengono comunemente utilizzate nei contesti dove è necessario passare piccole funzioni

come argomenti ad altre funzioni o metodi, in pratica il compilatore suggerisce di "inserire" direttamente il loro codice nel punto in cui vengono chiamate, anziché eseguire una chiamata a funzione tradizionale. Questo avviene tramite l'annotazione *#[inline],* suggerendo al compilatore di sostituire la chiamata alla funzione con il corpo della funzione stessa, riducendo così l'overhead delle chiamate di funzione.

### *Gli argomenti della funzione*

Gli argomenti o parametri di una funzione sono le variabili che gli vengono passate al momento della sua chiamata. Questi parametri permettono di fornire dati alla funzione affinché possa eseguire operazioni su di essi. Ogni parametro deve essere definito con un nome e deve essere associato a un tipo specifico. Rust richiede che il tipo di ogni parametro sia esplicitamente annotato, poiché si tratta di un linguaggio a tipizzazione statica.

Per definire una funzione che accetta parametri, si utilizza la parola chiave *fn*, seguita dal nome della funzione, dall'elenco dei parametri racchiuso tra parentesi tonde, e infine dal corpo della funzione racchiuso tra parentesi graffe. Ad esempio, se si volesse definire una funzione che calcola la somma di due numeri interi, si potrebbe scrivere:

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

Nel codice, *a* e *b* sono i parametri della funzione *somma*, entrambi del tipo intero a 32 bit. Questa prende i due parametri, esegue l'operazione di somma e restituisce il risultato, anch'esso di tipo i32. È importante notare che i parametri vengono passati per valore di default, il che significa che una copia del dato viene passata alla funzione, piuttosto che un riferimento al dato originale.

Tuttavia, Rust permette anche di passare parametri per riferimento, il che è utile quando si vuole evitare la copia dei dati, specialmente se questi sono grandi o se si

desidera modificare il dato originale. Per farlo si utilizza il simbolo &. Ad esempio, una funzione che prende un riferimento a una stringa senza modificarla potrebbe essere scritta come:

```
fn stampa(messaggio: &str) {
    println!("{}", messaggio);
}
```

Qui, *messaggio* è un riferimento a una stringa (*&str*), e la funzione stampa semplicemente mostra la stringa senza modificarla. Se la funzione avesse necessità di modificare il parametro passato, si utilizzerebbe un riferimento mutabile con *&mut*.
Inoltre, quando si passano parametri per riferimento, entra in gioco il concetto di *borrowing*, che regola il prestito dei dati in modo sicuro per evitare problemi come i *dangling pointers*. Ad esempio, se volessimo modificare un numero passato a una funzione, si potrebbe scrivere:

```
fn incrementa(numero: &mut i32) {
    *numero += 1;
}
```

In questo caso, *numero* è un riferimento mutabile (*&mut i32*), e la funzione incrementa il valore al quale numero fa riferimento. L'operatore * viene utilizzato per dereferenziare il puntatore e accedere al valore che si trova all'indirizzo di memoria a cui numero fa riferimento.
Infine, Rust permette anche di passare funzioni come parametri ad altre funzioni, utilizzando i puntatori a funzione o le *closure*. Questo consente di implementare comportamenti flessibili e componibili, tipici della programmazione funzionale.
Essenzialmente, la gestione dei parametri è strettamente legata al sistema di tipizzazione e proprietà del linguaggio, garantendo sicurezza e prestazioni. I programmatori devono essere consapevoli delle modalità di passaggio per valore e per riferimento, oltre che delle implicazioni del borrowing.

***Definizione di una funzione***

La parola chiave *fn* ci servirà per la definizione di una funzione, seguita dal nome, da una lista di parametri racchiusi tra parentesi tonde, e dal corpo racchiuso tra parentesi graffe. Una funzione può restituire un valore, il cui tipo deve essere specificato dopo il simbolo ->. La dichiarazione di *fn* implica l'esplicitazione dei tipi dei parametri e del tipo di ritorno, *poiché il linguaggio è a tipizzazione statica e richiede che ogni tipo sia conosciuto a tempo di compilazione.*

Proviamo con una funzione semplice che non accetta parametri e non restituisce alcun valore. Potrebbe essere definita così:

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

La funzione *saluta* non prende alcun parametro e il suo corpo esegue un'operazione di stampa a schermo di un messaggio. Non avendo un tipo di ritorno esplicitamente dichiarato, si assume che la funzione ritorni implicitamente l'unità (()), che rappresenta il concetto di "nessun valore significativo".

Quando una funzione deve accettare uno o più parametri, questi vengono elencati tra le parentesi tonde, separati da virgole, e ciascuno deve essere annotato con il tipo. Ad esempio, una funzione che prende due numeri interi e restituisce la loro somma può essere definita così:

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

Qui, *a* e *b* sono i parametri della funzione *somma*, entrambi di tipo i32, e la funzione restituisce un valore anch'esso di tipo i32, che è il risultato dell'operazione di *somma*. Il tipo di ritorno i32 è dichiarato dopo la freccia ->, e il valore di ritorno è l'ultima espressione del corpo della funzione. *In Rust, se l'ultima*

*espressione del corpo di una funzione non termina con un punto e virgola, essa viene implicitamente considerata come il valore di ritorno della funzione*.
È possibile dichiarare funzioni che accettano parametri per riferimento piuttosto che per valore. Questo è utile quando si desidera evitare la copia di grandi quantità di dati o quando si vuole permettere alla funzione di modificare i dati originali. Per esempio, una funzione che calcola la lunghezza di una stringa senza copiarla potrebbe essere definita così:

```
fn lunghezza(testo: &str) -> usize {
    testo.len()
}
```

In questo caso, testo è un riferimento immutabile a una stringa (*&str*), e la funzione *lunghezza* restituisce un valore di tipo *usize*, che rappresenta la lunghezza della stringa. Si utilizza il metodo *len()* per calcolare la lunghezza e ottenere questo valore. Passare per riferimento permette di evitare la copia della stringa, migliorando l'efficienza.
Rust supporta anche funzioni che non restituiscono valori significativi, ma che possono restituire *Result* o *Option*, tipologie utilizzate per gestire errori o assenza di valore. Queste funzioni vengono spesso utilizzate in contesti dove è necessaria una gestione esplicita degli errori o delle condizioni speciali.
Per quanto riguarda la mutabilità, se una funzione deve modificare i dati a cui fa riferimento, i parametri devono essere dichiarati come riferimenti mutabili. Ad esempio, una funzione che incrementa un numero potrebbe essere definita così:

```
fn incrementa(numero: &mut i32) {
    *numero += 1;
}
```

Qui, *numero* è un riferimento mutabile a un intero (*&mut i32*), e la funzione incrementa utilizza l'operatore * per dereferenziare il puntatore e accedere al valore a cui numero fa riferimento, incrementandolo di uno.

Oltre alla mutabilità, ci sono altri aspetti importanti da considerare nella progettazione delle funzioni, in particolare l'ottimizzazione delle prestazioni. A tal proposito, l'annotazione *#[inline]* gioca un ruolo cruciale. Utilizzandola, il compilatore può decidere di inlinare una funzione, riducendo l'overhead delle chiamate e migliorando l'efficienza, specialmente per quelle brevi o frequentemente utilizzate. *#[inline]* viene utilizzata per suggerire al compilatore di ottimizzare le chiamate a funzioni tramite l'inserimento del codice della funzione direttamente nel punto in cui viene chiamata, anziché procedere convenzianalmente:

```
#[inline]
fn somma(a: i32, b: i32) -> i32 {
    a + b
}

fn main() {
    let risultato = somma(5, 10);
    println!("Risultato: {}", risultato);
}
```

In questo esempio, la funzione *somma* è annotata con *#[inline],* suggerendo al compilatore di considerare l'inlining della funzione nelle sue chiamate. In caso positivo, se è reputato vantaggioso, l'ottimizzazione eviterà l'overhead di una chiamata di funzione e sostituirà direttamente il corpo della funzione nel punto di chiamata, contribuendo a migliorare le prestazioni dell'applicazione.
L'annotazione *#[inline]* è particolarmente utile in contesti in cui si scrivono librerie o moduli di codice che richiedono elevate prestazioni, poiché può ridurre il tempo di esecuzione e migliorare l'efficienza. Tuttavia, è sempre bene utilizzarla con attenzione, poiché l'inlining eccessivo di funzioni può aumentare le dimensioni del codice binario e influenzare negativamente la cache della CPU.

### *Ambito delle variabili nelle funzioni*

Lo scope (ambito) di una funzione in Rust definisce la regione del programma in

cui le variabili e le risorse dichiarate all'interno della funzione sono visibili e accessibili. Ogni volta che si dichiara una variabile all'interno di una funzione, *quella variabile vive solo all'interno dello scope della funzione*, il che significa che non può essere utilizzata al di fuori di esso. Rust adotta un approccio rigoroso alla gestione dello scope per garantire sicurezza e prevenire errori come l'uso di variabili non inizializzate o riferimenti a dati non più validi.
Quando una funzione viene invocata, viene creato un nuovo frame di stack in cui vengono allocate tutte le variabili locali definite all'interno della funzione. Queste variabili esistono solo per la durata dell'esecuzione della funzione e vengono deallocate automaticamente quando termina, sia per il normale flusso di esecuzione che in caso di un *return* anticipato o di un'uscita forzata tramite *panic!.*
Vediamo il seguente esempio di codice:

```
fn calcola_area() {
    let lunghezza = 5;
    let larghezza = 3;
    let area = lunghezza * larghezza;
    println!("L'area è {}", area);
}
```

In questo esempio, le variabili *lunghezza, larghezza* e *area* sono tutte dichiarate all'interno della funzione *calcola_area*. Queste variabili sono visibili e accessibili solo all'interno del corpo della funzione stessa. Al di fuori di questa funzione, le variabili non esistono e qualsiasi tentativo di accedervi genererebbe un errore di compilazione. Quando la funzione termina, lo stack frame associato alla funzione viene eliminato, e con esso tutte le variabili locali.
Anche qui, attenzione alla gestione della memoria e alle regole di ownership (lo vedremo meglio tra poche righe), che ha un impatto significativo sullo scope delle variabili. Quelle che possiedono risorse allocate dinamicamente (come heap memory o file descriptor) vengono automaticamente deallocate quando escono dallo scope, grazie al concetto di RAII (*Resource Acquisition Is Initialization*). Ciò garantisce che le risorse vengano sempre rilasciate in modo sicuro, prevenendo

perdite di memoria. Consideriamo un altro esempio:

```
fn crea_stringa() -> String {
    let s = String::from("Ciao");
    s
}
```

In questo caso, la variabile *s* è una stringa allocata dinamicamente (su heap), e il suo valore viene restituito dalla funzione *crea_stringa*. Anche se *s* esce dallo scope al termine della funzione, il valore viene trasferito al chiamante grazie al sistema di ownership. Questo trasferimento impedisce che il valore venga deallocato prematuramente. Se il valore non fosse stato restituito, il linguaggio avrebbe automaticamente deallocato la memoria associata alla stringa quando *s* fosse uscita dallo scope.
Lo scope di una funzione non è limitato solo alle variabili locali, ma si estende anche ai riferimenti che la funzione può prendere in input come parametri. Quando questa accetta riferimenti come argomenti, devono essere validi per tutta la durata della funzione. Rust utilizza i *lifetime* per garantire che i riferimenti non diventino invalidi durante l'esecuzione:

```
fn stampa_primo_elemento(v: &[i32]) {
    println!("Il primo elemento è {}", v[0]);
}
```

*v* è un riferimento a un array di interi (*&[i32]*). La funzione *stampa_primo_elemento* utilizza il riferimento per accedere al primo elemento dell'array e stamparlo. Il riferimento *v* è valido solo all'interno dello scope della funzione. Il sistema di borrowing di Rust garantisce che *v* non possa essere invalidato finché è utilizzato all'interno della funzione.
Lo scope diventa ancora più importante quando si lavora con un riferimento mutabile, infatti quest'ultimo non può coesistere con altri relativi al dato a cui si riferisce, e Rust applica rigorosamente queste regole per prevenire condizioni di race e altre problematiche legate alla concorrenza:

```
fn incrementa_valore(valore: &mut i32) {
    *valore += 1;
}
```

Quindi, valore è un riferimento mutabile a un intero. Lo scope di *valore* è limitato al corpo della funzione *incrementa_valore*. All'interno di questo scope, *valore* può essere modificato, ma non può esistere un altro riferimento a valore finché esso è in uso all'interno dello scope. Al termine della funzione, il riferimento mutabile valore esce dallo scope, permettendo eventuali ulteriori utilizzi o modifiche del dato originale.

### *Funzioni, Ownership e Borrowing*

Il concetto di ownership (proprietà) è uno dei pilastri fondamentali del linguaggio Rust, ed è strettamente collegato alla gestione della memoria e alla sicurezza del sistema, dove ogni valore ha un proprietario, e questo è responsabile della deallocazione della memoria associata a quel valore. Quando un valore esce dallo scope (ambito), Rust dealloca automaticamente la memoria, evitando perdite e garantendo la sicurezza della memoria senza la necessità di un garbage collector.
Nell'ambito delle funzioni, l'ownership gioca un ruolo cruciale, poiché ogni volta che un valore viene passato a una funzione, si può verificare un trasferimento di proprietà. Vediamo un esempio per chiarire questo concetto:

```
fn consuma_stringa(s: String) {
    println!("Ho ricevuto: {}", s);
}

fn main() {
    let mia_stringa = String::from("Ciao, mondo!");
    consuma_stringa(mia_stringa);
    // `mia_stringa` non è più accessibile qui, poiché la proprietà è stata trasferita
}
```

In pratica la funzione *consuma_stringa* prende in input una *String*, il cui tipo

gestisce memoria allocata dinamicamente su heap. Quando *mia_stringa* viene passata a *consuma_stringa*, la proprietà del valore viene trasferita alla funzione. Ciò significa che, dopo la chiamata alla funzione, *mia_stringa* non è più accessibile nel contesto del *main*. Tentare di accedervi dopo il suo utilizzo nella funzione genererebbe un errore di compilazione, poiché Rust non permette l'uso di variabili il cui valore è stato trasferito.

Questo trasferimento di ownership si applica automaticamente quando si passano valori che non implementano il *trait Copy*. I tipi come *String, Vec*, e strutture che contengono tali tipi trasferiscono la proprietà quando vengono passati come argomenti a una funzione. Questo comportamento è in contrasto con i tipi che implementano il *trait Copy*, come i tipi primiti (ad esempio, i32, f64, bool), i quali vengono copiati anziché trasferiti. Ecco un esempio con un tipo che implementa *Copy*:

```
fn stampa_numero(n: i32) {
    println!("Il numero è {}", n);
}

fn main() {
    let numero = 42;
    stampa_numero(numero);
    // `numero` è ancora accessibile qui, poiché è stato copiato
}
```

In questo caso, il valore *numero* viene copiato quando viene passato alla funzione *stampa_numero*, quindi rimane accessibile anche dopo la chiamata alla funzione. La funzione riceve una copia del valore originale, non il valore stesso.

Rust fornisce anche un meccanismo per evitare il trasferimento di proprietà quando si passa un valore a una funzione, utilizzando i riferimenti. Questi permettono a una funzione di accedere al dato senza prendere possesso della sua proprietà, preservando la possibilità di utilizzare il dato anche dopo la chiamata alla funzione:

```
fn stampa_stringa(s: &String) {
    println!("Ho ricevuto: {}", s);
}

fn main() {
    let mia_stringa = String::from("Ciao, mondo!");
    stampa_stringa(&mia_stringa);
    // `mia_stringa` è ancora accessibile qui, poiché è stata passata per riferimento
}
```

*stampa_stringa* prende un riferimento a una String (*&String*) invece della stringa stessa. Passando *mia_stringa* per riferimento con *&mia_stringa*, non si trasferisce la proprietà, e *mia_stringa* rimane utilizzabile dopo la chiamata alla funzione. *Questo meccanismo si chiama borrowing*, e garantisce che i dati rimangano al sicuro anche quando vengono utilizzati in più parti del programma.
È importante comprendere che quando si passano riferimenti a una funzione, si devono considerare le regole di borrowing. Se si passa un riferimento mutabile, per esempio, Rust garantisce che non ne esistano altri relativi al dato durante l'utilizzo del riferimento mutabile, prevenendo condizioni di race e altri problemi legati alla concorrenza.
Un altro concetto rilevante è il *lifetime*, che specifica la durata per cui un riferimento è valido. *Rust cerca di inferire i lifetimes automaticamente nella maggior parte dei casi*, ma in situazioni più complesse, potrebbe essere necessario annotare i *lifetimes* esplicitamente per indicare al compilatore la durata delle variabili riferite. Ecco un esempio:

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

In questo esempio, *'a* è un *lifetime generico* che indica che i riferimenti *s1* e *s2*

devono avere un ciclo di vita uguale, e che il riferimento restituito dalla funzione è valido fintanto che quelli in input lo sono. Questo garantisce che la funzione non restituisca un riferimento a un dato che potrebbe non essere più valido.
In sintesi, l'ownership relativo alle funzioni è un concetto centrale che governa il modo in cui i dati vengono passati e gestiti tra funzioni. Attraverso il trasferimento di proprietà, il borrowing e i lifetimes, Rust garantisce la sicurezza della memoria e la prevenzione di errori comuni come i dangling pointers o le race condition. Questo sistema rigoroso consente ai programmatori di scrivere codice sicuro ed efficiente, evitando i problemi tipici della gestione manuale della memoria.
Il concetto di borrowing (prestito) è un elemento chiave nel sistema di gestione della memoria e si integra perfettamente con il meccanismo di ownership (proprietà). Borrowing permette di prestare l'accesso a un valore senza trasferirne la proprietà, consentendo di manipolare i dati in modo sicuro ed efficiente senza sacrificare la sicurezza della memoria.
Quando una funzione prende in prestito un valore, ciò avviene passando un riferimento al valore anziché quest'ultimo direttamente. Un riferimento può essere immutabile o mutabile. Il primo permette la lettura dei dati, ma non la loro modifica. Un riferimento mutabile, al contrario, consente la modifica dei dati, ma con restrizioni rigorose per evitare condizioni di race e garantire la sicurezza del programma.
Consideriamo prima il borrowing immutabile. Supponiamo di avere una funzione che deve leggere una stringa senza modificarla:

```
fn stampa_lunghezza(s: &String) {
    println!("La lunghezza della stringa è: {}", s.len());
}

fn main() {
    let mia_stringa = String::from("Ciao, mondo!");
    stampa_lunghezza(&mia_stringa);
    println!("Posso ancora usare la stringa: {}", mia_stringa);
}
```

In questo esempio, la funzione *stampa_lunghezza* prende un riferimento immutabile a una String (*&String*). Il riferimento immutabile permette alla funzione di leggere il contenuto della stringa e calcolarne la lunghezza senza modificarne il contenuto originale. Dopo la chiamata alla funzione, *mia_stringa* rimane intatta e può ancora essere utilizzata nel *main*. Poiché *stampa_lunghezza* prende solo in prestito il riferimento alla stringa, non ne acquisisce la proprietà.
Rust permette di avere più riferimenti immutabili contemporaneamente, il che significa che più parti del codice possono accedere in lettura allo stesso dato nello stesso momento, senza rischio di conflitti:

```
fn main() {
    let mia_stringa = String::from("Ciao, mondo!");

    let ref1 = &mia_stringa;
    let ref2 = &mia_stringa;

    println!("ref1: {}", ref1);
    println!("ref2: {}", ref2);
}
```

Qui, sia *ref1* che *ref2* sono riferimenti immutabili alla stessa stringa. Poiché nessuno dei due può modificarla, possono essere utilizzati in lettura senza problemi.
Il borrowing mutabile, d'altra parte, consente la modifica dei dati, ma Rust impone restrizioni rigorose: può esistere un solo riferimento mutabile a un dato alla volta, e non possono coesistere riferimenti immutabili e mutabili al medesimo dato. Questo evita che si verifichino condizioni di race, dove due o più thread o parti di codice potrebbero cercare di modificare o leggere un dato contemporaneamente in modo incoerente:

```
fn aggiungi_punto_esclamativo(s: &mut String) {
    s.push_str("!");
}
```

```
fn main() {
    let mut mia_stringa = String::from("Ciao, mondo");
    aggiungi_punto_esclamativo(&mut mia_stringa);
    println!("La stringa modificata è: {}", mia_stringa);
}
```

La funzione *aggiungi_punto_esclamativo* prende un riferimento mutabile alla stringa (*&mut String*). Questo consente alla funzione di modificare direttamente il suo contenuto, aggiungendo un punto esclamativo alla fine. Quando *mia_stringa* viene passata alla funzione, si utilizza *&mut mia_stringa*, indicando chiaramente che si sta prestando la stringa in modo mutabile. Dopo la chiamata alla funzione, la stringa è stata modificata e il cambiamento è visibile nel contesto del *main*.
Tentare di creare un secondo riferimento mutabile o un riferimento immutabile mentre ne esiste già uno mutabile porta a un errore di compilazione, come dimostrato nel seguente codice:

```
fn main() {
    let mut mia_stringa = String::from("Ciao");
    let ref1 = &mut mia_stringa;
    let ref2 = &mia_stringa; // Errore: non si può avere un riferimento immutabile mentre esiste
un riferimento mutabile
}
```

Rust previene queste situazioni a tempo di compilazione, garantendo che il codice sia sicuro e privo di condizioni di race. Se si cerca di violare queste regole, il compilatore segnala l'errore, indicando esattamente dove si verifica il problema e aiutando a mantenere il programma sicuro.
Esiste anche un altro concetto interessante legato al borrowing, ovvero il *re-borrowing*. Si riferisce alla possibilità di prestare ulteriormente un riferimento, mantenendo però le regole di Rust. Ad esempio, si può passare un riferimento mutabile a una funzione che a sua volta lo presta temporaneamente a un'altra funzione:

```
fn append_and_print(s: &mut String) {
```

```
    aggiungi_punto_esclamativo(s);
    println!("Stringa modificata: {}", s);
}

fn main() {
    let mut mia_stringa = String::from("Ciao");
    append_and_print(&mut mia_stringa);
}
```

Qui, *append_and_print* prende in prestito mutabilmente la stringa *mia_stringa*, la modifica tramite *aggiungi_punto_esclamativo*, e la stampa. Questo *re-borrowing* consente di comporre funzioni in modo sicuro ed efficiente.

Essenzialmente, il borrowing è un potente meccanismo che consente di lavorare con i dati in modo sicuro, evitando il trasferimento di proprietà quando non necessario, e mantenendo il controllo rigoroso sulla mutabilità e l'accesso concorrente. Le regole di Rust riguardo a questo ambito sono progettate per prevenire errori comuni nella gestione della memoria, come i dangling pointers e le condizioni di race, garantendo la sicurezza del codice già a tempo di compilazione. Questi concetti, sebbene inizialmente complessi, sono fondamentali per padroneggiare il linguaggio e scrivere codice robusto e sicuro.

### *Funzioni generiche e con trait bounds*

Le funzioni generiche sono una delle caratteristiche che permettono di scrivere codice flessibile e riutilizzabile. *Utilizzando i tipi generici, è possibile definire funzioni che operano su una varietà di tipi senza dover duplicare il codice per ciascun tipo specifico*. Questo approccio riduce la ridondanza e consente di costruire astrazioni potenti e sicure a livello di tipo.

Un tipo generico viene indicato con una variabile di tipo, convenzionalmente rappresentata da una singola lettera maiuscola, come *T*. Questa può rappresentare qualsiasi tipo, a meno che non venga vincolata da specifici requisiti tramite i *trait bounds*.

Consideriamo una semplice funzione generica che accetta due parametri di tipo

generico e restituisce uno di essi:

```
fn restituisci_primo<T>(a: T, b: T) -> T {
    a
}

fn main() {
    let x = restituisci_primo(10, 20);
    let y = restituisci_primo("ciao", "mondo");
    println!("x: {}, y: {}", x, y);
}
```

Nel listato, *restituisci_primo* è una funzione generica che prende due parametri dello stesso tipo generico T e restituisce il primo parametro. Il tipo T è deciso dal compilatore in base ai tipi degli argomenti passati alla funzione. Nel *main*, *restituisci_primo* viene chiamata una volta con valori interi e una volta con stringhe. Grazie alla generalizzazione, la funzione lavora correttamente in entrambi i casi.

Tuttavia, ci sono situazioni in cui un tipo generico deve soddisfare certi requisiti per poter essere utilizzato in una funzione. Questo è il caso in cui entrano in gioco i trait *bounds*. Un trait, come vedremo nel capitolo dedicato, definisce un insieme di metodi che un tipo deve implementare per poter essere considerato conforme a quel trait. Ad esempio, il trait *std::cmp::PartialOrd* permette di confrontare due valori per determinare l'ordine relativo.

Supponiamo di voler scrivere una funzione generica che restituisce il maggiore tra due valori. In questo caso, il tipo generico deve implementare il trait *PartialOrd*, poiché la funzione dovrà confrontare i due valori:

```
fn massimo<T: PartialOrd>(a: T, b: T) -> T {
    if a > b {
        a
    } else {
        b
    }
}
```

```
fn main() {
    let max_intero = massimo(5, 10);
    let max_stringa = massimo("gatto", "cane");
    println!("Il massimo tra 5 e 10 è: {}", max_intero);
    println!("Il massimo tra 'gatto' e 'cane' è: {}", max_stringa);
}
```

Di conseguenza, *massimo* è una funzione generica che accetta due parametri dello stesso tipo T e restituisce quello con il valore maggiore. E, il trait bound *T: PartialOrd* indica che il tipo T deve implementare *PartialOrd*, che definisce l'operatore > per il confronto. Senza questo vincolo, il codice avrebbe dei problemi, *poiché il compilatore non avrebbe la garanzia che T supporti l'operatore di confronto*.
È possibile combinare più trait *bounds* su un singolo tipo generico utilizzando il segno +. Supponiamo di voler estendere la funzione *massimo* affinché, oltre a confrontare i valori, essi siano anche visualizzabili tramite il trait *std::fmt::Display*:

```
fn massimo_e_stampa<T: PartialOrd + std::fmt::Display>(a: T, b: T) -> T {
    let risultato = if a > b { a } else { b };
    println!("Il maggiore è: {}", risultato);
    risultato
}

fn main() {
    let max_intero = massimo_e_stampa(5, 10);
    let max_stringa = massimo_e_stampa("gatto", "cane");
}
```

In questo caso, *massimo_e_stampa* richiede che T implementi sia *PartialOrd* sia *Display*, in modo che la funzione possa confrontare i valori e stampare il risultato. Il compilatore garantisce che solo i tipi che soddisfano entrambi i requisiti possano essere utilizzati con questa funzione.
Di seguito una tebella con i principali trait bound e il loro uso:

| Trait bound | Utilizzo | Esempio |
|---|---|---|
| `PartialOrd` | *Consente il confronto parziale tra due valori per determinare l'ordine relativo.* | `fn max<T: PartialOrd>(a: T, b: T) -> T { if a > b { a } else { b } }` |
| `Ord` | *Permette il confronto totale tra i valori e garantisce un ordine completo.* | `fn ord_example<T: Ord>(a: T, b: T) -> Ordering { a.cmp(&b) }` |
| `Display` | *Consente la formattazione umana leggibile tramite il macro println! e altre funzioni di output.* | `fn print_value<T: Display>(val: T) { println!("{}", val); }` |
| `Debug` | *Permette la formattazione di output per il debugging, spesso usato con {:?} nei macros println!.* | `fn debug_value<T: Debug>(val: T) { println!("{:?}", val); }` |
| `Clone` | *Consente la creazione di una copia profonda di un valore.* | `fn duplicate<T: Clone>(x: T) -> T { x.clone() }` |
| `Copy` | *Permette la duplicazione di valori semplici senza chiamare clone() (valori copiabili).* | `fn duplicate<T: Copy>(x: T) -> T { x }` |
| `Eq` | *Consente il confronto di uguaglianza tra i valori.* | `fn are_equal<T: Eq>(a: T, b: T) -> bool { a == b }` |
| `Hash` | *Permette di calcolare l'hash di un valore, utile per collezioni come HashMap.* | `fn hash_value<T: Hash>(val: T) -> u64 { /* hash calcolo */ }` |

Un concetto avanzato è l'uso dei trait bounds per funzioni che lavorano con tipi generici complessi o che devono garantire che questi ultimi soddisfino una serie di condizioni. Ad esempio, in contesti più avanzati, si possono utilizzare per definire operazioni matematiche generiche o per lavorare con collezioni di dati in modo polimorfico.

Rust permette anche di scrivere funzioni che utilizzano generici con lifetimes. Un modello comune è una funzione che lavora con riferimenti e che garantisce che i riferimenti restituiti siano validi quanto quelli in input. Un esempio semplice potrebbe essere:

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

```
    }
}

fn main() {
    let stringa1 = "gatto";
    let stringa2 = "cane";
    let risultato = piu_lungo(stringa1, stringa2);
    println!("La stringa più lunga è: {}", risultato);
}
```

Come visto, *piu_lungo* è una funzione generica che utilizza il lifetime *'a*, il quale indica che i riferimenti passati alla funzione e quello restituito devono vivere almeno quanto *'a*, assicurando che il riferimento restituito sia valido fintanto che lo sono quelli di input.
In sintesi, le funzioni generiche e le funzioni con trait bounds offrono una potente astrazione che permette di scrivere codice sicuro, riutilizzabile e altamente performante. Il sistema di tipi, insieme ai trait bounds, garantisce che il codice generico sia sicuro e corretto, evitando molte delle insidie comuni in altri linguaggi di programmazione.

### *Funzioni anonime e chiusure*

Le chiusure (*closures*) e le funzioni anonime rappresentano un concetto potente e flessibile per la gestione del comportamento dinamico e del passaggio di funzioni come argomenti o risultati di altre funzioni. Abbiamo già anticipato l'argomento parlando delle espressioni. Le chiusure sono simili alle funzioni, ma con alcune caratteristiche distintive che le rendono particolarmente utili in contesti specifici, come la programmazione funzionale, la manipolazione di collezioni e l'uso di funzioni di ordine superiore (che vedremo nel prossimo paragrafo).
Una chiusura è una funzione anonima che *può catturare variabili dall'ambiente circostante in cui è definita*. Questo significa che, a differenza delle funzioni regolari, le chiusure possono accedere alle variabili che sono state dichiarate nel loro contesto esterno senza la necessità di passare esplicitamente tali variabili

come argomenti.

Per dichiarare una chiusura si utilizzano i simboli | per delimitarne i suoi parametri, seguiti da un'espressione o un blocco di codice che ne rappresenta il corpo. Un'altra caratteristica importante è che possono essere inferite dal compilatore, il che significa che nella maggior parte dei casi non è necessario specificare esplicitamente i tipi dei parametri o del valore di ritorno.

NB. spieghiamo meglio, come detto le funzioni anonime possono essere inferite dal compilatore, nel senso che il tipo della funzione anonima viene determinato automaticamente dal compilatore basandosi sul contesto in cui viene utilizzata. L'inferenza del tipo significa che il compilatore è in grado di dedurre i tipi degli argomenti e del ritorno della chiusura senza richiedere una dichiarazione esplicita da parte del programmatore. Ad esempio, se una chiusura è passata a una funzione che si aspetta un tipo specifico, Rust inferirà automaticamente il tipo della chiusura in base all'uso previsto. Questo semplifica la scrittura del codice e riduce il bisogno di annotazioni esplicite, migliorando la leggibilità e la concisione del codice.

Consideriamo un esempio semplice di una chiusura che aggiunge due numeri:

```
fn main() {
    let somma = |x, y| x + y;
    let risultato = somma(2, 3);
    println!("Il risultato della somma è: {}", risultato);
}
```

La chiusura *somma* è definita all'interno della funzione *main*. Prende due parametri x e y, li somma e restituisce il risultato. I tipi dei parametri e del valore di ritorno sono inferiti automaticamente dal compilatore, poiché vengono utilizzati con numeri interi. La chiusura viene poi chiamata con *somma(2, 3),* e il risultato viene stampato.

Parliamo adesso della caratteristica delle chiusure di catturare variabili dal contesto circostante. Questo avviene per valore, per riferimento immutabile o per riferimento mutabile, a seconda di come viene utilizzata la variabile nella chiusura:

```
fn main() {
    let moltiplicatore = 2;
    let moltiplica = |x| x * moltiplicatore;
```

```
    let risultato = moltiplica(5);
    println!("Il risultato della moltiplicazione è: {}", risultato);
}
```

In questo esempio, la chiusura *moltiplica* cattura la variabile *moltiplicatore* dal contesto esterno e la utilizza per moltiplicare il valore di *x*. Anche se *moltiplicatore* non è passato esplicitamente come argomento alla chiusura, essa può comunque accedervi. La cattura di *moltiplicatore* avviene per riferimento immutabile, poiché la chiusura non modifica il suo valore.
Invece, se una chiusura modifica le variabili catturate, è necessario catturare tali variabili per riferimento mutabile. Consideriamo un esempio in cui una chiusura modifica una variabile esterna:

```
fn main() {
    let mut conteggio = 0;
    let mut incrementa = || {
        conteggio += 1;
        println!("Conteggio incrementato: {}", conteggio);
    };
    incrementa();
    incrementa();
}
```

La chiusura *incrementa* cattura *conteggio* per riferimento mutabile e ne incrementa il valore ogni volta che viene chiamata. Poiché *conteggio* viene modificato all'interno della chiusura, Rust richiede che la chiusura sia dichiarata come mutabile (*let mut incrementa*). In questo modo, è possibile eseguire l'incremento più volte, mantenendo lo stato tra una chiamata e l'altra.
Un altro aspetto importante è la tipizzazione. Le chiusure hanno un tipo specifico, che viene generato automaticamente dal compilatore e che è unico per ognuna di esse. Tuttavia, è possibile utilizzare il trait *Fn, FnMut* o *FnOnce* per riferirsi ai tipi di chiusura in modo generico. Questi trait rappresentano rispettivamente chiusure che non catturano lo stato (o lo catturano per riferimento immutabile), chiusure che catturano lo stato per riferimento mutabile e chiusure che consumano lo stato

catturato (ad esempio, spostandolo).
Proviamo una funzione che accetta una chiusura come argomento può essere dichiarata in questo modo:

```
fn esegui_chiusura<F>(f: F)
where
    F: FnOnce(),
{
    f();
}

fn main() {
    let saluto = String::from("Ciao");
    let chiusura = || println!("{}", saluto);
    esegui_chiusura(chiusura);
}
```

La funzione *esegui_chiusura* accetta un argomento *f* di tipo generico F, che deve implementare il trait *FnOnce*. Questo significa che la chiusura passata può essere eseguita una sola volta, poiché potrebbe consumare (muovere) il valore catturato. All'interno del *main*, viene definita una chiusura che stampa una stringa e viene passata a *esegui_chiusura* per essere eseguita.
Le chiusure sono particolarmente utili quando si lavora con funzioni di ordine superiore, cioè funzioni che prendono altre funzioni come argomenti o restituiscono funzioni come risultati. Rust utilizza frequentemente le chiusure nelle sue librerie standard, ad esempio nei metodi di collezione come *map, filter,* e *fold,* che applicano una funzione a ogni elemento di una collezione.
Infine, è importante notare che le chiusure possono essere utilizzate anche per creare funzioni che hanno accesso a variabili esterne anche dopo che la funzione in cui sono state create è terminata. Questo è possibile grazie al fatto che la chiusura cattura e mantiene in vita tali variabili per tutta la durata della chiusura stessa.
In sintesi, le chiusure e le funzioni anonime offrono un modo flessibile e potente per gestire il comportamento dinamico e per lavorare con funzioni di ordine superiore. La capacità delle chiusure di catturare il contesto circostante, combinata

con la loro tipizzazione forte, garantisce che il codice sia non solo flessibile, ma anche sicuro ed efficiente. Questi strumenti sono essenziali per sfruttare appieno le potenzialità di Rust in ambiti come la programmazione funzionale, la manipolazione di collezioni e la gestione del comportamento dinamico.

| Trait | Utilizzo | Esempio |
|---|---|---|
| `Fn` | *Utilizzato per chiusure che catturano variabili dal contesto in modo immutabile.* | `fn call_fn<F: Fn()>(f: F) { f(); }` |
| `FnMut` | *Utilizzato per chiusure che modificano le variabili catturate dal contesto, richiede mutabilità.* | `fn call_fn_mut<F: FnMut()>(mut f: F) { f(); }` |
| `FnOnce` | *Utilizzato per chiusure che catturano e consumano le variabili dal contesto, possono essere chiamate una sola volta.* | `fn call_fn_once<F: FnOnce()>(f: F) { f(); }` |
| `Copy` | *Permette alla chiusura di essere copiata, se non cattura risorse che non implementano Copy.* | `fn use_copy<F: Fn() + Copy>(f: F) { let f_copy = f; f_copy(); }` |
| `Clone` | *Permette alla chiusura di essere clonata, utile quando si ha bisogno di riutilizzare una chiusura.* | `fn clone_closure<F: Fn() + Clone>(f: F) -> F { f.clone() }` |
| `Send` | *Indica che la chiusura può essere trasferita tra thread in modo sicuro.* | `fn thread_safe<F: Fn() + Send>(f: F) { std::thread::spawn(f); }` |
| `Sync` | *Indica che la chiusura può essere condivisa tra più thread contemporaneamente.* | `fn share_across_threads<F: Fn() + Sync>(f: &F) { /* uso */ }` |

### *Funzioni di ordine superiore*

Le funzioni di ordine superiore ne accettano altre come argomenti o ne restituiscono una come risultato. Queste funzioni sono fondamentali per implementare concetti della programmazione funzionale, come la composizione, la creazione di astrazioni più generali e il passaggio di comportamenti come parametri.

Una funzione di ordine superiore che ne accetta un'altra come argomento permette di astrarre comportamenti comuni. Per esempio, supponiamo di voler creare una funzione che esegue un'operazione specifica su due numeri interi. Invece di

scriverne una separata per ogni tipo di operazione (somma, sottrazione, moltiplicazione, ecc.), è possibile scrivere una funzione generica che ne accetta una la quale definisce l'operazione da eseguire.
Di seguito proveremo un esempio in cui definiamo una funzione *applica_operazione* che prende due interi e una funzione che specifica l'operazione da eseguire:

```
fn applica_operazione<F>(a: i32, b: i32, operazione: F) -> i32
where
    F: Fn(i32, i32) -> i32,
{
    operazione(a, b)
}

fn main() {
    let somma = |x, y| x + y;
    let risultato = applica_operazione(5, 3, somma);
    println!("Il risultato della somma è: {}", risultato);

    let moltiplica = |x, y| x * y;
    let risultato = applica_operazione(5, 3, moltiplica);
    println!("Il risultato della moltiplicazione è: {}", risultato);
}
```

Nel codice, *applica_operazione* è una funzione di ordine superiore che accetta due interi e una chiusura *operazione* come parametri, specificata tramite il trait *Fn(i32, i32) -> i32,* il che significa che accetta due i32 come argomenti e restituisce un i32. Nel *main,* definiamo due chiusure, *somma* e *moltiplica*, e le passiamo a *applica_operazione* per eseguire le operazioni corrispondenti.
Le funzioni di ordine superiore possono anche restituirne altre. Questo concetto è utile per creare funzioni che ne generano altre in base a certi parametri o configurazioni. Proviamo un esempio in cui definiamo una funzione *crea_incrementatore* che restituisce una chiusura che incrementa un valore di un certo numero:

```
fn crea_incrementatore(incremento: i32) -> impl Fn(i32) -> i32 {
    move |x| x + incremento
}

fn main() {
    let incrementa_di_cinque = crea_incrementatore(5);
    let risultato = incrementa_di_cinque(10);
    println!("10 incrementato di 5 è: {}", risultato);

    let incrementa_di_dieci = crea_incrementatore(10);
    let risultato = incrementa_di_dieci(10);
    println!("10 incrementato di 10 è: {}", risultato);
}
```

In questo esempio, *crea_incrementatore* è una funzione di ordine superiore che accetta un i32 come parametro e restituisce una chiusura che incrementa un valore. La chiusura cattura il valore di incremento dal contesto esterno e lo utilizza per incrementare qualsiasi valore le venga passato. Nel *main,* chiamiamo *crea_incrementatore* con diversi valori di incremento per generare chiusure che incrementano rispettivamente di 5 e di 10.
Le funzioni di ordine superiore sono utilizzate anche in molte delle API della libreria standard, soprattutto nella manipolazione delle collezioni. Per esempio, i metodi *map, filter,* e *fold* delle iterazioni sono tutti esempi di funzioni di ordine superiore. Questi metodi accettano chiusure come argomenti e applicano tali chiusure agli elementi della collezione per produrre un nuovo risultato.
Vediamo subito un esempio che utilizza *map* per trasformare una lista di numeri:

```
fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    let quadrati: Vec<i32> = numeri.iter().map(|x| x * x).collect();
    println!("I quadrati dei numeri sono: {:?}", quadrati);
}
```

Quindi, *map* è un metodo di ordine superiore che accetta una chiusura come argomento. La chiusura prende ciascun elemento della collezione, lo eleva al quadrato e restituisce il risultato. Il metodo *collect* viene poi utilizzato per

raccogliere i risultati in un nuovo *Vec*.
Queste caratteristiche viste negli esempi si dimostrano particolarmente utili nella scrittura di librerie e nella gestione di algoritmi complessi, dove la flessibilità e la chiarezza del codice sono essenziali.

### *Lifetimes*

Ne abbiamo già discusso in precedenza, parliamo di una componente chiave del sistema di tipi, utilizzata per garantire che i riferimenti siano sempre validi. Il concetto di *lifetime* rappresenta la durata di validità di un riferimento, e il compilatore di Rust utilizza questa informazione per prevenire errori comuni come i *dangling references* (riferimenti pendenti) e i *data races* (condizioni di gara sui dati). Nelle funzioni diventano particolarmente importanti quando si lavora con riferimenti come argomenti o valori di ritorno.
Il *lifetime elision* è una caratteristica di Rust che semplifica l'uso dei lifetimes nelle funzioni. In molti casi, il compilatore può dedurre automaticamente i lifetimes corretti senza che sia necessario specificarli esplicitamente. Tuttavia, ci sono situazioni più complesse in cui è richiesto di annotare esplicitamente i lifetimes per chiarire al compilatore come i riferimenti sono correlati tra loro.
Consideriamo un esempio semplice in cui non è necessario specificare esplicitamente i lifetimes grazie all'*elision*:

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

fn main() {
    let risultato = restituisci_primo("gatto", "elefante");
    println!("La stringa più lunga è: {}", risultato);
}
```

*restituisci_primo* accetta due riferimenti a stringhe (*&str*) e restituisce uno di essi. Non è necessario specificare esplicitamente i lifetimes perché il compilatore è in grado di dedurli automaticamente grazie alle regole di *lifetime elision*. In particolare, Rust applica tre regole quando deduce i lifetimes:

1) Ogni riferimento in ingresso riceve il proprio lifetime.
2) Se c'è un solo riferimento in ingresso, il lifetime del valore di ritorno è uguale.
3) Se ci sono più riferimenti in ingresso e uno di essi è *&self* o *&mut self,* il lifetime del valore di ritorno corrisponde a *self*.

Nel caso di *restituisci_primo*, entrambe le stringhe di input hanno lo stesso lifetime dedotto, *quindi il valore di ritorno condivide il lifetime di entrambe*.
Tuttavia, ci sono situazioni in cui i lifetimes devono essere annotati esplicitamente, soprattutto quando il compilatore non può dedurli in modo univoco. Supponiamo di avere una funzione che restituisce un riferimento a una stringa tra due riferimenti con lifetimes potenzialmente diversi:

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

fn main() {
    let stringa1 = "gatto";
    let stringa2 = "elefante";
    let risultato = restituisci_primo(stringa1, stringa2);
    println!("La stringa più lunga è: {}", risultato);
}
```

In questa situazione, *restituisci_primo* utilizza l'annotazione esplicita del lifetime *'a* per indicare che i riferimenti *a, b* e il valore di ritorno devono avere tutti lo stesso lifetime. Questo significa che la funzione può restituire un riferimento che sarà

valido fintanto che entrambi i riferimenti di input sono validi. L'annotazione *'a* specifica un lifetime generico che viene associato ai riferimenti quando la funzione viene chiamata.
Quando i lifetimes degli argomenti non corrispondono, il compilatore emetterà un errore. Ad esempio, se una funzione prova a restituire un riferimento con un lifetime più lungo rispetto a quello degli argomenti:

```
fn restituisci_conflitto<'a, 'b>(a: &'a str, b: &'b str) -> &'a str {
    a // Potrebbe causare un errore se 'a e 'b non corrispondono
}

fn main() {
    let stringa1 = "gatto";
    {
        let stringa2 = String::from("elefante");
        let risultato = restituisci_conflitto(stringa1, &stringa2);
        println!("Risultato: {}", risultato);
    }
}
```

Nel caso, *restituisci_conflitto* accetta due riferimenti con lifetimes distinti *'a* e *'b,* e restituisce un riferimento con il lifetime *'a*. Se i lifetimes di *a* e *b* non coincidono, il compilatore potrebbe non essere in grado di garantire che il riferimento restituito sia sempre valido. Se *stringa2* esce dallo scope prima che il risultato venga utilizzato, ciò potrebbe portare a un riferimento non valido.
Un caso più complesso in cui è necessario specificare i lifetimes espliciti si verifica quando si gestiscono funzioni che lavorano con riferimenti multipli che devono essere validi per durate differenti. In questi casi, è possibile utilizzare diverse annotazioni di lifetimes per indicare come i riferimenti sono correlati e per garantire che la funzione possa essere utilizzata in modo sicuro:

```
fn combina_stringhe<'a, 'b>(a: &'a str, b: &'b str) -> String {
    format!("{}{}", a, b)
}
```

```
fn main() {
    let stringa1 = "gatto";
    let stringa2 = String::from("elefante");
    let combinata = combina_stringhe(stringa1, &stringa2);
    println!("La stringa combinata è: {}", combinata);
}
```

Perciò, la funzione *combina_stringhe* accetta due riferimenti con lifetimes distinti *'a* e *'b*, ma, alla fine restituisce una nuova *String*. Poiché il valore ottenuto non è un riferimento ma una nuova allocazione, non è necessario preoccuparsi dei lifetimes associati ai riferimenti in ingresso, e i lifetimes espliciti sono usati solo per garantire che in generale, non solo questi ultimi, rimangano validi durante l'elaborazione.
In conclusione, i lifetimes rappresentano un meccanismo potente e cruciale per garantire la sicurezza dei riferimenti. Il *lifetime elision* permette di omettere la specificazione esplicita dei lifetimes in molti casi comuni, semplificando il codice, mentre le annotazioni esplicite sono necessarie per situazioni più complesse in cui il compilatore non può inferire correttamente i lifetimes.

### *I callback*

Parleremo di una tecnica di programmazione che consente di passare una funzione come argomento a un'altra. Questo meccanismo è utile per eseguire una funzione specifica in risposta a determinati eventi o condizioni. I callback sono anche ampiamente utilizzati per implementare comportamenti personalizzabili e flessibili, e possono essere definiti utilizzando le chiusure o i puntatori a funzioni.
Le chiusure sono particolarmente adatte per l'implementazione di callback, in quanto possono catturare variabili dal contesto circostante e sono più flessibili rispetto ai puntatori a funzioni tradizionali. Le chiusure possono essere passate a funzioni che accettano come parametri dei trait specifici, a seconda del comportamento richiesto.
Un esempio di utilizzo di un callback può essere una funzione che esegue

un'operazione sui dati e accetta un'altra funzione come parametro per gestire i risultati:

```
fn esegui_callback<F>(dati: i32, callback: F)
where
    F: Fn(i32),
{
    // Eseguiamo qualche operazione sui dati
    let risultato = dati * 2;

    // Richiamiamo la funzione callback passandole il risultato
    callback(risultato);
}

fn main() {
    let valore = 10;

    // Definiamo una chiusura come callback che stampa il risultato
    esegui_callback(valore, |x| {
        println!("Risultato del callback: {}", x);
    });
}
```

Nel listato, la funzione *esegui_callback* accetta un intero dati e un callback generico *callback* che implementa il trait *Fn(i32).* All'interno della funzione, viene eseguita una semplice operazione sui dati, raddoppiandoli, e quindi il callback viene invocato con il risultato.
La funzione *main* mostra come definire un callback utilizzando una chiusura. Quest'ultima è passata alla funzione *esegui_callback*, la quale la esegue con il risultato calcolato. In questo caso, la chiusura cattura il valore del parametro passato e lo stampa.
Rust gestisce i callback in modo efficiente, garantendo che eventuali catture di variabili da parte delle chiusure siano sicure, rispettando le regole di ownership e borrowing. A seconda del tipo di chiusura (immutabile, mutabile o che consuma il contesto), come detto è possibile utilizzare i trait *Fn, FnMut* o *FnOnce*.
Se invece si preferiscono i puntatori a funzione, è possibile scrivere un callback

senza chiusure. Questo approccio è meno flessibile, ma utile per casi in cui non è necessario catturare variabili dal contesto:

```
fn esegui_callback(dati: i32, callback: fn(i32)) {
    let risultato = dati * 2;
    callback(risultato);
}

fn stampa_risultato(x: i32) {
    println!("Risultato del callback: {}", x);
}

fn main() {
    let valore = 10;
    esegui_callback(valore, stampa_risultato);
}
```

In questo caso, *stampa_risultato* è una funzione che viene passata come callback senza catturare variabili esterne. Il callback viene eseguito nello stesso modo, ma non beneficia della flessibilità delle chiusure, che permettono di accedere e manipolare dati dal contesto locale. I puntatori a funzione non possono catturare il contesto come le chiusure, quindi risultano limitati in termini di espressività.
Approfondendo l'argomento, è importante capire come la gestione dei callback si integri con i concetti fondamentali di Rust come il sistema di ownership e il controllo dei lifetime.
Le chiusure sono un meccanismo centrale per implementare callback, poiché possono catturare variabili dal contesto circostante in tre modalità diverse: per valore, per riferimento immutabile o per riferimento mutabile. Queste tre modalità, come anticipato precedentemente, sono rappresentate rispettivamente dai trait *FnOnce, Fn,* e *FnMut*. La scelta di quale utilizzare dipende dal tipo di cattura effettuata dalla chiusura e da come questa cattura viene utilizzata all'interno del callback.
Nel caso di una cattura per valore, la chiusura consuma la variabile catturata e quindi non può essere chiamata più di una volta, rendendo necessario l'utilizzo del

trait *FnOnce.* Questo tipo di callback è utile quando la chiusura ha bisogno di prendere possesso delle risorse, ad esempio quando si desidera trasferire la proprietà di un valore a un'altra parte del codice tramite il callback.
La cattura per riferimento immutabile, invece, permette alla chiusura di accedere alle variabili senza modificarle, utilizzando il trait *Fn.* In questo caso, il callback può essere invocato ripetutamente senza rischio di mutare lo stato del programma, poiché le variabili catturate rimangono immutate. Questo è un uso comune quando il callback ha solo bisogno di leggere o analizzare i dati del contesto, ma non deve apportare modifiche dirette.
La cattura per riferimento mutabile consente alla chiusura di modificare le variabili catturate, ed è legata al trait *FnMut.* Questa modalità è particolarmente utile quando il callback ha bisogno di aggiornare o manipolare lo stato del programma esterno in modo diretto, ma richiede che la chiusura venga invocata in un contesto che garantisce l'accesso mutabile alle variabili catturate.
Un'altra considerazione importante riguarda l'uso dei callback in ambienti concorrenti. Poiché Rust ha un forte modello di concorrenza basato sui trait *Send* e *Sync*, le chiusure possono essere utilizzate in modo sicuro anche tra diversi thread. Se una chiusura implementa il trait *Send*, significa che può essere trasferita tra thread senza problemi, mentre se implementa il trait *Sync,* significa che può essere condivisa tra thread contemporaneamente. Questo è cruciale per garantire che i callback utilizzati in contesti concorrenti non causino corruzione dei dati o condizioni di gara.
Un esempio più avanzato dell'uso dei callback si può vedere nelle librerie asincrone. In questo contesto, vengono spesso passati come chiusure ad operazioni che vengono eseguite in modo non bloccante. Le funzioni asincrone utilizzano il runtime per eseguire attività in background, e i callback sono invocati una volta che le operazioni asincrone sono completate. Questo modello, spesso visto in combinazione con il concetto di *future*, permette di scrivere codice reattivo e performante, dove le operazioni pesanti non bloccano il flusso principale

dell'applicazione.
Infine, è utile notare che, sebbene Rust offra una gestione molto sicura e rigida dei callback attraverso chiusure e puntatori a funzione, il linguaggio rimane flessibile. In alcuni casi, potrebbe essere utile combinarli con altre strutture di dati o tecniche come i pattern di functional programming, portando a un'ulteriore astrazione e flessibilità nel modo in cui i callback vengono utilizzati nel codice.

### *Funzioni variadiche (argomento avanzato)*

In Rust, le funzioni variadiche, ovvero quelle che accettano un numero variabile di argomenti, sono limitate e non sono supportate nativamente come in altri linguaggi come C o Python. Tuttavia, il linguaggio permette di interagire con funzioni variadiche in linguaggi esterni come il C attraverso l'uso del sistema FFI (*Foreign Function Interface*), ma non fornisce un supporto diretto per dichiararle e utilizzarle all'interno del proprio codice.
Quando si lavora con FFI, Rust permette di dichiarare funzioni esterne che accettano un numero variabile di argomenti utilizzando la parola chiave *extern* insieme a una specifica ABI (*Application Binary Interface*), come C. Questo è utile quando si deve interagire con librerie C che espongono funzioni variadiche, come *printf*. Osserviamo subito come applicare tutto ciò con un esempio in cui si utilizza una funzione C variadica come *printf* da Rust:

```
extern "C" {
    fn printf(formato: *const i8, ...) -> i32;
}

fn main() {
    let formato = b"Numero: %d\n\0".as_ptr() as *const i8;
    unsafe {
        printf(formato, 42);
    }
}
```

La funzione *printf* è dichiarata con l'ABI *C* e accetta un numero variabile di

argomenti. La parola chiave *extern* indica che la funzione è definita in un altro linguaggio (C, in questo caso). Il primo parametro, formato, è un puntatore a una stringa di caratteri in stile C (**const i8*), e i successivi argomenti sono variadici. Nel *main,* la stringa di formato è definita come un array di byte terminato da un carattere nullo (\0), e l'uso di *unsafe* è necessario per chiamare la funzione, poiché interagire con codice esterno può essere pericoloso.
Tuttavia, questa interazione con funzioni variadiche è limitata a scenari FFI, e Rust stesso non supporta la dichiarazione di funzioni variadiche native. Questo è una scelta progettuale per mantenere la sicurezza e la prevedibilità del linguaggio. Le funzioni variadiche presentano sfide complesse per il sistema di tipi e la gestione della memoria, come la verifica del tipo e la gestione della lunghezza variabile degli argomenti, che possono introdurre vulnerabilità e incertezze.
Nonostante ciò, Rust offre altre modalità per ottenere comportamenti simili, sfruttando le funzioni che accettano collezioni come *array, slice* o *tuple* come parametri. In questo modo si può passare un numero variabile di argomenti alla funzione, anche se sotto forma di una struttura dati esplicita anziché come argomenti individuali variabili.
Ecco un esempio in cui si utilizza una *slice* per simulare una funzione variadica:

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

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

Come detto, la funzione *somma* accetta una *slice* di i32 come argomento. Questo approccio permette di passare un numero arbitrario di interi alla funzione, ottenendo così un comportamento simile a quello di una variadica, ma con i benefici della sicurezza e della prevedibilità di Rust. La funzione utilizza il metodo

*iter().sum()* per iterare sulla *slice* e sommare tutti gli elementi, restituendo il risultato.

Per concludere, mentre Rust non supporta direttamente le funzioni variadiche come in altri linguaggi, offre alternative sicure e potenti attraverso l'uso di collezioni e interfacce con linguaggi esterni. Questa scelta è in linea con la filosofia del linguaggio, che privilegia la sicurezza e la gestione rigorosa della memoria rispetto alla flessibilità delle API variadiche, che possono introdurre rischi di sicurezza e complessità indesiderate.

### *Iteratori e funzioni di utilità*

In Rust, l'uso di idiomi come gli iteratori e le funzioni di utilità è fondamentale per scrivere codice che sia non solo efficiente e sicuro, ma anche espressivo e idiomatico.

NB. Scrivere codice idiomatico *significa seguire le convenzioni e le pratiche consigliate dal linguaggio per ottenere il massimo dalla sua sintassi e semantica*. Nell'ambito delle funzioni, questo include utilizzare caratteristiche come le chiusure per operazioni concise, sfruttare i trait per la generazione di funzioni generiche e utilizzare i metodi degli iteratori per manipolare collezioni in modo funzionale e efficiente. Inoltre, implica la corretta gestione di ownership e borrowing per garantire la sicurezza della memoria e l'assenza di condizioni di gara. Adottare questi idiomi aiuta a scrivere codice che è non solo efficace ma anche leggibile, manutenibile e conforme alle aspettative della comunità.

Gli iteratori, in particolare, sono un pilastro della programmazione idiomatica in Rust, poiché permettono di lavorare con sequenze di dati in modo fluido, conciso e con prestazioni ottimizzate. Le funzioni di utilità, spesso integrate nelle collezioni e negli iteratori stessi, facilitano l'elaborazione di dati complessi con operazioni funzionali come la mappatura, il filtraggio e la riduzione.

Un iteratore è un oggetto che implementa il trait Iterator, il quale richiede l'implementazione del metodo *next().* Questo metodo restituisce un elemento della sequenza a ogni chiamata, incapsulato in un'opzione (*Option<T>*), restituendo *Some(T)* finché ci sono elementi, e *None* quando l'iteratore è esaurito. Grazie a questo meccanismo, è possibile gestire in modo efficiente il consumo di sequenze

di dati senza la necessità di mantenere tutto in memoria.
Per esempio, se volessimo iterare su una sequenza di numeri, potremmo utilizzare un ciclo *for*:

```
fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    for numero in numeri.iter() {
        println!("Numero: {}", numero);
    }
}
```

Il metodo *iter()* crea un iteratore sulla *Vec<i32>,* che viene consumato dal ciclo *for*. Questo approccio è semplice e idiomatico, sfruttando il pattern comune dell'iterazione sulle collezioni.
Le funzioni di utilità sugli iteratori offrono potenti astrazioni funzionali per trasformare e combinare sequenze di dati. Tra queste, le più comuni includono *map, filter, collect, fold* e *zip*. Queste permettono di eseguire operazioni complesse su dati in modo conciso e senza mutare lo stato della collezione originale.
Ad esempio, se volessimo trasformare una lista di numeri elevandoli al quadrato e raccogliere il risultato in un nuovo vettore, potremmo farlo in modo idiomatico con *map* e *collect*:

```
fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    let quadrati: Vec<i32> = numeri.iter().map(|&x| x * x).collect();
    println!("Quadrati: {:?}", quadrati);
}
```

In questo caso, *map* applica una chiusura a ciascun elemento dell'iteratore, trasformando ogni valore al quadrato. Il risultato viene poi raccolto in un nuovo *Vec<i32>* usando *collect*. Questo metodo è particolarmente flessibile, poiché può raccogliere i risultati in diverse collezioni, a seconda del tipo che viene specificato o inferito dal contesto.
Un altro esempio comune è l'uso di *filter* per selezionare solo gli elementi che

soddisfano una certa condizione:

```
fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    let pari: Vec<i32> = numeri.iter().filter(|&&x| x % 2 == 0).collect();
    println!("Numeri pari: {:?}", pari);
}
```

Qui, *filter* applica una chiusura che restituisce *true* solo per i numeri pari, e *collect* raccoglie i risultati filtrati in un nuovo vettore.
Passiamo a qualcosa di più avanzato, come l'uso di *fold*, che consente di ridurre una sequenza a un singolo valore applicando un'operazione binaria. Per esempio, possiamo sommare tutti i numeri in una collezione:

```
fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    let somma: i32 = numeri.iter().fold(0, |acc, &x| acc + x);
    println!("Somma: {}", somma);
}
```

Nel codice, *fold* inizia con un accumulatore (0) e applica la chiusura per sommare ogni elemento al valore accumulato, restituendo infine il risultato totale. Questa operazione è molto versatile e può essere utilizzata per una varietà di riduzioni, non solo per sommare, ma anche per moltiplicare, concatenare stringhe, o qualsiasi altra operazione aggregata.
Gli iteratori in Rust sono anche *lazy*, *cioè non eseguono effettivamente l'operazione fino a quando non vengono consumati,* per esempio da un ciclo *for* o da una funzione come *collect*. Questo comportamento permette di costruire pipeline di trasformazione molto efficienti, poiché ogni elemento viene processato solo quando necessario, ad esempio:

```
fn main() {
    let numeri = vec![1, 2, 3, 4, 5];
    let risultato: Vec<i32> = numeri.iter()
        .filter(|&&x| x % 2 != 0)
```

```
        .map(|&x| x * x)
        .collect();
    println!("Quadrati dei numeri dispari: {:?}", risultato);
}
```

I numeri dispari vengono prima filtrati, poi elevati al quadrato, e infine raccolti in un nuovo vettore. La catena di operazioni è espressiva e chiaramente leggibile, incarnando l'idioma del linguaggio relativo alla composizione di funzioni e alla manipolazione funzionale delle collezioni.
Rust incoraggia l'uso di iteratori e funzioni di utilità per scrivere codice che sia non solo sicuro ed efficiente, ma anche altamente espressivo e riutilizzabile. Questi idiomi permettono di trattare sequenze di dati in modo naturale, riducendo il rischio di errori legati alla gestione manuale delle collezioni e migliorando la leggibilità del codice. Gli iteratori rappresentano quindi una parte essenziale del modo idiomatico di scrivere Rust, e il loro uso diffuso nelle librerie standard e nei progetti contribuisce a rendere il codice più conciso, modulare e performante.

### *Funzioni asincrone*

Le funzioni asincrone rappresentano un elemento fondamentale per scrivere codice concorrenziale e non bloccante, particolarmente utile in applicazioni che richiedono l'esecuzione simultanea di compiti che possono comportare attese, come operazioni I/O, chiamate di rete o interazioni con database.
In Rust, la programmazione asincrona si basa su un meccanismo di *futures* (futuri) che rappresentano valori o risultati che potrebbero non essere immediatamente disponibili *ma saranno completati in un momento successivo*. Una funzione asincrona è dichiarata con la parola chiave *async*, non restituisce direttamente un valore finale, ma piuttosto un *future*, il quale è un oggetto che incapsula il risultato dell'operazione asincrona e che può essere atteso (*awaited*) per ottenere il valore una volta che è disponibile.
Consideriamo un esempio di una funzione asincrona che simula un'attesa prima di completare un'operazione:

```
use std::time::Duration;
use tokio::time::sleep;

async fn operazione_lenta() -> i32 {
    println!("Inizio dell'operazione lenta...");
    sleep(Duration::from_secs(2)).await;
    println!("Operazione completata!");
    42
}

#[tokio::main]
async fn main() {
    let risultato = operazione_lenta().await;
    println!("Risultato: {}", risultato);
}
```

In questo esempio, *operazione_lenta* è una funzione asincrona che restituisce un i32. Al suo interno viene utilizzata la funzione *sleep* di *tokio* per simulare un'attesa di 2 secondi. La chiamata a *sleep* è seguita dall'operatore *.await*, che sospende l'esecuzione della funzione fino al completamento dell'operazione. Durante questo tempo, altre operazioni asincrone possono essere eseguite, permettendo al programma di continuare a funzionare senza blocchi.

Il *main asincrono*, anch'esso dichiarato con *async*, utilizza l'attributo *#[tokio::main],* che specifica che la funzione *main* sarà eseguita all'interno del runtime di *tokio* (https://tokio.rs/), un framework popolare per la programmazione asincrona. Quando *operazione_lenta().await* viene chiamato, il programma aspetta il completamento del *future* e, una volta completato, il valore restituito viene assegnato a risultato, che poi viene stampato.

Un aspetto chiave della programmazione asincrona è che le funzioni *async* sono *lazy,* ovvero non iniziano l'esecuzione fino a quando non vengono attese. Questo permette di comporre più operazioni asincrone in modo efficiente, utilizzando combinatori e altre tecniche per coordinare l'esecuzione di più compiti in parallelo o in sequenza.

Ad esempio, è possibile lanciare più operazioni asincrone in parallelo utilizzando

*tokio::join!*:

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

async fn operazione1() -> i32 {
    sleep(Duration::from_secs(2)).await;
    println!("Operazione 1 completata");
    1
}

async fn operazione2() -> i32 {
    sleep(Duration::from_secs(3)).await;
    println!("Operazione 2 completata");
    2
}

#[tokio::main]
async fn main() {
    let (ris1, ris2) = tokio::join!(operazione1(), operazione2());
    println!("Risultati: {}, {}", ris1, ris2);
}
```

Si parte con *operazione1* e *operazione2* che vengono eseguite in parallelo. La macro *tokio::join!* attende che entrambe le operazioni siano completate, restituendo una tupla con i risultati. Anche se *operazione2* richiede più tempo per completarsi, l'uso di *join!* permette di eseguire le operazioni in modo concorrente, ottimizzando il tempo di esecuzione complessivo.
Un altro punto importante è che Rust, a differenza di altri linguaggi, non ha un runtime asincrono incorporato. Invece, si affida a librerie esterne come *tokio* o *async-std* per gestire l'esecuzione di queste operazioni. Questi runtime sono responsabili di eseguire i *futures*, gestire la concorrenza, e fornire primitive asincrone come i timer e le primitive di I/O asincrono.
Rust offre anche una stretta integrazione tra il sistema di tipi e la programmazione asincrona, garantendo che il codice sia sicuro e privo di data races (condizioni di gara sui dati). Ogni *future* è *Send* e *Sync* solo se i dati al suo interno lo permettono, assicurando che l'accesso concorrente ai dati avvenga in modo sicuro.

Inoltre, il compilatore verifica che tutte le referenze all'interno di una funzione *async* siano valide per tutta la durata del *future*, prevenendo errori comuni come i dangling references.
Le funzioni asincrone offrono una combinazione unica di prestazioni, sicurezza e flessibilità, permettendo di scrivere codice concorrente senza sacrificare la prevedibilità o la robustezza.

### *Quiz & Esercizi*

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.
2) Scrivere una funzione che prenda come input una lista di numeri e restituisca la somma dei suoi elementi.
3) Scrivere una funzione che prenda due argomenti e restituisca la loro somma.
4) Definire una funzione che prenda un numero come parametro e restituisca il suo quadrato.
5) Scrivere una funzione che prenda in input una stringa e ne restituisca la lunghezza.
6) Scrivere una funzione che prenda in input una lista di parole e ne restituisca una lista di queste in ordine alfabetico.
7) Scrivere una funzione che prenda in input una lista di numeri e ne restituisca una di numeri pari.
8) Scrivere una funzione lambda che prenda in input un intero e ne restituisca il suo doppio.
9) Scrivere una funzione lambda che prenda in input una stringa e restituisca la stessa in maiuscolo.
10) Scrivere una funzione lambda che prenda in input due numeri e restituisca il loro prodotto.
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.
12) Scrivere una funzione che restituisca la somma di due numeri e li converta in float.
13) Scrivere una funzione che restituisca una stringa applicandogli un prefisso.
14) Scrivere una funzione che applichi un callback ad ogni elemento di una lista.
15) Scrivere una funzione che sommi gli elementi di una lista e applichi una funzione di callback al risultato.
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*.
17) Scrivere un programma che effettui richieste HTTP a più URL in contemporanea.
18) Scrivere un programma che effettui la lettura di file in parallelo.
19) Scrivere un programma che effettui una richiesta HTTP e che attenda un tempo massimo di cinque secondi per ricevere la risposta.

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

***Riassunto***

Abbiamo esplorato diversi concetti legati alle funzioni in Rust, partendo dalla loro definizione, sintassi e scopo, fino a temi più avanzati come il borrowing, il lifetimes e i trait bounds. Nella definizione di una funzione in Rust, è fondamentale comprendere come il linguaggio gestisca in modo rigoroso i tipi di input e output e come il compilatore aiuti a prevenire errori comuni attraverso un sistema di verifica statica.

Successivamente, abbiamo esaminato la gestione dell'ownership e del borrowing nel contesto delle funzioni, chiarendo come le variabili passate a una funzione possano trasferire proprietà o semplicemente essere prese in prestito in modo sicuro. Questo ha portato all'approfondimento dell'importanza dei lifetime e delle annotazioni esplicite che permettono al compilatore di gestire correttamente la durata delle variabili in relazione alle funzioni, specialmente quando ci sono riferimenti multipli o complessi.

Abbiamo poi trattato le funzioni generiche e i trait bounds, che consentono di scrivere codice generico e riutilizzabile, limitando i tipi ammessi dalle funzioni tramite trait come *PartialOrd* o *Display*. Ci siamo soffermati anche sulle chiusure e funzioni anonime, spiegando come queste catturino variabili dal contesto in modi diversi (per valore, mutabilità o consumo) e siano gestite da specifici trait come *Fn, FnMut* e *FnOnce*.

Un altro tema chiave affrontato sono state le funzioni di ordine superiore, che accettano altre funzioni come parametri o restituiscono funzioni come risultato, promuovendo una programmazione funzionale più espressiva. Abbiamo anche approfondito il concetto di *lifetime elision*, spiegando come Rust spesso possa inferire i lifetime senza la necessità di annotazioni esplicite, ma quando sia comunque necessario definirli per garantire la sicurezza.

Nell'ambito delle funzioni asincrone, abbiamo discusso l'uso del runtime asincrono per gestire in modo efficiente le operazioni non bloccanti, facendo emergere il modello di programmazione reattiva basato su *future* e callback asincroni.

Infine, abbiamo affrontato il concetto di callback, spiegando come le funzioni anonime o i puntatori a funzione possano essere passati ad altre funzioni e invocati in seguito. Si è discusso della sicurezza e dell'efficienza offerte dal linguaggio grazie all'integrazione con i concetti di ownership, mutabilità e concorrenza, chiarendo come Rust possa garantire la sicurezza della memoria anche con i callback in ambienti multi-threaded o asincroni. In tutto ciò, si dimostra un linguaggio che, pur complesso, offre un controllo dettagliato sulla gestione della memoria e della concorrenza, grazie a strumenti come i trait, le chiusure, le funzioni asincrone e i lifetime.

# # 4 - I trait e le struct

*Programmazione a tipi forti*
*Trait e Struct*
*Oggetti Trait*
*Lifetimes*

I trait rappresentano un concetto chiave, assimilabile alle interfacce di altri linguaggi di programmazione, ma con alcune peculiarità che li distinguono. Un trait è una collezione di metodi che può essere implementata per diverse tipologie di dati. Questi metodi possono avere un comportamento predefinito, oppure possono essere lasciati come "astratti", cioè privi di una specifica implementazione, lasciando ai tipi concreti che implementano il trait la responsabilità di definirne il funzionamento.
Inoltre permettono a Rust di seguire il principio di astrazione, garantendo flessibilità e riutilizzo del codice, senza compromettere la sicurezza tipica del linguaggio. Attraverso l'uso di trait, si apre la porta alla definizione di comportamenti condivisi fra diversi tipi di dati, pur mantenendo l'efficienza e la sicurezza della gestione della memoria. Il compilatore può utilizzarli per verificare la correttezza dei tipi in fase di compilazione, riducendo il rischio di errori run-time.
Un'altra caratteristica rilevante è la possibilità di estendere i trait. Ciò permette di definirne di nuovi che dipendono da altri, fornendo un potente meccanismo di composizione di funzionalità e promuovendo la modularità. Inoltre, è supportato il loro utilizzo per la definizione di metodi generici, il che rende possibile scrivere funzioni o strutture dati che funzionano con diversi tipi di input, purché rispettino

un certo comportamento (cioè implementino uno o più trait).
I trait vengono anche utilizzati per gestire il concetto di *polimorfismo ad-hoc*, consentendo di scrivere funzioni che possono accettare parametri di tipi diversi, a patto che questi rispettino le interfacce definite dai trait. Questo differisce dal polimorfismo delle classi tradizionali, offrendo un controllo maggiore sul comportamento delle funzioni e sul tipo di dati che accettano.
Non è tutto, abbiamo anche il cosiddetto *trait object*, un meccanismo che consente di trattare variabili di diversi tipi come se appartenessero allo stesso tipo. Tuttavia, a differenza delle interfacce tradizionali di linguaggi orientati agli oggetti, il suo utilizzo comporta una perdita di alcune ottimizzazioni a livello di prestazioni, poiché introduce una forma di dinamicità durante l'esecuzione.
Infine, è importante notare che i trait supportano sia l'implementazione automatica per alcuni tipi predefiniti, sia la possibilità di estenderne l'uso su tipi esterni, una tecnica nota come "*implementazioni orfane*", che però è soggetta a restrizioni per evitare conflitti e ambiguità nelle definizioni. L'utilizzo attento e bilanciato dei trait permette a questo linguaggio di mantenere la sua reputazione di essere performante e sicuro, in grado di offrire elevati livelli di astrazione senza sacrificare il controllo sui dettagli implementativi.
Il concetto di trait si collega in modo naturale a quello delle struct, che rappresentano un altro elemento fondamentale, utilizzate per definire nuovi tipi di dati complessi composti da altri tipi, analogamente a quanto avviene con le classi in linguaggi orientati agli oggetti, ma con una distinzione cruciale, Rust non è un linguaggio OOP tradizionale e, di conseguenza, *le struct non possiedono concetti come ereditarietà o polimorfismo di classe*.
Le struct consentono la definizione di dati organizzati in campi, ciascuno con un proprio tipo, offrendo così un potente strumento per modellare concetti del mondo reale o strutture dati specifiche. La loro funzione principale è incapsulare i dati in un tipo coerente e ben definito, permettendo una gestione più ordinata e sicura delle informazioni. Tuttavia, a differenza delle classi, le struct non contengono

metodi di per sé: il comportamento dei dati incapsulati viene tipicamente definito tramite i trait. È qui che i due concetti si intersecano.
Quando si associa un trait a una struct, si permette a quest'ultima di "adottare" un comportamento specifico. I metodi definiti dal trait possono essere implementati per la struct, fornendo così un'interfaccia uniforme per diversi tipi che possono condividere lo stesso comportamento pur mantenendo una rappresentazione interna differente. Questo favorisce la separazione tra i dati (gestiti dalle *struct*) e i comportamenti (definiti dai *trait*), un approccio che riduce la complessità tipica dell'ereditarietà nelle gerarchie di classi presenti in altri linguaggi.
Inoltre, l'uso combinato consente di sfruttare il sistema di tipi di Rust in maniera estremamente flessibile e potente. Le struct possono essere utilizzate come tipi concreti per funzioni generiche che accettano parametri basati su trait, permettendo così a diverse struct di essere trattate in modo polimorfico, purché implementino i trait necessari. Questo permette di definire funzioni e strutture dati generiche senza sacrificare l'efficienza, grazie alla capacità del compilatore di Rust di monomorfizzare i tipi generici e risolvere il comportamento in fase di compilazione.
Un ulteriore collegamento significativo riguarda la gestione della memoria e la sicurezza tipica di Rust. Le struct possono incapsulare risorse come puntatori o riferimenti, e i trait vengono spesso utilizzati per definire come queste devono essere gestite, ad esempio in termini di clonazione, copia o distruzione. L'implementazione dei trait consente di verificare in fase di compilazione che tutte le regole di proprietà, prestiti e vita utile dei dati siano rispettate, riducendo il rischio di errori come i dangling pointers o le doppie deallocazioni.
In sintesi, l'interazione tra struct e trait rappresenta una delle caratteristiche più eleganti e potenti del linguaggio. Gli ultimi forniscono il mezzo per estendere e definire comportamenti generici e flessibili, mentre le struct offrono un meccanismo robusto per organizzare e gestire i dati. Questa separazione tra dati e comportamento, unita a un sistema di tipi avanzato, permette a Rust di

raggiungere un equilibrio unico tra sicurezza, performance e astrazione, superando molte delle limitazioni imposte dai paradigmi di programmazione più tradizionali.
Questa stretta relazione non si limita solo alla separazione tra dati e comportamento, ma si estende anche alla capacità di creare sistemi modulari e flessibili attraverso la composizione. Anziché affidarsi a gerarchie rigide come avviene nei linguaggi orientati agli oggetti tradizionali, Rust adotta un approccio compositivo che consente di combinare diversi trait per definire comportamenti complessi in modo più sicuro e flessibile. Questa filosofia permette di costruire soluzioni scalabili e manutenibili, valorizzando l'interoperabilità tra struct e trait.
La composizione è considerata un meccanismo più flessibile e sicuro per costruire software. In Rust, si preferisce comporre diversi trait per ottenere comportamenti complessi piuttosto che creare una gerarchia di classi. Le struct sono simili a quelle di linguaggi come C, ma possono contenere metodi, utilizzate per definire e manipolare i dati. I trait, d'altro canto, rappresentano una sorta di interfacce che definiscono un insieme di metodi che un tipo deve implementare per essere considerato conforme. Rust consente a una singola struct di implementare più trait, favorendo la composizione di comportamenti anziché l'ereditarietà, offrendo un potente strumento per definire interfacce, creare polimorfismo e riutilizzare codice comune senza duplicazioni. Questo rende i trait fondamentali nel linguaggio, offrendo un mezzo flessibile e modulare per creare gerarchie di tipo e comportamenti riutilizzabili senza le rigidità tipiche dell'ereditarietà classica.

***Programmazione a tipi forti***

La programmazione a tipi forti si distingue per la rigidità e la precisione con cui il linguaggio gestisce i tipi, garantendo che gli errori relativi al tipo siano catturati durante la compilazione, *piuttosto che a runtime*. Questo approccio si riflette profondamente nell'interazione tra trait e struct, dove Rust impone regole stringenti per assicurare la coerenza dei tipi, la sicurezza della memoria e la corretta gestione delle risorse.

Le struct permettono di definire tipi composti che raggruppano variabili di tipo diverso in un'unica entità. Per esempio, potresti avere una struct che rappresenta un punto su un piano cartesiano:

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

*Punto* è un nuovo tipo di dato, definito dall'utente, che contiene due interi. La definizione dei tipi è fondamentale, poiché Rust richiede che ogni valore associato a una struct sia conforme ai tipi dichiarati. Se si tenta di assegnare un valore che non corrisponde a quello previsto, il compilatore blocca l'operazione, come mostrato qui:

```
let p = Punto { x: 10, y: "20" }; // Errore: il tipo di 'y' non è i32.
```

In aggiunta alle struct, i trait permettono di stabilire comportamenti che possono essere condivisi tra tipi diversi. In pratica definiscono una collezione di metodi che un tipo può implementare. Ad esempio, potresti avere un trait che definisce un comportamento per calcolare la distanza tra due punti:

```
trait Distanza {
    fn distanza(&self, altro: &Self) -> f64;
}
```

La programmazione a tipi forti di Rust garantisce che un tipo non possa essere usato in contesti che richiedono un trait specifico, a meno che quel tipo non ne esegua esplicitamente uno. Se implementiamo *Distanza* per la struct *Punto*, otteniamo un comportamento chiaro e sicuro:

```
impl Distanza per Punto {
    fn distanza(&self, altro: &Self) -> f64 {
        let dx = (self.x - altro.x) as f64;
        let dy = (self.y - altro.y) as f64;
```

```
        (dx.powi(2) + dy.powi(2)).sqrt()
    }
}
```

In questo modo, il compilatore garantisce che qualsiasi utilizzo di distanza su un oggetto *Punto* sia corretto dal punto di vista del tipo. Questo evita errori di esecuzione e assicura che le operazioni avvengano solo tra tipi che soddisfano i requisiti imposti dal trait. Ad esempio, un tentativo di chiamare il metodo *distanza* su un tipo che non implementa *Distanza* produrrebbe un errore di compilazione, bloccando il programma prima che venga eseguito.
Un altro aspetto in questo ambito è la gestione dei generici. È possibile definire funzioni che lavorano su tipi che implementano un certo trait, mantenendo la flessibilità ma senza sacrificare la sicurezza dei tipi. Ad esempio, si può scrivere una funzione che accetta due oggetti che implementano il trait *Distanza*:

```
fn calcola_distanza<T: Distanza>(a: &T, b: &T) -> f64 {
    a.distanza(b)
}
```

Questo assicura che la funzione *calcola_distanza* possa essere utilizzata solo con tipi che implementano il trait *Distanza*, garantendo la correttezza a livello di tipi. Se si tenta di passare un tipo non compatibile, il compilatore segnalerà un errore prima della compilazione.
La rigidità del sistema dei tipi, unita all'uso dei trait e delle struct, permette di scrivere codice sia flessibile che sicuro. I tipi forti evitano molti errori comuni ad essi legati, riducendo la possibilità di incorrere in comportamenti imprevisti o bug difficili da individuare. Allo stesso tempo, grazie ai trait, Rust riesce a bilanciare la sicurezza del tipo con la possibilità di definire comportamenti generici e riutilizzabili.

### *Definizione e utilizzo delle struct*

Le struct rappresentano un modo per definire tipi di dati personalizzati che

possono contenere variabili, dette *campi*, con tipi eterogenei. *Sono simili alle classi di altri linguaggi orientati agli oggetti, ma senza includere meccanismi di ereditarietà.* Le struct consentono di raggruppare insieme dati correlati sotto un'unica entità, migliorando la chiarezza del codice e permettendo di gestire in modo più ordinato i dati complessi.
Per esempio, si può definire una struct chiamata *Persona* per rappresentarne una con un nome e un'età:

```
struct Persona {
    nome: String,
    eta: u8,
}
```

Questa definizione crea un tipo chiamato *Persona*, che ha due campi: *nome,* che è una String, e *eta*, che è un intero non negativo (u8). Una volta definita, si può creare un'istanza della struct assegnando valori ai suoi campi:

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

In questo caso, la variabile persona contiene un'istanza di *Persona*, con il nome "Mario" e un'età di 30 anni. I campi di una struct possono essere accessibili usando il punto (.), come mostrato nel seguente esempio:

```
println!("Nome: {}, Età: {}", persona.nome, persona.eta);
```

Le struct supportano anche mutabilità selettiva. Se ne viene dichiarata una come mutabile, è possibile modificare i suoi campi, ma solo quelli esplicitamente marcati come mutabili:

```
let mut persona = Persona {
    nome: String::from("Mario"),
    eta: 30,
```

```
};
persona.eta = 31;
```

In questo caso, l'età della persona viene aggiornata da 30 a 31.
Rust consente anche la definizione di costruttori tramite il blocco *impl*, che permette di associare funzioni a una struct. Per esempio, possiamo definire una funzione nuova che crea un'altra *Persona*:

```
impl Persona {
    fn nuova(nome: String, eta: u8) -> Persona {
        Persona { nome, eta }
    }
}
```

Questa funzione può essere chiamata per creare una nuova istanza di *Persona*:

```
let persona = Persona::nuova(String::from("Mario"), 30);
```

Una variante utile è la cosiddetta *tuple struct,* che è simile a una tuple ma con un tipo specifico. Ad esempio:

```
struct Punto(i32, i32);

let p = Punto(10, 20);
```

Qui, *Punto* è una struct che contiene due interi, e i campi possono essere accessibili tramite l'indice:

```
println!("x: {}, y: {}", p.0, p.1);
```

Le struct forniscono anche una base per garantire che i dati siano utilizzati in modo sicuro, rispettando le regole di ownership e borrowing. I campi possono essere passati per riferimento o spostati in base alle necessità, consentendo un controllo preciso sulla gestione della memoria e sul ciclo di vita degli oggetti.
Una delle differenze principali tra una struct e una *tuple struct* riguarda l'accessibilità e la leggibilità dei campi. Nella struct nominale come *Persona*, i

campi sono nominati e possono essere acceduti in maniera chiara tramite i loro nomi, il che rende il codice più leggibile. Perciò, quando si definisce una struct nominale come:

```
struct Persona {
    nome: String,
    eta: u8,
}
```

Il campo *nome* può essere facilmente riconosciuto e acceduto con *persona.nome*, offrendo una semantica chiara per chi legge il codice. Le tuple struct, invece, hanno campi anonimi, identificati solo dalla loro posizione numerica. Ad esempio:

```
struct Coordinate(i32, i32);
```

Qui, l'accesso ai campi avviene con indici, come *coordinate.0* e *coordinate.1*. Sebbene le tuple struct siano utili in contesti specifici, la loro leggibilità diminuisce con l'aumentare della complessità, poiché non è immediatamente chiaro cosa rappresentino i campi senza conoscere la definizione. Al contrario, le struct nominali offrono chiarezza sul significato di ciascun campo grazie ai nomi espliciti.
Un altro concetto importante è il *pattern di ownership* nelle struct. Ogni suo campo segue le regole di ownership, il che significa che può essere trasferito o preso in prestito a seconda del contesto. Quando si crea una *Persona*, l'ownership dei dati viene trasferita all'istanza della struct. Ad esempio:

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

Qui, la *String Mario* viene spostata nella struct, rendendo persona il proprietario di quei dati. Se si tenta di usare il campo nome altrove senza rispettare le regole di borrowing, il compilatore genererà un errore. Per prendere in prestito i campi di una struct senza trasferire l'ownership, si possono usare i *riferimenti*:

```
let nome_ref = &persona.nome;
```

In questo caso, *nome_ref* è un riferimento a *persona.nome,* e Rust garantisce che il dato non venga modificato o spostato finché il riferimento è in uso. Se la struct è dichiarata come mutabile, anche i campi possono essere mutabili:

```
let mut persona = Persona {
    nome: String::from("Mario"),
    eta: 30,
};
persona.eta = 31; // mutabile
```

Qui, poiché la struct è dichiarata mutabile, è possibile modificare il campo *eta* senza trasferire l'ownership.
I modificatori di visibilità offrono un controllo preciso sull'accesso ai campi di una struct. Per impostazione predefinita, i suoi campi sono privati, cioè accessibili solo all'interno del modulo in cui sono definiti. Se si desidera che siano accessibili dall'esterno, è necessario dichiararli pubblici con il modificatore *pub*. Ad esempio, possiamo rendere il nome di una *Persona* pubblico, ma mantenere l'*età* privata:

```
pub struct Persona {
    pub nome: String,
    eta: u8,
}
```

Quindi, *nome* è accessibile dall'esterno del modulo, mentre *eta* rimane privata. Quando si tenta di accedervi dall'esterno del modulo, il compilatore segnalerà un errore:

```
println!("Nome: {}", persona.nome); // Funziona, perché 'nome' è pubblico
println!("Età: {}", persona.eta); // Errore, perché 'eta' è privato
```

Questa capacità di specificare la visibilità dei campi consente una maggiore flessibilità nel progettare il comportamento di una struct. Questi possono essere resi pubblici o privati in base alle esigenze, garantendo che l'accesso ai dati sia

sicuro e controllato.
Oltre a ciò, le struct possono essere sia immutabili che mutabili, e questo si riflette nel modo in cui è possibile accedervi e modificare i campi. Quando una struct viene dichiarata immutabile, questi ultimi non possono essere modificati dopo la creazione:

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

Qui, *persona* è immutabile. Questo significa che non è possibile modificare alcuno dei suoi campi, e se si tenta di farlo, il compilatore genererà un errore:

```
persona.eta = 31; // Errore: non è possibile modificare un campo di una struct immutabile
```

Se, invece, la struct viene dichiarata come mutabile, i suoi campi possono essere modificati. Questo concetto, come avremo oramai intuito, è trasversale in Rust:

```
let mut persona = Persona {
    nome: String::from("Mario"),
    eta: 30,
};
persona.eta = 31; // Funziona, perché la struct è mutabile
```

La parola chiave *mut* indica al compilatore che i campi della struct possono essere modificati. È importante notare che Rust *distingue chiaramente tra mutabilità della struct e mutabilità dei riferimenti ai suoi campi*. Anche se una struct è mutabile, i riferimenti ai suoi campi possono essere sia mutabili che immutabili, a seconda di come vengono gestiti, vediamolo meglio:

```
let mut persona = Persona {
    nome: String::from("Mario"),
    eta: 30,
};
```

```
// Prendere un riferimento immutabile a nome
let nome_ref = &persona.nome;
// Prendere un riferimento mutabile a eta
let eta_ref = &mut persona.eta;

// Questo funziona: il campo eta è mutabile tramite eta_ref
*eta_ref = 31;

// Ma non si può modificare nome tramite nome_ref perché è immutabile
```

Oltre alla semplice definizione, possiamo anche associare metodi alle struct tramite il blocco *impl* (abbreviazione di implementazione), il quale consente di definire funzioni e metodi che operano sulle istanze di una struct, migliorando la modularità e la leggibilità del codice. I metodi sono simili alle funzioni, ma la differenza principale è che i primi hanno un parametro implicito chiamato *self*, che rappresenta l'istanza della struct su cui il metodo viene invocato.
Per esempio, possiamo definire un metodo *mostra_dati* per la struct *Persona*, che ne stampa i dettagli:

```
impl Persona {
    fn mostra_dati(&self) {
        println!("Nome: {}, Età: {}", self.nome, self.eta);
    }
}
```

Il parametro *&self* indica che il metodo prende in prestito un riferimento immutabile all'istanza della struct. Questo significa che il metodo può leggere i suoi campi, ma non può modificarli. Possiamo chiamare questo metodo su una istanza di *Persona* in questo modo:

```
persona.mostra_dati();
```

Se volessimo definire un metodo che modifica i campi della struct, dobbiamo usare un riferimento mutabile a *self*. Ad esempio, possiamo definire un metodo *incrementa_eta* che aumenta l'età della persona di un anno:

```
impl Persona {
    fn incrementa_eta(&mut self) {
        self.eta += 1;
    }
}
```

Qui, *&mut self* permette al metodo di modificare i campi della struct. Per usarlo, quest'ultima deve essere dichiarata mutabile:

```
let mut persona = Persona {
    nome: String::from("Mario"),
    eta: 30,
};
persona.incrementa_eta();
persona.mostra_dati(); // Nome: Mario, Età: 31
```

Inoltre, possiamo definire funzioni che non richiedono un'istanza della struct e *che agiscono come costruttori.* Una funzione costruttore viene spesso utilizzata per creare nuove istanze di una struct e di solito viene chiamata *new*. Queste funzioni non accettano *self* come parametro, perché sono associate al tipo, non a una singola istanza. Per esempio, possiamo definire una funzione *nuova* che crea un'altra *Persona*:

```
impl Persona {
    fn nuova(nome: String, eta: u8) -> Persona {
        Persona { nome, eta }
    }
}
```

Qui, la funzione *nuova* prende due parametri, nome e eta, e restituisce un'altra istanza di *Persona*. Possiamo usare questa funzione per creare una persona in questo modo:

```
let persona = Persona::nuova(String::from("Luigi"), 25);
persona.mostra_dati(); // Nome: Luigi, Età: 25
```

Grazie a *impl,* è possibile separare la logica associata alla struct, come la creazione

di nuove istanze o la modifica dei campi, direttamente nella definizione della struct stessa, rendendo il codice più organizzato e manutenibile.

### *Trait, astrazione e comportamento*

I trait sono uno strumento chiave per definire astrazioni di comportamento, in modo simile alle interfacce nei linguaggi orientati agli oggetti, *ma con alcune differenze sostanziali.* Un trait è una collezione di metodi che un tipo può implementare, permettendo di definire comportamenti comuni e garantire che certi metodi siano disponibili su tipi che implementano quel trait. Questo non contiene dati, ma solo la definizione di metodi che possono essere astratti (cioè senza implementazione) o avere un'implementazione predefinita.
Ad esempio, possiamo definire un trait chiamato *Descrivibile*, che rappresenta il concetto di "descrivere" un oggetto. Questo trait può essere implementato da vari tipi, come la struct *Persona*:

```
trait Descrivibile {
    fn descrivi(&self) -> String;
}
```

In questo caso, il metodo *descrivi* è astratto, il che significa che i tipi che implementano questo trait devono fornire una loro versione di tale metodo. Ad esempio, possiamo implementare *Descrivibile* per *Persona* in questo modo:

```
impl Descrivibile per Persona {
    fn descrivi(&self) -> String {
        format!("Nome: {}, Età: {}", self.nome, self.eta)
    }
}
```

Con questa implementazione, ogni istanza di *Persona* avrà il metodo *descrivi,* che restituisce una stringa descrittiva. Possiamo usare questo trait per descrivere una persona:

```
let persona = Persona::nuova(String::from("Mario"), 30);
```

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

Un tratto distintivo dei trait rispetto alle interfacce nei linguaggi orientati agli oggetti è che non richiedono necessariamente metodi astratti; possono anche fornire un'implementazione predefinita per uno o più metodi. Ciò significa che un tipo può implementare un trait senza dover ridefinire tutti i metodi, a meno che non sia necessario modificare il comportamento predefinito. Ad esempio, possiamo aggiungere un'implementazione predefinita del metodo *descrivi* per qualsiasi tipo che implementi *Descrivibile*:

```
trait Descrivibile {
    fn descrivi(&self) -> String {
        String::from("Oggetto non descritto")
    }
}
```

Ora, se una struct implementa *Descrivibile* ma non fornisce una propria implementazione di *descrivi,* verrà utilizzata questa versione predefinita. Tuttavia, se necessario, un tipo può comunque sovrascrivere il comportamento predefinito fornendo la propria implementazione.
Un'altra differenza importante tra i trait e le interfacce nei linguaggi orientati agli oggetti riguarda il concetto di *dynamic dispatch* e *static dispatch*. Nei linguaggi orientati agli oggetti, le interfacce sono spesso utilizzate per fornire il supporto al polimorfismo tramite il *dynamic dispatch*, ovvero la risoluzione del metodo da eseguire a runtime. In Rust, i trait possono essere usati per ottenere un comportamento simile, utilizzando puntatori di trait (*&dyn Trait*), ma il linguaggio incoraggia fortemente l'uso dello *static dispatch*, che viene risolto a compile-time per garantire una maggiore efficienza.
Per esempio, possiamo scrivere una funzione che accetta qualsiasi tipo che implementa *Descrivibile* utilizzando *static dispatch*:

```
fn stampa_descrizione<T: Descrivibile>(oggetto: &T) {
    println!("{}", oggetto.descrivi());
```

```
}
```

In questo caso, il compilatore genera versioni specifiche di questa funzione per ogni tipo che implementa *Descrivibile*, migliorando le prestazioni senza il costo di un *dynamic dispatch* a runtime.
Al contrario, se si desidera un polimorfismo a runtime, si può usare una versione con puntatore di trait:

```
fn stampa_descrizione_dyn(oggetto: &dyn Descrivibile) {
    println!("{}", oggetto.descrivi());
}
```

Questa versione, pur essendo meno efficiente in termini di prestazioni, permette di lavorare con tipi eterogenei che implementano *Descrivibile* a runtime.
Infine, rispetto alle interfacce, i trait non sono vincolati dall'ereditarietà tipica degli oggetti. Un tipo può implementare molti trait diversi, e i trait stessi possono essere composti tra loro, rendendoli più flessibili rispetto a un modello basato su classi e interfacce tradizionali. Questo approccio fornisce una maggiore modularità e riduce la complessità delle gerarchie di classi tipiche nei linguaggi orientati agli oggetti.
Il meccanismo di implementazione dei trait è fondamentale per conferire comportamenti a tipi personalizzati come le struct. Per farlo, è necessario utilizzare la parola chiave *impl* seguita dal nome del trait e dal tipo per cui viene implementato. Ciò consente di associare una serie di metodi e funzionalità che la struct dovrà rispettare. Come già visto con *Descrivibile*, l'implementazione di un trait su una struct permette di specificare come quel tipo si comporterà quando viene usato con metodi definiti dal trait.
Per estendere l'esempio della struct *Persona*, supponiamo di voler implementare il trait *Clone*, che è un trait predefinito in Rust. *Clone* permette di creare una copia profonda di un valore, ovvero una copia in cui tutti i dati vengono duplicati anziché essere semplicemente referenziati. Questo può essere utile quando una struct contiene dati allocati dinamicamente, come una String, che non può essere

semplicemente copiata per valore. L'implementazione di *Clone* su *Persona* consente di duplicarne un'istanza in modo sicuro:

```
#[derive(Clone)]
struct Persona {
    nome: String,
    eta: u8,
}
```

L'attributo *#[derive(Clone)]* indica al compilatore di generare automaticamente l'implementazione di *Clone* per *Persona*, evitando la necessità di scrivere manualmente il codice. Ora possiamo clonare una persona in questo modo:

```
let persona1 = Persona {
    nome: String::from("Mario"),
    eta: 30,
};
let persona2 = persona1.clone(); // Cloniamo 'persona1'
```

Analogamente, un altro trait importante è *Copy*. A differenza di *Clone*, è usato per tipi che possono essere copiati in modo triviale, senza la necessità di una copia profonda. NB. In Rust, i tipi che possono essere "copiati in modo triviale" sono quelli per cui la copia è un'operazione semplice e poco costosa in termini di risorse. Questo significa che, invece di trasferire la proprietà del dato (come avviene per la maggior parte dei tipi), viene fatta una copia bit a bit del valore senza alcuna logica aggiuntiva. Il trait *Copy* definisce questa proprietà: i tipi che lo implementano, come numeri primitivi e booleani, non richiedono allocazioni di memoria o gestione complessa, quindi la copia è "triviale" perché avviene senza overhead:

```
let a = 5;
let b = a; // Copia triviale del valore 'a', senza trasferimento di proprietà.
```

Tipi primitivi come interi e booleani implementano *Copy* per impostazione predefinita. Tuttavia, una struct può implementare *Copy* solo se tutti i suoi campi implementano a loro volta questa istruzione. Dato che *String* non lo fa, la struct *Persona* non può implementare questo trait, a meno che non vengano utilizzati tipi primitivi per i campi.

Un altro trait molto comune è *Debug*, che permette di stampare il contenuto di una struct in un formato leggibile per il debugging. Anche questo può essere derivato automaticamente:

```
#[derive(Debug)]
struct Persona {
    nome: String,
    eta: u8,
}
```

Con *Debug* implementato, è possibile stampare una rappresentazione leggibile di Persona usando la macro *println!* con il formato {:?}:

```
let persona = Persona {
    nome: String::from("Mario"),
    eta: 30,
};
println!("{:?}", persona);
```

Il risultato sarà una stringa leggibile contenente il nome e l'età della persona, utile durante lo sviluppo per visualizzare rapidamente lo stato delle istanze delle struct.
Un altro trait di base è *Drop*, il quale consente di definire comportamenti personalizzati da eseguire quando un'istanza di una struct viene distrutta. Questo è utile per liberare risorse o eseguire operazioni di pulizia. Ad esempio, possiamo implementare *Drop* su *Persona* per stampare un messaggio quando un'istanza viene eliminata:

```
impl Drop per Persona {
    fn drop(&mut self) {
        println!("Eliminata Persona: {}", self.nome);
    }
}
```

Quando un'istanza di *Persona* esce dallo scope, il metodo *drop* viene automaticamente invocato:

```
{
    let persona = Persona {
        nome: String::from("Mario"),
        eta: 30,
    };
} // Qui 'persona' esce dallo scope e viene eliminata, chiamando 'drop'
```

Alla fine del blocco, il messaggio "Eliminata Persona: Mario" verrà stampato, segnalando che l'istanza è stata distrutta.

Questi trait predefiniti offrono funzionalità di base che migliorano la flessibilità e la sicurezza nel gestire struct e tipi personalizzati. Usando il sistema di trait, è possibile aggiungere facilmente comportamenti complessi come clonazione, debugging, e gestione delle risorse, garantendo al contempo il controllo rigoroso della memoria e delle risorse gestite dal programma.

Oltre a ciò, i trait possono definire sia *metodi di istanza* che *metodi statici.* I primi operano su un'istanza di un tipo e hanno accesso a *self,* che può essere un riferimento immutabile (*&self*), un riferimento mutabile (*&mut self*) o lo stesso self trasferito per valore. I metodi statici, invece, non operano su un'istanza specifica e non hanno accesso a *self*; agiscono solo a livello del tipo.

Riprendendo l'esempio della *Persona*, possiamo aggiungere un metodo di istanza che restituisce una descrizione della persona e anche un metodo statico che crea una nuova persona predefinita:

```
trait Descrivibile {
    fn descrivi(&self) -> String; // Metodo di istanza
    fn nuova_persona() -> Self; // Metodo statico
}

impl Descrivibile per Persona {
    fn descrivi(&self) -> String {
        format!("Nome: {}, Età: {}", self.nome, self.eta)
    }

    fn nuova_persona() -> Self {
        Persona {
            nome: String::from("Sconosciuto"),
```

```
            eta: 0,
        }
    }
}
```

In questo esempio, *descrivi* è un metodo di istanza, che richiede un riferimento a una specifica *Persona* per restituire i dettagli. Invece, *nuova_persona* è un metodo statico che crea una nuova *Persona* predefinita senza la necessità di avere un'istanza esistente. Possiamo chiamare i metodi in questo modo:

```
let persona = Persona::nuova_persona();
println!("{}", persona.descrivi());
```

Un'altra caratteristica importante dei trait è il loro utilizzo per il polimorfismo ad-hoc, *che consente di lavorare con tipi diversi che condividono un comportamento comune, definito da un trait*. Questo è possibile sia tramite lo *static dispatch*, come mostrato in precedenza, sia tramite il *dynamic dispatch*. Quest'ultimo permette di risolvere il metodo da chiamare a runtime piuttosto che a compile-time, ed è utile quando non si conoscono i tipi esatti durante la compilazione.
Per implementare il polimorfismo ad-hoc usando il *dynamic dispatch,* si usa un puntatore al trait, come *&dyn Trait.* Riprendendo il trait *Descrivibile*, possiamo definire una funzione che accetta qualsiasi tipo che implementa il trait, usando il *dynamic dispatch*:

```
fn stampa_descrizione(oggetto: &dyn Descrivibile) {
    println!("{}", oggetto.descrivi());
}
```

Con questo approccio, possiamo passare qualsiasi tipo che implementa *Descrivibile* e la funzione risolverà quale metodo *descrivi* chiamare a runtime:

```
let persona = Persona::nuova_persona();
stampa_descrizione(&persona);
```

Il *dynamic dispatch* comporta un leggero overhead a livello di prestazioni, poiché

la risoluzione del metodo viene eseguita a runtime, ma fornisce una maggiore flessibilità in contesti dove non è noto il tipo esatto durante la compilazione.
Un'altra peculiarità dei trait è la regola degli orfani (*orphan rules*), che limita quali implementazioni possono essere definite. Quindi, essenzialmente, è possibile implementare un trait su un tipo solo se appartiene al proprio *crate*. Questo significa che non si può assegnare un trait esterno su un tipo esterno, per prevenire possibili conflitti in diversi contesti. Infatti, come appena visto, *Persona* e *Descrivibile* sono entrambi definiti nel nostro *crate*, e quindi possiamo implementare il trait su *Persona*. Non potremo farlo se *Descrivibile* appartenesse a un altro crate.
La motivazione alla base di questa restrizione è evitare che diversi moduli o *crate* forniscano implementazioni concorrenti o incoerenti dello stesso trait per lo stesso tipo. Senza le *orphan rules*, ci potrebbero essere implementazioni in conflitto che renderebbero il comportamento del programma ambiguo o imprevedibile.

### *Interazione tra struct e trait*

L'interazione tra struct e trait è uno degli elementi centrali nella progettazione di sistemi in Rust, permettendo di arricchire i tipi di comportamento e funzionalità comuni. Una struct come *Persona* può implementare diversi trait, sia definiti dall'utente sia predefiniti dal linguaggio, per conferire capacità specifiche. Ad esempio, è possibile che *Persona* implementi *Clone, Debug,* e *Descrivibile*, fornendo meccanismi di clonazione, debugging e descrizione personalizzata.
Implementare trait su struct permette di separare chiaramente i comportamenti dai dati, facilitando un approccio modulare. Come detto proviamo a utilizzare *Persona* in un sistema dove è necessario clonarla, stamparne una rappresentazione leggibile e fornirne una descrizione più specifica:

```
#[derive(Clone, Debug)]
struct Persona {
    nome: String,
    eta: u8,
```

```
}

trait Descrivibile {
    fn descrivi(&self) -> String;
}

impl Descrivibile per Persona {
    fn descrivi(&self) -> String {
        format!("Nome: {}, Età: {}", self.nome, self.eta)
    }
}
```

Qui vediamo l'utilizzo di *derive*, un'annotazione che permette di derivare in automatico l'implementazione di alcuni trait standard come *Clone* e *Debug*. Grazie a *#[derive(Clone, Debug)]*, non è necessario implementare manualmente questi trait, perché Rust genera automaticamente il codice per essi. Questo approccio è estremamente pratico quando non è richiesto un comportamento personalizzato. In alternativa, è possibile implementare un trait manualmente per fornire un comportamento più sofisticato o specifico.
Ad esempio, se desiderassimo che la clonazione di *Persona* alterasse un campo in modo specifico, potremmo implementare manualmente *Clone*:

```
impl Clone per Persona {
    fn clone(&self) -> Persona {
        Persona {
            nome: self.nome.clone(),
            eta: self.eta + 1, // Incrementiamo l'età nella clonazione
        }
    }
}
```

In questo caso, l'implementazione manuale permette un controllo più fine sul comportamento di clonazione, cosa che non sarebbe possibile con *derive*. Questa differenza tra comportamento derivato e comportamento implementato manualmente è cruciale quando si progettano sistemi che richiedono personalizzazioni specifiche.

Tuttavia, la capacità di clonare o ispezionare un valore si intreccia con il concetto di mutabilità: per poter clonare o modificare un dato, è essenziale comprendere se l'oggetto è immutabile o mutabile, dato che il linguaggio applica restrizioni rigorose sull'accesso ai dati in base alla loro mutabilità.
Perciò, il pattern di mutabilità si interseca profondamente con l'uso dei trait e delle struct. Poiché Rust separa rigorosamente la mutabilità, i trait possono definire metodi che operano su strutture sia immutabili sia mutabili, utilizzando rispettivamente *&self* e *&mut self*. Supponiamo che vogliamo modificare la *Persona* utilizzando un metodo di un trait che richiede un riferimento mutabile. Ad esempio, possiamo estendere *Descrivibile* con un metodo che cambia il nome della *persona*:

```
trait Descrivibile {
    fn descrivi(&self) -> String;
    fn cambia_nome(&mut self, nuovo_nome: String);
}

impl Descrivibile per Persona {
    fn descrivi(&self) -> String {
        format!("Nome: {}, Età: {}", self.nome, self.eta)
    }

    fn cambia_nome(&mut self, nuovo_nome: String) {
        self.nome = nuovo_nome;
    }
}
```

Questo codice mostra come l'implementazione di metodi mutabili tramite un trait rispetti il pattern. In seguito è necessario avere una struct mutabile per poter invocare il metodo che modifica i dati:

```
let mut persona = Persona {
    nome: String::from("Mario"),
    eta: 30,
};

println!("{}", persona.descrivi()); // Nome: Mario, Età: 30
persona.cambia_nome(String::from("Luigi"));
```

```
println!("{}", persona.descrivi()); // Nome: Luigi, Età: 30
```

L'uso combinato di trait, *derive*, e mutabilità crea un potente paradigma per la progettazione di sistemi robusti in Rust. I trait derivabili sono utili per i comportamenti generici che non richiedono modifiche, mentre l'implementazione manuale permette di personalizzare i comportamenti in modo specifico. Inoltre, la mutabilità controllata tramite trait garantisce che i cambiamenti alle struct siano sempre sicuri e tracciabili dal compilatore, contribuendo alla robustezza e affidabilità del codice.

### *Gli oggetti trait*

Questa categoria di oggetto permette di implementare il *polimorfismo dinamico*, offrendo la possibilità di lavorare con tipi diversi che implementano lo stesso trait, senza conoscere a priori il tipo esatto durante la compilazione. Questo si contrappone al *polimorfismo statico* ottenuto tramite i generici, dove il compilatore risolve tutto a compile-time, garantendo la massima efficienza. L'uso dei trait object è abilitato tramite la parola chiave *dyn*, che segnala al compilatore che la risoluzione del metodo deve avvenire a runtime.
Consideriamo l'esempio della struct *Persona* e del trait *Descrivibile*. Se vogliamo lavorare con un gruppo di oggetti che implementano *Descrivibile*, possiamo utilizzare un trait object per rappresentare una collezione eterogenea di tipi:

```
fn stampa_descrizione(oggetto: &dyn Descrivibile) {
    println!("{}", oggetto.descrivi());
}

let persona = Persona {
    nome: String::from("Mario"),
    eta: 30,
};
stampa_descrizione(&persona);
```

Qui, *&dyn Descrivibile* è un *trait object*. Questo permette di passare qualsiasi tipo

che implementi *Descrivibile*, e il metodo *descrivi* sarà risolto dinamicamente a runtime. La differenza fondamentale rispetto ai generici è che con i trait object il tipo concreto non è noto durante la compilazione; invece, con i generici il compilatore risolve tutti i tipi a compile-time, generando codice altamente ottimizzato. Il costo del dynamic dispatch consiste nel fatto che la risoluzione del metodo richiede un livello di indirezione aggiuntivo (termine tecnico per descrivere l'accesso a un valore attraverso un livello intermedio, come un puntatore o un riferimento), che introduce un piccolo overhead a livello di prestazioni, poiché il compilatore non può ottimizzare completamente il codice come nel caso del static dispatch.
Il polimorfismo statico (tramite generici) è spesso preferibile per motivi di performance, poiché il compilatore può generare una versione del codice per ogni tipo concreto, evitando il dynamic dispatch. Tuttavia, il polimorfismo dinamico tramite trait object ha il vantaggio di una maggiore flessibilità in situazioni dove i tipi esatti non sono noti o sono variabili a runtime. In questi casi, la versatilità dei trait object supera i costi in termini di prestazioni.
Un altro esempio pratico è quando si vuole gestire una collezione di tipi diversi che implementano lo stesso trait. Non possiamo usare un vettore di tipi diversi senza un trait object, perché Rust richiede che tutti gli elementi di un vettore abbiano lo stesso tipo. Utilizzando *Box<dyn Trait>,* possiamo aggirare questo problema:

```
let persone: Vec<Box<dyn Descrivibile>> = vec![
    Box::new(Persona {
        nome: String::from("Mario"),
        eta: 30,
    }),
    Box::new(Persona {
        nome: String::from("Luigi"),
        eta: 40,
    }),
];

for persona in persone {
    println!("{}", persona.descrivi());
}
```

In questo esempio, *Box<dyn Descrivibile>* viene usato per consentire l'allocazione dinamica di diversi tipi che implementano *Descrivibile*. Poiché *Persona* implementa *Descrivibile*, possiamo inserirla nel vettore. Ogni elemento del vettore è un puntatore a un tipo concreto che implementa il trait, ma la risoluzione del metodo descrivi avviene dinamicamente a runtime.
L'uso di *dyn* per i trait object offre vantaggi in termini di flessibilità e gestione di tipi eterogenei, ma ha alcuni svantaggi. Oltre al già menzionato overhead del dynamic dispatch, un'altra limitazione è che un trait object può utilizzare solo metodi che non richiedono il parametro di tipo *Self*, ovvero non è possibile chiamare metodi generici o associati che dipendono dal tipo concreto, perché il tipo esatto non è noto a runtime. Questo limita le capacità dei trait object rispetto all'uso dei generici.
In poche parole, i trait object consentono un'elevata flessibilità e permettono di lavorare con collezioni eterogenee di tipi che implementano lo stesso trait. Tuttavia, comportano un costo di performance e non supportano metodi che dipendono dal tipo concreto. In contesti dove le prestazioni sono cruciali e i tipi concreti sono noti, sono preferibili i generici. Infine, quando si richiede flessibilità e non è possibile conoscere i tipi esatti in fase di compilazione, i trait object sono la soluzione ideale per gestire comportamenti polimorfici.

### *Generics*

I *generics* permettono di scrivere codice flessibile e riutilizzabile, consentendo a tipi diversi di condividere la stessa logica. Un vantaggio principale è che permettono di definire funzioni, struct, e trait che operano su tipi generici, senza sacrificare le prestazioni. Il compilatore Rust utilizza la *monomorfizzazione*, un processo che genera versioni concrete del codice generico per ogni tipo utilizzato, perciò, nonostante l'uso di generics, il codice risultante è altamente ottimizzato.
Un esempio di concetto generico è l'uso di una funzione che può accettare

qualsiasi tipo, purché soddisfi determinate condizioni. Supponiamo di voler implementare una funzione che possa accettare qualsiasi oggetto che implementa il trait *Descrivibile*. Questo è un esempio classico di come i generics funzionano in combinazione con i trait:

```
fn stampa_descrizione<T: Descrivibile>(oggetto: T) {
    println!("{}", oggetto.descrivi());
}

let persona = Persona {
    nome: String::from("Mario"),
    eta: 30,
};
stampa_descrizione(persona);
```

In questo codice, la funzione *stampa_descrizione* accetta qualsiasi tipo *T*, purché quel tipo implementi il trait *Descrivibil*e. Questa forma di generics vincolati ai trait si chiama *bounded generics*. Il parametro *T* è limitato da un bound (*T: Descrivibile*), che garantisce che qualsiasi tipo passato a *stampa_descrizione* implementi il metodo descrivi. Questo approccio è molto potente poiché permette di creare codice altamente flessibile senza compromettere la sicurezza o l'ottimizzazione.

A livello di performance, l'uso dei generics con i trait è gestito tramite il processo di monomorfizzazione. Durante la compilazione, Rust crea versioni specifiche della funzione per ogni tipo concreto passato. Ad esempio, se si utilizza *Persona* e un altro tipo che implementa *Descrivibile*, Rust genererà due versioni separate di *stampa_descrizione*, una per ogni tipo, garantendo prestazioni equivalenti al codice non generico. Questo è lo static dispatch, dove il compilatore risolve le chiamate ai metodi a compile-time, consentendo ottimizzazioni aggressive.

Un'altra applicazione dei generics con i trait è l'implementazione di struct generiche. Supponiamo di voler creare una struct che possa contenere un tipo generico che implementa un particolare trait:

```
struct Contenitore<T: Descrivibile> {
    oggetto: T,
}

impl<T: Descrivibile> Contenitore<T> {
    fn descrivi(&self) -> String {
        self.oggetto.descrivi()
    }
}

let persona = Persona {
    nome: String::from("Mario"),
    eta: 30,
};
let contenitore = Contenitore { oggetto: persona };
println!("{}", contenitore.descrivi());
```

In questo caso, la struct *Contenitore* è generica e può contenere qualsiasi tipo che implementi *Descrivibile*. L'implementazione di metodi per *Contenitore* utilizza il trait *Descrivibile* per accedere ai metodi da esso definiti, rendendo la struct flessibile per vari tipi.

Una differenza chiave tra generic dispatch (dispatch statico) e dynamic dispatch è che, con i generics, il tipo concreto è noto durante la compilazione, il che consente ottimizzazioni come *l'inlining del codice (un'ottimizzazione in cui il compilatore sostituisce le chiamate a funzioni con il loro corpo direttamente nel codice chiamante, riducendo l'overhead della chiamata stessa. Questo può migliorare le performance, soprattutto per funzioni piccole o frequentemente chiamate, ma può aumentare la dimensione del codice compilato. Rust permette di suggerire al compilatore l'inlining tramite l'annotazione #[inline])*. Con il dynamic dispatch, utilizzato per i trait object come *&dyn Descrivibile*, la risoluzione dei metodi avviene a runtime, introducendo un overhead in termini di performance. In altre parole, mentre i generics offrono flessibilità e prestazioni elevate attraverso la monomorfizzazione, il polimorfismo dinamico con i trait object offre una maggiore flessibilità a scapito di una leggera perdita di efficienza.

Un ulteriore esempio dell'uso dei bounded generics potrebbe essere quello di una funzione che accetta non solo un tipo che implementa *Descrivibile,* ma anche uno

che implementa altri trait. Possiamo usare più bound per garantire che il tipo generico soddisfi più vincoli:

```
fn stampa_dettagli<T: Descrivibile + Clone>(oggetto: T) {
    let clone_oggetto = oggetto.clone();
    println!("{}", clone_oggetto.descrivi());
}
```

Qui, il tipo generico *T* deve implementare sia *Descrivibile* che *Clone*. Ciò consente di clonare l'oggetto prima di descriverlo, mostrando come i generics possano essere utilizzati in combinazione con più trait per costruire funzioni che operano su tipi complessi.
In sintesi, i generics consentono di scrivere codice estremamente flessibile e riutilizzabile, mantenendo comunque ottime prestazioni grazie alla monomorfizzazione. La combinazione con i trait permette di vincolare i generics a comportamenti specifici, mentre l'uso di bounded generics garantisce che i tipi rispettino determinati vincoli, assicurando al tempo stesso la sicurezza e l'efficienza del codice generato.

### *Modularità del codice con i trait*

Le conseguenze di tutto ciò che abbiamo visto finora, tra l'altro, sono che i trait sono anche strumenti potenti per promuovere la modularità del codice e la separazione delle preoccupazioni. Consentono di definire comportamenti in modo indipendente dai dati concreti, rendendo possibile la composizione di funzionalità attraverso più tipi e implementazioni. Questo approccio si integra bene con la filosofia di Rust, che favorisce un design modulare e orientato alla composizione piuttosto che all'ereditarietà classica dei linguaggi orientati agli oggetti.
I *supertrait* sono un meccanismo per estendere un trait. Quando quest'ultimo dipende da un altro, si parla di *supertrait*. Ciò è utile quando si vogliono definire relazioni gerarchiche tra i trait o combinare il comportamento di più trait in uno solo. Consideriamo l'estensione del nostro esempio della struct *Persona*

introducendo un nuovo trait *Identificabile*, che richiede che il tipo implementi anche *Descrivibile*:

```
trait Descrivibile {
    fn descrivi(&self) -> String;
}

trait Identificabile: Descrivibile {
    fn id(&self) -> u32;
}

impl Descrivibile per Persona {
    fn descrivi(&self) -> String {
        format!("Nome: {}, Età: {}", self.nome, self.eta)
    }
}

impl Identificabile per Persona {
    fn id(&self) -> u32 {
        self.eta as u32 // Usare l'età come identificatore fittizio
    }
}
```

Come anticipato, *Identificabile* è un supertrait di *Descrivibile*, il che significa che qualsiasi tipo che implementa il primo deve anche farlo per il secondo. *Questo permette di costruire gerarchie di trait che favoriscono una progettazione modulare, dove i singoli comportamenti possono essere aggiunti in modo incrementale e combinati tra loro.* Il vantaggio di questa composizione di trait è che permette di specificare interfacce più complesse e flessibili senza bisogno di usare l'ereditarietà rigida.

I trait sono anche un ottimo strumento per promuovere la separazione delle preoccupazioni, poiché consentono di separare i dati dai comportamenti. Ad esempio, un oggetto può contenere dati puri (come *Persona*), mentre i trait definiscono i comportamenti applicabili a quell'oggetto. Questo approccio è particolarmente utile in progetti più grandi, dove è importante mantenere una chiara distinzione tra responsabilità e implementazioni specifiche.

Progettare codice orientato ai trait richiede una mentalità che favorisca la composizione piuttosto che l'ereditarietà. Invece di creare gerarchie di classi, i trait permettono di costruire piccoli moduli di comportamento che possono essere combinati per creare strutture complesse. Supponiamo di voler aggiungere una variante di comportamento che consenta a *Persona* di essere salvata su disco. Possiamo definire un trait separato per questo comportamento e implementarlo senza alterare la logica esistente:

```
trait Salvabile {
    fn salva(&self);
}

impl Salvabile per Persona {
    fn salva(&self) {
        println!("Salvo persona: {}", self.descrivi());
    }
}
```

Ed ecco che, magicamente *Persona* può ora essere "salvata" grazie all'implementazione di *Salvabile*. Questo comportamento è completamente separato dagli altri trait come *Descrivibile* o *Identificabile*. La combinazione di questi trait consente di mantenere il codice modulare, riducendo il rischio di creare sistemi monolitici difficili da manutenere.

Nella progettazione del codice orientato ai trait, una buona pratica è definirne di piccoli e specifici. Invece di creare trait che tentano di descrivere comportamenti complessi o generici, è preferibile segmentare i comportamenti in moduli riutilizzabili e combinabili. Ad esempio, si potrebbe separare il comportamento di un oggetto *serializzabile* da quello di un oggetto *descrivibile*:

```
trait Descrivibile {
    fn descrivi(&self) -> String;
}

trait Serializzabile {
    fn serializza(&self) -> String;
```

```
}

impl Descrivibile per Persona {
    fn descrivi(&self) -> String {
        format!("Nome: {}, Età: {}", self.nome, self.eta)
    }
}

impl Serializzabile per Persona {
    fn serializza(&self) -> String {
        format!("{{\"nome\": \"{}\", \"eta\": {}}}", self.nome, self.eta)
    }
}
```

Qui, *Persona* è *descrivibile* e *serializzabile*, ma i due comportamenti sono completamente separati, promuovendo una chiara divisione tra i ruoli di ciascun trait. Questo pattern facilita anche l'espansione del sistema. Se un altro tipo, ad esempio *Animale*, deve implementare *Serializzabile* ma non *Descrivibile*, è semplice farlo mantenendo la coerenza del design.

I pattern di progettazione con i trait permettono di gestire varianti di comportamento senza dover duplicare il codice o dipendere da gerarchie complesse. In Rust, una soluzione comune per gestire comportamenti differenti è usare trait per descrivere diverse capacità o ruoli che un tipo può assumere, permettendo al compilatore di ottimizzare le varie implementazioni attraverso la monomorfizzazione.

| Trait | Descrizione | Esempio Pratico |
|---|---|---|
| `Copy` | *Consente di copiare valori in modo triviale, senza trasferire la proprietà del dato.* | `let a = 5;`<br>`let b = a; // Copia triviale del valore 'a', senza trasferimento di proprietà.` |
| `Clone` | *Permette la creazione di una copia profonda di un valore, quando Copy non è applicabile.* | `let s = String::from("ciao");`<br>`let s2 = s.clone(); // Crea una copia profonda di 's'.` |
| `Debug` | *Permette di formattare e stampare un valore per il debug.* | `let p = Persona { nome: String::from("Mario"), eta: 30 };`<br>`println!("{:?}", p); // Stampa il valore per il debug.` |
| `Deref` | *Consente di dereferenziare un puntatore smart, come Box, per comportarsi come un riferimento normale.* | `let x = Box::new(5);`<br>`let y = *x; // Dereferenzia il puntatore smart 'Box'.` |

| | | |
|---|---|---|
| `Add` | *Permette la sovraccarica dell'operatore + per i tipi che implementano questo trait.* | `use std::ops::Add;`<br>`struct Punto { x: i32, y: i32 }`<br>`impl Add for Punto {`<br>`type Output = Punto;`<br>`fn add(self, altro: Punto) -> Punto {`<br>`Punto { x: self.x + altro.x, y: self.y + altro.y`<br>`} } }` |
| `Iterator` | *Definisce il comportamento per gli oggetti iterabili, permettendo di scorrere gli elementi di una collezione.* | `let v = vec![1, 2, 3];`<br>`let mut iter = v.iter();`<br>`while let Some(val) = iter.next() {`<br>`println!("{}", val);`<br>`}` |
| `Drop` | *Consente di eseguire del codice specifico quando un valore esce dallo scope, utile per il rilascio delle risorse.* | `struct Risorsa;`<br>`impl Drop for Risorsa {`<br>`fn drop(&mut self) {`<br>`println!("Risorsa rilasciata!");}}`<br>`let r = Risorsa; // Alla fine dello scope, viene chiamato 'drop'.` |
| `PartialEq` | *Permette la comparazione parziale di valori di un tipo per uguaglianza (==) e diversità (!=).* | `#[derive(PartialEq)]`<br>`struct Punto { x: i32, y: i32 }`<br>`let p1 = Punto { x: 1, y: 2 };`<br>`let p2 = Punto { x: 1, y: 2 };`<br>`assert!(p1 == p2);` |
| `PartialOrd` | *Definisce la relazione di ordinamento parziale tra valori di un tipo, permettendo l'uso di <, >, <=, >=.* | `#[derive(PartialOrd)]`<br>`struct Punto { x: i32, y: i32 }`<br>`let p1 = Punto { x: 1, y: 2 };`<br>`let p2 = Punto { x: 2, y: 3 };`<br>`assert!(p1 < p2);` |

### *Lifetimes, borrowing e trait*

I lifetime sono una parte essenziale del sistema di gestione della memoria e servono a garantire che i riferimenti siano sempre validi. Rappresentano la durata di validità di un riferimento, e il compilatore di Rust utilizza i lifetime per prevenire errori legati alla gestione della memoria, come i *dangling pointer* (puntatori a memoria non valida). Quando si usano i trait in combinazione con riferimenti, i lifetime diventano fondamentali per assicurare che i riferimenti all'interno delle struct e dei metodi rimangano validi per tutta la durata dell'uso.

Introducendo i lifetime nel contesto di una struct come *Persona*, possiamo pensare a una situazione in cui uno o più campi della struct contengano riferimenti anziché valori posseduti. In questo caso, è necessario specificare esplicitamente i lifetime per assicurare che i riferimenti all'interno della struct siano ovalidi per la durata del loro utilizzo:

```
struct Persona<'a> {
    nome: &'a str,
    eta: u32,
}

impl<'a> Descrivibile for Persona<'a> {
    fn descrivi(&self) -> String {
        format!("Nome: {}, Età: {}", self.nome, self.eta)
    }
}
```

Qui, la struct *Persona* contiene un riferimento al campo nome, che ha un lifetime 'a. Questo assicura che il riferimento *&'a str* rimanga valido almeno per la durata del lifetime 'a. Quando si implementa il trait *Descrivibile* per *Persona,* il lifetime 'a viene propagato anche nell'implementazione del trait, poiché i suoi metodi devono garantire che i riferimenti siano validi per la durata specificata dal lifetime.
L'uso dei lifetime nei metodi di trait diventa cruciale quando i metodi operano su riferimenti. Rust richiede che ogni riferimento utilizzato all'interno di un metodo sia annotato con il suo lifetime, per garantire la sicurezza durante il borrowing. Supponiamo di voler modificare il metodo *descrivi* per restituire una stringa che fa riferimento ai dati interni della struct. In tal caso, è necessario specificare chiaramente i lifetime:

```
impl<'a> Descrivibile for Persona<'a> {
    fn descrivi(&self) -> &'a str {
        self.nome
    }
}
```

Di conseguenza, il metodo *descrivi* restituisce un riferimento a *self.nome*, e il lifetime di questo riferimento è legato al lifetime 'a della struct *Persona*. Questo assicura che il valore restituito rimanga valido finché *Persona* è valida.
Le restrizioni sull'uso dei trait con il borrowing e l'ownership derivano dalla necessità di rispettare le regole del *borrow checker*. Quando un trait viene implementato per una struct che contiene riferimenti, bisogna sempre considerare

le regole di mutabilità e borrowing di Rust. Ad esempio, un metodo che prende un riferimento mutabile (*&mut self*), secondo le regole deve garantire che nessun altro riferimento immutabile o mutabile esista contemporaneamente. Inoltre, se un trait prevede la mutazione di dati, come nel caso di *fn descrivi_mut(&mut self)*, l'implementazione deve rispettare le regole di borrowing:

```
impl<'a> Descrivibile for Persona<'a> {
    fn descrivi(&mut self) -> &'a str {
        self.nome = "Anonimo"; // Modifica del nome
        self.nome
    }
}
```

In questo caso, descrivi prende un riferimento mutabile a *self*, il che significa che durante l'esecuzione del metodo non possono esistere altri riferimenti a esso. La gestione dell'ownership e del borrowing attraverso i lifetime garantisce la sicurezza dell'accesso concorrente ai dati.
Un altro aspetto interessante riguarda i trait associati ai lifetime, dove i vincoli e le regole di sicurezza sono legati alla durata di vita dei riferimenti utilizzati. Quando un trait ha bisogno di farne uso, bisogna specificare chiaramente i lifetime per evitare situazioni dove i dati potrebbero diventare invalidi. Ad esempio, se si volesse creare un trait che opera su un riferimento, il suo lifetime deve essere vincolato a quelli dei riferimenti passati:

```
trait Salvataggio<'a> {
    fn salva(&self, database: &'a str);
}

impl<'a> Salvataggio<'a> per Persona<'a> {
    fn salva(&self, database: &'a str) {
        println!("Salvataggio su {} per {}", database, self.nome);
    }
}
```

Come dimostrato, il trait *Salvataggio* è vincolato al lifetime *'a*, che rappresenta la

durata di validità del riferimento database. Quando implementiamo *Salvataggio* per *Persona*, dobbiamo assicurarci che il lifetime di database coincida con quello della struct, garantendo che i riferimenti siano validi per tutta la durata dell'operazione di salvataggio.
Infine, è importante notare che Rust impone vincoli rigorosi sui lifetime per evitare che un riferimento possa mai diventare invalido. Questo sistema di sicurezza assicura che non ci siano mai accessi a memoria non valida, fornendo una robusta garanzia di sicurezza a compile-time.

### *Trait automatici*

I trait automatici giocano un ruolo importante nel rendere più fluida e sicura la conversione tra tipi e nel semplificare la manipolazione dei riferimenti. Alcuni trait predefiniti come *Deref, AsRef* e *Into* sono comunemente usati per la conversione implicita ed esplicita tra tipi, migliorando l'ergonomia del linguaggio senza compromettere la sicurezza.
Il trait *Deref* permette di trattare un tipo come se fosse un riferimento a un altro tipo. Questo è utile, ad esempio, quando si vuole che un oggetto possa essere dereferenziato in modo implicito per accedere ai suoi campi o metodi. Se prendiamo la struct *Persona* e ne creiamo una nuova che la incapsula, possiamo usare *Deref* per dereferenziare automaticamente l'oggetto incapsulato:

```
use std::ops::Deref;

struct PersonaContainer<'a> {
    persona: Persona<'a>,
}

impl<'a> Deref per PersonaContainer<'a> {
    type Target = Persona<'a>;

    fn deref(&self) -> &Self::Target {
        &self.persona
    }
}
```

Quindi, *PersonaContainer* contiene una *Persona,* e implementando *Deref* possiamo trattare un'istanza di *PersonaContainer* come se fosse direttamente una *Persona.* Ciò consente di accedere ai metodi della struct *Persona* senza dover esplicitamente estrarre il valore incapsulato, rendendo l'interazione con il tipo più naturale.
Il trait *AsRef* è usato per ottenere un riferimento a un tipo diverso, purché la conversione sia sicura e non perda informazioni. È particolarmente utile quando si ha un metodo che può accettare più tipi, ma si desidera lavorare con uno specifico. Se volessimo permettere a *PersonaContainer* di essere passato come riferimento a *Persona*, potremmo quindi implementare *AsRef*:

```
impl<'a> AsRef<Persona<'a>> for PersonaContainer<'a> {
    fn as_ref(&self) -> &Persona<'a> {
        &self.persona
    }
}
```

Questo rende possibile l'uso di *PersonaContainer* in funzioni che si aspettano un riferimento a *Persona*, migliorando la flessibilità del codice senza sacrificare la sicurezza del tipo.
Il trait *Into* permette una conversione esplicita da un tipo a un altro. Quando lo implementiamo, stiamo dichiarando che il nostro tipo può essere convertito in un altro in modo esplicito attraverso il metodo *into*. Perciò, se volessimo convertire un *PersonaContainer* in una *Persona*, faremmo così:

```
impl<'a> Into<Persona<'a>> for PersonaContainer<'a> {
    fn into(self) -> Persona<'a> {
        self.persona
    }
}
```

Questo permette all'utente di convertire un oggetto *PersonaContainer* in *Persona* chiamando semplicemente *into(). La conversione è esplicita e sicura, e fornisce un meccanismo per gestire facilmente i tipi incapsulati.*

Il pattern di tipo *newtype* è utilizzato per creare nuovi tipi che ne incapsulano un altro. Questo pattern è utile quando si desidera aggiungere funzionalità o restrizioni a un tipo esistente senza modificarne l'originale. Per esempio, se volessimo creare un tipo che rappresenta un'età valida, potremmo usare il pattern *newtype* per assicurare che questa sia sempre positiva:

```
struct EtaPositiva(u32);

impl EtaPositiva {
    fn nuova(eta: u32) -> Option<Self> {
        if eta > 0 {
            Some(EtaPositiva(eta))
        } else {
            None
        }
    }
}
```

Infatti, *EtaPositiva* è un nuovo tipo che incapsula un u32, ma con l'aggiunta di una logica che assicura che il valore sia sempre positivo. Questo pattern è comune quando si vuole usare un tipo base con delle garanzie aggiuntive. Con *newtype*, si può anche implementare trait su tipi che normalmente non si potrebbero modificare, ad esempio:

```
impl Descrivibile for EtaPositiva {
    fn descrivi(&self) -> String {
        format!("Età: {}", self.0)
    }
}
```

Il pattern ci consente quindi anche di implementare *Descrivibile* per *EtaPositiva,* mantenendo la separazione tra la rappresentazione interna (un semplice u32) e il comportamento definito dal trait.
In poche parole, il meccanismo di coercizione implicita è limitato e rigoroso per mantenere la sicurezza del sistema di tipi. Tuttavia, può avvenire automaticamente in alcune situazioni, ad esempio con *Deref,* dove un tipo può

essere automaticamente convertito (dal compilatore) in un altro se implementa il trait appropriato.

### *Trait avanzati (argomento avanzato)*

I trait avanzati permettono di estendere ulteriormente il linguaggio con la definizione di comportamenti specifici, come l'overloading degli operatori o la creazione di iteratori personalizzati. Un esempio possibile è *Add*, che consente di sovraccaricare l'operatore +. Per farlo, Rust fornisce dei trait predefiniti nella libreria standard, come *Add* per la somma o *Sub* per la sottrazione. Se volessimo sovraccaricare l'operatore + per la nostra struct *Persona* in modo che due persone possano "essere sommate" attraverso la concatenazione dei loro nomi, possiamo implementare il trait *Add*:

```
use std::ops::Add;

impl<'a> Add for Persona<'a> {
    type Output = String;

    fn add(self, other: Persona<'a>) -> Self::Output {
        format!("{} {}", self.nome, other.nome)
    }
}
```

Implementiamo quindi il trait *Add* per *Persona*, dove l'operatore + concatena i nomi delle due persone. La funzione *add* definisce come combinare i due valori, e il tipo *Output* specifica che il risultato sarà una *String*.

Analogamente, possiamo sovraccaricare altri operatori come *Sub, Mul* o *Div* utilizzando i rispettivi trait della libreria standard, definendo il comportamento specifico per il nostro tipo. Questi sono spesso usati quando si vuole creare tipi matematici personalizzati o quando si desidera un comportamento specifico per operazioni algebriche.

Un altro ambito importante è l'implementazione degli iteratori. Questi sono definiti attraverso il trait *Iterator*, il quale produce una sequenza di elementi e consente di

iterare su collezioni o tipi personalizzati in modo sicuro ed efficiente, attraverso l'implementazione del metodo *next()*. Possiamo estendere *Persona* per creare un iteratore personalizzato che restituisca il nome lettera per lettera:

```
struct PersonaIterator<'a> {
    persona: &'a Persona<'a>,
    indice: usize,
}

impl<'a> Iterator for PersonaIterator<'a> {
    type Item = char;

    fn next(&mut self) -> Option<Self::Item> {
        let chars: Vec<char> = self.persona.nome.chars().collect();
        if self.indice < chars.len() {
            let result = chars[self.indice];
            self.indice += 1;
            Some(result)
        } else {
            None
        }
    }
}

impl<'a> Persona<'a> {
    fn iter(&'a self) -> PersonaIterator<'a> {
        PersonaIterator {
            persona: self,
            indice: 0,
        }
    }
}
```

*PersonaIterator* è un iteratore che scorre le lettere del nome della persona. Implementiamo il trait *Iterator*, dove il metodo *next* restituisce la lettera successiva o *None* quando l'iterazione è completa. La struct *Persona* contiene un metodo *iter* che restituisce un'istanza di *PersonaIterator*, permettendo di iterare sul nome di una persona lettera per lettera.

Un altro trait strettamente collegato agli iteratori è *IntoIterator,* che permette di

trasformare un tipo in un iteratore. Se implementassimo *IntoIterator* per *Persona*, potremmo iterare direttamente su una persona, invece di chiamare esplicitamente *iter*:

```
impl<'a> IntoIterator for &'a Persona<'a> {
    type Item = char;
    type IntoIter = PersonaIterator<'a>;

    fn into_iter(self) -> Self::IntoIter {
        self.iter()
    }
}
```

Così, possiamo usare una *persona* direttamente in un contesto iterativo come un ciclo *for*. Questo pattern è utile quando si desidera che un tipo personalizzato possa essere utilizzato come un oggetto iterabile senza richiedere una chiamata esplicita a un metodo specifico.
Proseguendo, passiamo a un concetto strettamente legato ai trait avanzati e agli iteratori, cioè le *closure* (chiusure). Parliamo di funzioni anonime che possono catturare variabili dall'ambiente in cui sono definite. Possono essere passate come parametri a funzioni e metodi, e sono spesso utilizzate in combinazione con iteratori. Le closure possono essere usate per definire comportamenti dinamici su tipi complessi come ad esempio *Persona*, dove possiamo applicare una trasformazione personalizzata al suo nome:

```
impl<'a> Persona<'a> {
    fn trasforma<F>(&self, f: F) -> String
    where
        F: Fn(&str) -> String,
    {
        f(self.nome)
    }
}

let p = Persona {
    nome: "Mario",
```

```
    eta: 30,
};
let nome_maiuscolo = p.trasforma(|n| n.to_uppercase());
```

Perciò, *trasforma* accetta una closure che prende in input una stringa e ne restituisce una trasformata. In questo caso, la chiusura converte il nome in maiuscolo. L'uso delle closure in combinazione con i trait come *Iterator* consente una programmazione flessibile e modulare, mantenendo il controllo rigoroso sulla sicurezza del tipo.

Questi pattern di implementazione di trait per operatori sovraccaricati, iteratori e closure rappresentano un potente insieme di strumenti che Rust mette a disposizione per la creazione di codice sicuro, flessibile e altamente espressivo.

Non è tutto qui, i trait avanzati come *Fn, FnMut* e *FnOnce* consentono di rappresentare le closure in modo più preciso, basandosi sul tipo di borrowing che ciascuna di esse effettua sull'ambiente circostante. Questi trait permettono di modellarle in base a come catturano e manipolano le variabili esterne, garantendo il corretto comportamento rispetto alle regole di borrowing e ownership.

Il trait *Fn* rappresenta le closure che *non modificano lo stato catturato*. Questo significa che possiamo chiamarla ripetutamente senza cambiare il suo stato interno e possiamo condividerla tra più contesti grazie al borrowing immutabile. Se volessimo definire un metodo che accetta una closure che non muta lo stato di *Persona*, potremmo farlo in questo modo:

```
impl<'a> Persona<'a> {
    fn esegui_fn<F>(&self, f: F) -> String
    where
        F: Fn(&str) -> String,
    {
        f(self.nome)
    }
}

let p = Persona {
    nome: "Mario",
```

```
    eta: 30,
};

let risultato = p.esegui_fn(|nome| format!("Ciao, {}", nome));
```

Qui, il metodo *esegui_fn* accetta una closure che implementa il trait *Fn*, ovvero una closure che può essere eseguita più volte e che cattura in modo immutabile l'ambiente circostante.
Il trait *FnMut* rappresenta closure che mutano il loro stato interno o l'ambiente circostante attraverso un borrowing mutabile. Questo permette di modificare le variabili che ha catturato. Quindi, se volessimo permettere a una closure di cambiare lo stato della struct *Persona*, potremmo implementare un metodo che accetta una closure *FnMut*:

```
impl<'a> Persona<'a> {
    fn esegui_fn_mut<F>(&mut self, mut f: F)
    where
        F: FnMut(&mut String),
    {
        f(&mut self.nome);
    }
}

let mut p = Persona {
    nome: "Mario",
    eta: 30,
};

p.esegui_fn_mut(|nome| nome.push_str(" Rossi"));
```

Qui, *esegui_fn_mut* accetta una closure che modifica il nome della persona. Per farlo implementa il trait *FnMut* che può quindi modificare le variabili catturate in mutabilità, in questo caso aggiungendo un cognome a Mario.
Un'altra possibilità è data dal trait *FnOnce,* il quale rappresenta chiusure che possono essere chiamate una sola volta perché consumano l'ambiente catturato. Questo avviene quando la closure prende la proprietà dei valori catturati, il che

significa che dopo l'esecuzione non può essere richiamata. Ad esempio, se volessimo trasferire definitivamente il nome di una *Persona* all'interno di una closure, useremmo *FnOnce*:

```
impl<'a> Persona<'a> {
    fn esegui_fn_once<F>(self, f: F) -> String
    where
        F: FnOnce(String) -> String,
    {
        f(self.nome)
    }
}

let p = Persona {
    nome: "Mario",
    eta: 30,
};

let nuovo_nome = p.esegui_fn_once(|nome| nome.to_uppercase());
```

In questo caso, la chiusura consuma il nome di *Persona* (lo sposta al suo interno), quindi può essere chiamata solo una volta. Il trait *FnOnce* è utile per scenari in cui è necessario trasferire la proprietà dei dati alla closure.

Per quanto riguarda l'uso di trait iterativi e la loro combinazione con le strutture dati generiche, Rust offre potenti strumenti per creare algoritmi flessibili e sicuri. Un esempio comune è la creazione di iteratori generici basati su *Iterator* o *IntoIterator.* In questo contesto, i trait permettono di costruire algoritmi che operano su una vasta gamma di tipi di dati, garantendo al contempo la massima riusabilità del codice.

Un pattern comune è la creazione di un metodo generico che accetta un iteratore e applica una funzione su ogni elemento. Possiamo espandere *Persona* per avere una funzione che accetti un iteratore e applichi una trasformazione al suo nome:

```
impl<'a> Persona<'a> {
    fn trasforma_iter<I, F>(&self, mut iter: I, f: F) -> String
    where
```

```
        I: Iterator<Item = char>,
        F: Fn(char) -> char,
    {
        iter.map(f).collect()
    }
}

let p = Persona {
    nome: "Mario",
    eta: 30,
};

let iteratore = p.nome.chars();
let nome_trasformato = p.trasforma_iter(iteratore, |c| c.to_uppercase().next().unwrap());
```

In pratica, *trasforma_iter* prende un iteratore su caratteri (prodotto dal nome della persona) e applica una funzione di trasformazione (f) a ciascun carattere, trasformandoli in maiuscolo. Questo esempio dimostra come gli iteratori possano essere combinati con i trait per creare funzioni generiche che lavorano su tipi iterabili.

| Trait | Descrizione | Esempio Pratico |
|---|---|---|
| Sub | *Consente la sovraccarica dell'operatore - per i tipi che implementano questo trait.* | `use std::ops::Sub;`<br>`struct Punto { x: i32, y: i32 }`<br>`impl Sub for Punto {`<br>`type Output = Punto;`<br>`fn sub(self, altro: Punto) -> Punto {`<br>`Punto { x: self.x - altro.x, y: self.y - altro.y }`<br>`}`<br>`}` |
| Mul | *Permette la sovraccarica dell'operatore * per i tipi che implementano questo trait.* | `use std::ops::Mul;`<br>`struct Punto { x: i32, y: i32 }`<br>`impl Mul for Punto {`<br>`type Output = Punto;`<br>`fn mul(self, altro: Punto) -> Punto {`<br>`Punto { x: self.x * altro.x, y: self.y * altro.y }`<br>`}`<br>`}` |
| Div | *Consente la sovraccarica dell'operatore / per i tipi che implementano questo trait.* | `use std::ops::Div;`<br>`struct Punto { x: i32, y: i32 }`<br>`impl Div for Punto {`<br>`type Output = Punto;`<br>`fn div(self, altro: Punto) -> Punto {`<br>`Punto { x: self.x / altro.x, y: self.y / altro.y }`<br>`}`<br>`}` |
| Rem | *Permette la sovraccarica dell'operatore % per i tipi che implementano questo trait.* | `use std::ops::Rem;`<br>`struct Punto { x: i32, y: i32 }`<br>`impl Rem for Punto {`<br>`type Output = Punto;`<br>`fn rem(self, altro: Punto) -> Punto {`<br>`Punto { x: self.x % altro.x, y: self.y % altro.y }`<br>`}`<br>`}` |
| Neg | *Consente la sovraccarica dell'operatore - unario per i tipi che* | `use std::ops::Neg;`<br>`struct Punto { x: i32, y: i32 }`<br>`impl Neg for Punto {` |

| | | |
|---|---|---|
| | *implementano questo trait.* | `type Output = Punto;`<br>`fn neg(self) -> Punto {`<br>`Punto { x: -self.x, y: -self.y }`<br>`}`<br>`}` |
| `AddAssign` | *Permette la sovraccarica dell'operatore += per i tipi che implementano questo trait.* | `use std::ops::AddAssign;`<br>`struct Punto { x: i32, y: i32 }`<br>`impl AddAssign for Punto {`<br>`fn add_assign(&mut self, altro: Punto) {`<br>`self.x += altro.x;`<br>`self.y += altro.y;`<br>`}`<br>`}` |
| `SubAssign` | *Consente la sovraccarica dell'operatore -= per i tipi che implementano questo trait.* | `use std::ops::SubAssign;`<br>`struct Punto { x: i32, y: i32 }`<br>`impl SubAssign for Punto {`<br>`fn sub_assign(&mut self, altro: Punto) {`<br>`self.x -= altro.x;`<br>`self.y -= altro.y;`<br>`}`<br>`}` |
| `MulAssign` | *Permette la sovraccarica dell'operatore *= per i tipi che implementano questo trait.* | `use std::ops::MulAssign;`<br>`struct Punto { x: i32, y: i32 }`<br>`impl MulAssign for Punto {`<br>`fn mul_assign(&mut self, altro: Punto) {`<br>`self.x *= altro.x;`<br>`self.y *= altro.y;`<br>`}`<br>`}` |

### *Trait nelle librerie standard e nella programmazione asincrona*

I trait nella libreria standard di Rust giocano un ruolo cruciale anche nella programmazione concorrente e asincrona. Il linguaggio fornisce numerosi strumenti attraverso questi trait per gestire task concorrenti e operazioni asincrone, mantenendo al contempo i principi di sicurezza dell'ownership e del borrowing.

Nella programmazione asincrona, un trait fondamentale è *Future*, il quale rappresenta un valore che potrebbe non essere ancora disponibile, ma lo sarà in futuro. Questo tipo di astrazione permette di gestire operazioni asincrone in modo non bloccante. Ad esempio, quando implementiamo il supporto asincrono per una funzione che riguarda *Persona*, possiamo utilizzare il sistema di *async/await*, che si basa sul trait *Future*. Potremo farlo per creare una funzione asincrona che simuli il caricamento del nome di una *Persona* da una fonte esterna, utilizzando *Future* in combinazione con *async/await*:

```
use std::time::Duration;
use tokio::time::sleep;

impl<'a> Persona<'a> {
    async fn carica_nome(&self) -> String {
```

```
        sleep(Duration::from_secs(2)).await;
        format!("Nome caricato: {}", self.nome)
    }
}

#[tokio::main]
async fn main() {
    let p = Persona {
        nome: "Mario",
        eta: 30,
    };

    let nome = p.carica_nome().await;
    println!("{}", nome);
}
```

Ed ecco che il metodo *carica_nome* simula un'operazione asincrona utilizzando *sleep* per ritardare l'esecuzione di due secondi. L'uso di *async/await* consente di gestire l'operazione in modo non bloccante. Grazie al trait *Future*, il compilatore può trasformare le chiamate asincrone in *state machine* efficienti, minimizzando l'overhead.

Oltre a *Future*, un altro trait importante in questo ambito è *Stream*. Mentre il primo rappresenta un singolo valore che sarà disponibile in futuro, *Stream* rappresenta una sequenza di valori che verranno prodotti nel tempo. Potremmo implementare un flusso di aggiornamenti per la nostra struct *Persona*, dove vengono generati nuovi valori periodicamente:

```
use tokio_stream::{Stream, StreamExt};
use tokio::time::{sleep, interval};
use std::pin::Pin;
use std::task::{Context, Poll};

struct AggiornamentiPersona<'a> {
    persona: &'a Persona<'a>,
    count: usize,
    interval: tokio::time::Interval,
}
```

```
impl<'a> Stream for AggiornamentiPersona<'a> {
    type Item = String;

    fn poll_next(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Option<Self::Item>> {
        match self.interval.poll_tick(cx) {
            Poll::Ready(_) => {
                self.count += 1;
                Poll::Ready(Some(format!("Aggiornamento {}: {}", self.count,
self.persona.nome)))
            }
            Poll::Pending => Poll::Pending,
        }
    }
}

#[tokio::main]
async fn main() {
    let p = Persona {
        nome: "Mario",
        eta: 30,
    };

    let mut stream = AggiornamentiPersona {
        persona: &p,
        count: 0,
        interval: interval(Duration::from_secs(1)),
    };

    while let Some(aggiornamento) = stream.next().await {
        println!("{}", aggiornamento);
    }
}
```

In questo codice, *AggiornamentiPersona* implementa il trait *Stream*, producendo aggiornamenti periodici per *Persona.* L'uso del modulo *tokio::time::interval* ci permette di produrre aggiornamenti a intervalli regolari. La funzione *poll_next* gestisce il ciclo di aggiornamenti.

I trait sono anche fondamentali nella gestione dei thread e nella sincronizzazione. Rust fornisce una serie di strumenti che permettono di sfruttare la concorrenza in modo sicuro e privo di data race, grazie a trait come *Send* e *Sync*. Questi regolano

il comportamento di un tipo in un contesto multithread, assicurando che i dati possano essere trasferiti in sicurezza. Il trait *Send* indica che una struttura può essere inviata tra thread, mentre *Sync* garantisce che una struttura possa essere condivisa tra thread in modo sicuro.
Proviamo ad eseguire un'operazione concorrente su *Persona* sfruttando il multi-threading con *Send*:

```
use std::thread;

impl<'a> Persona<'a> {
    fn esegui_in_thread(self) {
        thread::spawn(move || {
            println!("Esecuzione nel thread separato per {}", self.nome);
        })
        .join()
        .unwrap();
    }
}

fn main() {
    let p = Persona {
        nome: "Mario",
        eta: 30,
    };

    p.esegui_in_thread();
}
```

In questo esempio, *esegui_in_thread* sposta la proprietà di *Persona* in un thread separato usando *thread::spawn*. Questo funziona perché *Persona* soddisfa il trait *Send,* quindi può essere trasferito in un altro thread.
Rust fornisce anche meccanismi di sincronizzazione sicura, come *Mutex* e *RwLock*, che utilizzano i trait *Send* e *Sync* per garantire che l'accesso concorrente a una risorsa condivisa avvenga in modo sicuro. Se volessimo sincronizzare l'accesso a *Persona* tra più thread, potremmo utilizzare un *Mutex*:

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

```
use std::thread;

fn main() {
    let persona = Arc::new(Mutex::new(Persona {
        nome: "Mario",
        eta: 30,
    }));

    let p1 = Arc::clone(&persona);
    let thread_1 = thread::spawn(move || {
        let mut persona = p1.lock().unwrap();
        persona.nome.push_str(" Rossi");
    });

    let p2 = Arc::clone(&persona);
    let thread_2 = thread::spawn(move || {
        let persona = p2.lock().unwrap();
        println!("Nome nel thread 2: {}", persona.nome);
    });

    thread_1.join().unwrap();
    thread_2.join().unwrap();
}
```

In questo caso, usiamo *Arc* e *Mutex* per consentire a più thread di accedere e modificare *Persona* in modo sicuro. Il trait *Sync* permette a *Mutex* di essere condiviso tra più thread, garantendo che non vi siano accessi simultanei non sincronizzati che potrebbero portare a race condition. Ne parleremo nel prossimo capitolo.

### *Pattern e anti-pattern nell'uso di trait e struct*

Nell'uso di trait e struct, è importante seguire alcuni pattern per garantire la leggibilità, la manutenibilità e l'efficienza del codice. I trait offrono un modo potente per esprimere il comportamento comune tra tipi diversi, mentre le struct consentono di organizzare i dati in modo chiaro e strutturato. Tuttavia, è facile cadere in alcune trappole comuni, che possono portare a una progettazione non ottimale e a errori difficili da gestire.

Un pattern fondamentale nell'uso di trait è quello di evitare l'eccessiva generalizzazione. Definire trait troppo ampi o vaghi, con troppi metodi o con metodi che non sono strettamente correlati, può portare a un codice difficile da mantenere e da estendere. I trait dovrebbero essere concisi e rappresentare comportamenti specifici. Ad esempio, invece di creare un singolo trait che definisca un insieme generico di comportamenti per *Persona*, è meglio creare più trait specializzati che modellano comportamenti distinti:

```
trait Descrivibile {
    fn descrizione(&self) -> String;
}

trait Aggiornabile {
    fn aggiorna_nome(&mut self, nuovo_nome: &str);
}

impl<'a> Descrivibile for Persona<'a> {
    fn descrizione(&self) -> String {
        format!("{} ha {} anni", self.nome, self.eta)
    }
}

impl<'a> Aggiornabile for Persona<'a> {
    fn aggiorna_nome(&mut self, nuovo_nome: &str) {
        self.nome = nuovo_nome;
    }
}
```

Essenzialmente, *Descrivibile* e *Aggiornabile* sono trait distinti, ognuno dei quali si occupa di un singolo aspetto del comportamento di *Persona*. Questo favorisce la separazione delle preoccupazioni, facilitando l'estensione o la modifica del comportamento senza toccare codice non correlato.

Un altro pattern utile è quello di utilizzare correttamente l'ownership e il borrowing nei metodi di trait e struct. Rust impone regole rigorose sull'ownership, e progettare metodi che rispettino queste regole migliora la manutenibilità del codice. Nei metodi che non devono modificare lo stato interno di una struct,

dovremo utilizzare il borrowing immutabile per evitare modifiche accidentali:

```
impl<'a> Persona<'a> {
    fn visualizza(&self) {
        println!("{} ha {} anni", self.nome, self.eta);
    }
}
```

D'altra parte, quando un metodo deve modificare lo stato di una struct, useremo un borrowing mutabile (*&mut self*) per rendere esplicita l'intenzione. Questo previene errori di concorrenza e facilita la lettura del codice.
Per quanto riguarda i trait, uno degli anti-pattern più comuni è l'implementazione superflua di trait predefiniti. Se un comportamento comune come la clonazione o il confronto di strutture può essere derivato automaticamente, è preferibile utilizzare l'annotazione *#[derive]*, piuttosto che implementare manualmente il trait, a meno che non ci sia una ragione specifica per farlo:

```
#[derive(Clone, Debug, PartialEq)]
struct Persona<'a> {
    nome: &'a str,
    eta: u32,
}
```

In questo caso, *Persona* eredita automaticamente l'implementazione dei trait *Clone, Debug* e *PartialEq*, rendendo il codice più conciso.
Un altro pattern importante riguarda l'uso dei trait per promuovere la modularità e la separazione delle preoccupazioni. Quando un'applicazione cresce in complessità, è essenziale mantenere il codice organizzato in moduli che incapsulino il comportamento in modo chiaro e coerente. *I trait possono essere utilizzati per definire interfacce che separano le responsabilità tra moduli diversi*.
Ad esempio, se stessimo costruendo un sistema per gestire diverse entità in una base di dati, potremmo definire un trait *Persistente* che si occupa esclusivamente della logica di salvataggio e caricamento di una *Persona* da un database:

```
trait Persistente {
    fn salva(&self);
    fn carica(id: u32) -> Self;
}

impl<'a> Persistente for Persona<'a> {
    fn salva(&self) {
        println!("Salvataggio di {} nel database", self.nome);
    }

    fn carica(_id: u32) -> Self {
        // Simulazione caricamento da database
        Persona {
            nome: "Mario",
            eta: 30,
        }
    }
}
```

Questo pattern promuove la modularità del codice e permette di estendere o cambiare il comportamento legato alla persistenza senza toccare altre parti del sistema.

Da considerare che Rust permette di esprimere concetti senza necessariamente legarli a dati concreti, mantenendo coerenza con il sistema dei tipi con le *unit-like struct*, una variante delle struct che non contengono alcun dato al loro interno. Si definiscono in modo simile a una tradizionale, ma senza specificarne i campi. Si comportano quindi come tipi che esistono solo per indicare un concetto o un'identità, senza portare con sé informazioni:

```
struct Punto;
```

In questo caso, la struct *Punto* non ha campi, ma può essere utilizzata per indicare l'esistenza di un'entità o un tipo. Sebbene non ci siano dati associati, questa struct può ancora implementare trait, essere usata per il pattern matching, o come marker per tipi specifici. Ad esempio, potremmo implementare un trait su una *unit-like struct* per definirne il comportamento:

```
impl Punto {
    fn descrivi(&self) -> &'static str {
        "Questo è un punto."
    }
}
```

Pur non avendo dati, la struct Punto è in grado di fornire metodi e comportamenti. Le *unit-like struct* sono spesso usate come "marker types" o "flag types", per indicare proprietà o capacità di altri tipi nel sistema. Possono essere utili anche per modellare concetti che non richiedono una rappresentazione di dati, ma servono a conferire un'identità specifica.

Nel contesto dell'esempio precedente, possiamo utilizzare una *unit-like struct* per rappresentare un tipo che funge da marker o categorizzatore per il concetto di entità persistente. L'idea è di separare ulteriormente il concetto di "persistenza" dalla logica della struct *Persona*, rendendo più modulare il design. Ad esempio, potremmo creare una *unit-like struct* chiamata *Persistenza*, che agisca come marker per indicare che la struct può essere salvata e caricata dal database:

```
struct Persistenza;

trait Persistente {
    fn salva(&self);
    fn carica(id: u32) -> Self;
}

impl<'a> Persistente for Persona<'a> {
    fn salva(&self) {
        println!("Salvataggio di {} nel database", self.nome);
    }

    fn carica(_id: u32) -> Self {
        // Simulazione caricamento da database
        Persona {
            nome: "Mario",
            eta: 30,
        }
    }
```

```
}

impl Persistenza {
    fn persistente<T: Persistente>(entita: &T) {
        entita.salva();
    }
}
```

In questo caso, *Persistenza* è una unit-like struct che rappresenta il concetto di gestione della persistenza, separato dalla struct *Persona* stessa. Grazie a questo pattern, possiamo chiamare il metodo *salva* o gestire la persistenza senza dover necessariamente legare tali operazioni alla logica interna di *Persona*. Questa separazione rafforza la modularità, poiché l'implementazione di Persistente può essere associata a qualunque struct, e il marker *Persistenza* può essere utilizzato per estendere ulteriormente il sistema con nuove entità persistenti o logiche di persistenza più avanzate.

In conclusione, un anti-pattern che dovrebbe essere evitato è l'*overengineering*, ovvero l'implementazione di trait e pattern complessi quando non necessario. Se un problema può essere risolto in modo semplice con una struct e qualche metodo associato, non è sempre necessario introdurre trait o generici complessi. Ad esempio, se *Persona* ha solo un tipo di comportamento specifico, non è sempre utile creare più trait solo per astrarre troppo il concetto.

Infine, per migliorare la leggibilità del codice, è una buona pratica utilizzare nomi significativi per i trait e le struct. Nomi come *Aggiornabile* o *Descrivibile* comunicano chiaramente il ruolo del trait nel sistema. Inoltre, dividere il codice in moduli ben organizzati e utilizzare test per assicurarsi che ogni parte del sistema funzioni correttamente sono strategie fondamentali per mantenere un codice manutenibile a lungo termine.

### *Marker Trait e Polimorfismo ad-hoc (argomento avanzato)*

I *marker trait* non definiscono alcun metodo, ma servono per contrassegnare o qualificare un tipo con una certa proprietà. Un esempio comune è il trait *Copy*, che

indica che un tipo può essere copiato in modo triviale, senza movimenti di ownership. Le *unit-like struct* sono strutture che non contengono dati e sono spesso utilizzate in combinazione con marker trait per rappresentare concetti astratti o proprietà. In questo contesto, potremmo voler contrassegnare la struct *Persona* con un *marker trait* che definisce una proprietà, senza introdurre comportamenti specifici:

```
trait Autenticabile {}

struct PersonaAutenticata;

impl Autenticabile for PersonaAutenticata {}
```

L'utilizzo di una *unit-like* struct come *PersonaAutenticata* in questo caso serve a definire una persona che è autenticata nel sistema, ma non introduce dati o comportamenti nuovi. Questa tecnica può essere utile per la meta-programmazione, per separare concettualmente le diverse versioni di *Persona*.
Invece, un pattern come il *Decorator* può essere implementato tramite trait per estendere il comportamento di una struct senza modificarne direttamente la definizione. Possiamo creare un trait che decori il comportamento di *Persona*, ad esempio aggiungendo funzionalità di logging:

```
trait Loggable {
    fn log_operazione(&self);
}

impl<'a> Loggable for Persona<'a> {
    fn log_operazione(&self) {
        println!("Operazione effettuata da {}", self.nome);
    }
}
```

In questo modo, *Persona* mantiene il suo comportamento originale, ma possiamo aggiungere nuove capacità senza modificarne la struttura di base. Il pattern *Visitor* può essere implementato tramite trait per separare operazioni da compiere su un

tipo specifico dalla definizione stessa del tipo. Nel caso di *Persona*, potremmo implementarlo per eseguire operazioni come il calcolo di statistiche, basandoci sulle informazioni della persona:

```
trait Visitabile {
    fn accetta<V: Visitor>(&self, visitatore: &V);
}

trait Visitor {
    fn visita_persona(&self, persona: &Persona);
}

impl<'a> Visitabile for Persona<'a> {
    fn accetta<V: Visitor>(&self, visitatore: &V) {
        visitatore.visita_persona(self);
    }
}
```

*Visitor* può essere esteso per aggiungere nuove operazioni su *Persona* senza modificarne direttamente il comportamento o l'implementazione.
Infine, parlando del polimorfismo ad-hoc con i trait object, possiamo usare il *dynamic dispatch* per permettere a *Persona* di lavorare con moduli intercambiabili a runtime. Se ad esempio vogliamo che implementi diverse versioni del salvataggio nel database, possiamo definire un trait che rappresenti questa azione e usare *Box<dyn Persistente>* per selezionare il comportamento a runtime:

```
trait Persistente {
    fn salva(&self);
}

struct DatabaseSQL;
struct DatabaseNoSQL;

impl Persistente for DatabaseSQL {
    fn salva(&self) {
        println!("Salvataggio in database SQL");
    }
}
```

```
impl Persistente for DatabaseNoSQL {
    fn salva(&self) {
        println!("Salvataggio in database NoSQL");
    }
}

fn esegui_salvataggio(db: Box<dyn Persistente>) {
    db.salva();
}
```

In questo modo, *Persona* può salvare i suoi dati in diversi tipi di database senza conoscere a priori quale verrà usato. L'uso di *static dispatch* con i generici, invece, fornisce una maggiore efficienza, ma richiede che tutti i tipi siano conosciuti a compile-time. Il *dynamic dispatch* ha il vantaggio della flessibilità a runtime, ma introduce un piccolo overhead in termini di performance.
Questi concetti si inseriscono naturalmente nel discorso sulla modularità e l'uso avanzato dei trait, mostrando come estendere e personalizzare il comportamento dei tipi in modo sicuro e performante.

### *Quiz & Esercizi*

1) Definire una struct *Libro* con titolo, autore e pagine, e un trait *Leggibile* con un metodo *leggi* che descrive il libro.
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).
3) Definire una struct *Punto3D* che estende *Punto2D* tramite un trait *Posizionabile* con un metodo *sposta* che cambia le coordinate di un punto.
4) Definire un trait *Disegnabile* per disegnare forme diverse (Cerchio, Rettangolo) e mostrare l'area e il perimetro.
5) Creare una struct *ContoCorrente* che implementa un trait *Rendiconto* con un metodo *mostra_saldo*.
6) Definire una struct *Dipendente* con nome e salario e un trait *Lavoratore* con un metodo *paga* che calcola il salario annuale.
7) Implementare un trait *Misurabile* per una struct *Cilindro* che calcola il volume e l'area della superficie.

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

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

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

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

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.

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

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

15) Scrivere un trait *Calcolabile* che somma di una lista di numeri e lo implementi come *generics*.

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

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

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

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

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

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

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

23) Implementare un trait *Somma* per sommare due struct *Punto* che contengono coordinate x e 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.

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

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

27) Creare una funzione asincrona che effettua una richiesta HTTP e gestisce il timeout utilizzando il

crate *tokio* e trait asincroni.
28) Definire un anti-pattern dove viene creata una struct con riferimenti non necessari e mostrare la sua ottimizzazione.
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.
30) Utilizzare lifetimes in combinazione con generics e trait per implementare un *wrapper* (una struct che avvolge un tipo di dato esistente o una risorsa, aggiungendo funzionalità o controlli extra. Viene utilizzato per fornire un'interfaccia sicura, astrarre dettagli di implementazione o implementare nuovi metodi per un tipo esistente. I *wrapper* sono particolarmente utili quando si vogliono applicare delle restrizioni o manipolazioni senza modificare il comportamento originale del tipo sottostante) che mantenga un riferimento a un valore di tipo generico e fornisca un metodo per restituire quel valore.

***Riassunto***

In questo capitolo abbiamo esplorato vari concetti fondamentali, concentrandoci principalmente su struct e trait e sulla loro interazione. Abbiamo discusso le struct come meccanismi per organizzare e gestire i dati in modo sicuro e strutturato, evidenziando i modificatori di visibilità che controllano l'accesso ai loro campi e la differenza tra struct immutabili e mutabili. Successivamente, abbiamo analizzato l'uso dei trait in Rust, che fungono da interfacce per definire comportamenti comuni e riutilizzabili. Attraverso i trait come *Clone, Debug, Copy* e *Drop*, Rust consente di implementare funzionalità predefinite o astratte che possono essere applicate a diverse struct, mantenendo una forte separazione tra dati e comportamento.
Abbiamo poi esplorato concetti più avanzati come il polimorfismo dinamico tramite i *trait object*, che permettono di gestire tipi eterogenei in modo flessibile, pur con qualche compromesso in termini di performance rispetto al dispatch statico. I generics hanno offerto un'altra prospettiva, dimostrando come si possa implementare codice generico utilizzando trait per vincolare i tipi a determinati comportamenti. Si è parlato anche di come i trait contribuiscano alla modularità e alla composizione del codice, evitando gerarchie rigide tipiche dell'ereditarietà dei linguaggi orientati agli oggetti.
Abbiamo analizzato il pattern di mutabilità, che è gestito in modo molto rigido e sicuro attraverso borrowing e ownership, con particolare attenzione all'uso dei lifetime nei trait per mantenere la sicurezza durante la gestione di riferimenti. Infine, abbiamo trattato trait avanzati come *Add, Sub, Mul,* e il loro utilizzo per la sovraccarica degli operatori, insieme a pattern come newtype e l'uso di trait automatici come *Deref* e *AsRef* per facilitare la conversione implicita tra tipi. Il riassunto tocca i principali aspetti legati alla definizione del comportamento attraverso i trait e alla gestione dei dati con le struct, ponendo l'accento sulla sicurezza, l'efficienza e la flessibilità del linguaggio.

# # 5 - Ownership e Borrowing

*Trasferimento e prestito*
*Smart Pointers*
*Ereditarietà*
*Altri principi OOP*

Il sistema di ownership e borrowing è uno dei pilastri fondamentali del linguaggio, garantendo una gestione della memoria sicura e senza la necessità di un meccanismo automatico di pulizia. La sua logica si basa su un insieme di regole statiche che il compilatore verifica al momento della compilazione, consentendo di prevenire errori come i *data race*, l'uso di memoria non inizializzata o la doppia liberazione della stessa. Questi concetti sono strettamente legati e lavorano insieme per gestire in modo sicuro l'accesso alla memoria e la durata dei dati.
L'ownership è il principio secondo cui o*gni valore ha un solo proprietario alla volta*. Quando una variabile assume la proprietà di un dato, essa è la responsabile esclusiva per la sua gestione e deallocazione. Questo implica che quando la variabile esce dallo scope, il dato viene automaticamente deallocato, seguendo una logica simile al RAII (*Resource Acquisition Is Initialization*) presente in linguaggi come C++. La proprietà viene trasferita (mossa) quando si passa una variabile a un'altra, invalidando l'uso della variabile originale. *Questa mossa è cruciale per impedire la presenza di due riferimenti mutabili alla stessa risorsa, che potrebbe portare a comportamenti indefiniti*.
Accanto all'ownership si trova il concetto di borrowing, che consente di prendere in prestito una risorsa temporaneamente senza trasferirne la proprietà. Questo

borrowing può essere effettuato in due modi: tramite il prestito immutabile o il prestito mutabile. Nel primo caso, più parti possono leggere contemporaneamente un dato senza modificarlo. Nel secondo, si permette a una sola parte di modificare il dato, garantendo così l'assenza di concorrenza tra lettori e scrittori. La regola fondamentale del borrowing è che non possono esserci prestiti mutabili e immutabili attivi sullo stesso dato contemporaneamente. Questa restrizione garantisce la coerenza della memoria e previene accessi concorrenti non sicuri.
Per gestire la durata dei prestiti, Rust introduce i *lifetimes,* che esprimono le relazioni temporali tra i riferimenti. Il compilatore è in grado di *inferire* la maggior parte di queste relazioni in modo implicito, ma in casi complessi può essere necessario specificarle esplicitamente. I *lifetimes* prevengono l'uso di riferimenti a memoria che non è più valida, come accade quando un dato viene deallocato perché il suo proprietario è uscito dallo scope.
La combinazione di ownership, borrowing e lifetimes consente a Rust di fornire un alto grado di sicurezza e performance, evitando sia il costo di un garbage collector che i rischi tipici della gestione manuale della memoria. Inoltre, l'approccio statico del linguaggio, basato sul controllo a tempo di compilazione, permette di rilevare gli errori prima dell'esecuzione del programma, contribuendo a migliorare l'affidabilità del codice.
Infine, questi concetti non sono solo applicabili alla gestione della memoria in senso stretto, ma si estendono a qualsiasi risorsa che necessiti di un controllo rigoroso del ciclo di vita, come file o socket, rendendo il modello di Rust particolarmente adatto per sistemi embedded e applicazioni ad alte prestazioni, dove la prevedibilità e la sicurezza sono prioritarie.

### *Ownership e trasferimenti complessi di proprietà*

Partiamo dal concetto di *move*, cruciale per comprendere come il linguaggio gestisce l'ownership e, di conseguenza, la memoria. Quando un valore viene "mosso" da una variabile a un'altra, la proprietà del dato viene trasferita,

invalidando la variabile originale. Questo comportamento è diverso da una semplice copia, *poiché in molti casi Rust preferisce spostare i dati piuttosto che duplicarli*. Ciò consente al compilatore di evitare copie inutili, migliorando l'efficienza e prevenendo condizioni di gara (race conditions) o doppi accessi a memoria non sicuri. Ad esempio:

```
let s1 = String::from("ciao");
let s2 = s1;
```

Ecco che il valore di *s1* viene trasferito a *s2*. Dopo il trasferimento, *s1* non è più utilizzabile; ogni tentativo di accedervi causerà un errore a tempo di compilazione. Questa operazione di *move* non è una semplice copia del contenuto della stringa: invece di duplicare i dati, Rust trasferisce direttamente la proprietà. Di conseguenza, non ci saranno duplicati in memoria, e quando *s2* esce dallo scope, sarà quest'ultima variabile a deallocare la memoria associata alla stringa.
In poche parole abbiamo appena dimostrato il *move implicito*, il quale avviene in molti contesti quando un dato che non implementa il *trait Copy* viene passato o assegnato a una nuova variabile. Specificatamente, i tipi semplici come gli interi e i valori di dimensione fissa sullo stack implementano il *trait Copy*, *il che significa che possono essere copiati invece di essere mossi*. Tuttavia, per tipi più complessi come *String* o *Vec<T>,* che gestiscono memoria allocata dinamicamente, il *move* è il comportamento predefinito. Questo meccanismo evita inefficienze, specialmente quando si manipolano grandi quantità di dati o strutture che non devono essere duplicate.
Ovviamente abbiamo anche il *move esplicito*, che si manifesta quando il programmatore intende trasferire esplicitamente la proprietà di un dato attraverso operazioni come il passaggio di una variabile a una funzione o il ritorno di una variabile da una funzione. Ad esempio:

```
fn prendi_ownership(s: String) {
    println!("{}", s);
}
```

```
let s1 = String::from("salve");
prendi_ownership(s1);
```

In questo caso, *s1* viene passato alla funzione *prendi_ownership*, e con esso anche la sua *ownership*. Dopo la chiamata alla funzione, *s1* non sarà più accessibile, perché la proprietà è stata trasferita alla funzione. Questo è un esempio di *move* esplicito, in cui la variabile originale perde il controllo del dato.
Esistono però situazioni in cui si desidera evitare un trasferimento di proprietà non desiderato. Una strategia comune è utilizzare un prestito (*borrowing*), *ovvero passare un riferimento a un dato invece di trasferirne la proprietà*. In tal caso, la proprietà rimane alla variabile originale, ma si concede a un'altra parte del programma di accedere temporaneamente al dato, in modalità immutabile o mutabile, senza invalidarlo:

```
fn prendi_prestito(s: &String) {
    println!("{}", s);
}

let s1 = String::from("buongiorno");
prendi_prestito(&s1); // s1 non viene mosso, solo preso in prestito
println!("{}", s1);    // s1 è ancora valido
```

In questo esempio, *s1* viene preso in prestito dalla funzione *prendi_prestito*. Poiché è solo un riferimento immutabile, *s1* rimane valido dopo la chiamata della funzione, e possiamo continuare a usarlo nel codice successivo.
Se invece il trasferimento di proprietà è inevitabile *ma non desiderato*, si può optare per una *clonazione esplicita del dato*. Il metodo *clone* ne crea una copia profonda, duplicandone il contenuto in memoria, e consente alla variabile originale di mantenere la propria ownership:

```
let s1 = String::from("ciao");
let s2 = s1.clone(); // Clonazione del dato
println!("{}", s1);  // s1 è ancora valido
```

In questo caso, il valore di *s1* non viene trasferito a *s2*, ma duplicato. Di conseguenza, entrambi restano validi, e quando escono dallo scope, ciascuno libererà la propria copia della stringa in modo indipendente.
L'uso di *clone* permette di mantenere l'ownership originale, ma va usato con cautela, poiché la clonazione ha un costo in termini di performance, specialmente quando si lavora con strutture dati di grandi dimensioni o complessi oggetti heap-allocati. D'altra parte, affidarsi esclusivamente a *move* può risultare problematico quando si desidera continuare a usare la variabile originale dopo un'operazione, portando così a un compromesso tra efficienza e flessibilità.

### *Prestiti (Borrowing) complessi*

Il concetto di prestito è strettamente collegato al sistema di ownership, permettendo di accedere a un dato senza trasferirne la proprietà. I prestiti possono essere immutabili o mutabili, con regole molto precise per garantire la sicurezza della memoria e l'assenza di condizioni di corsa (*data races*). La gestione dei prestiti multipli e della mutabilità segue un principio chiave: è possibile avere un numero indefinito di prestiti immutabili o un solo prestito mutabile alla volta, ma mai entrambi simultaneamente.
I *prestiti immutabili* permettono di accedere a un dato per la lettura, senza modificarlo. Più parti del programma possono prendere in prestito un riferimento immutabile allo stesso dato contemporaneamente. Questo tipo di accesso concorrente è sicuro perché nessuno dei prestiti può modificare il dato, evitando così problemi di inconsistenza:

```
let s = String::from("ciao");
let r1 = &s;
let r2 = &s;
println!("{} e {}", r1, r2);
```

Quindi, *r1* e *r2* prendono in prestito il dato immutabilmente e possono accedervi senza problemi, poiché non vi è alcuna modifica in corso. Entrambi i riferimenti

convivono senza conflitti, seguendo la regola che consente prestiti immutabili multipli.
I *prestiti mutabili*, al contrario, permettono la modifica del dato, ma con la restrizione che può esistere un solo prestito mutabile alla volta. Questa limitazione evita che più parti del programma possano accedere e modificare lo stesso dato contemporaneamente, prevenendo così condizioni di gara che potrebbero corrompere lo stato del programma:

```
let mut s = String::from("ciao");
let r1 = &mut s;
r1.push_str(", mondo");
println!("{}", r1);
```

In questo caso, *r1* prende in prestito mutabilmente la stringa *s*, e può modificarne il contenuto. Tuttavia, mentre esiste il prestito mutabile *r1*, non è possibile creare altri prestiti, né immutabili né mutabili, a *s*. Se si tentasse di prenderne un altro (immutabile o mutabile) durante il prestito mutabile, il compilatore genererebbe un errore, come nel seguente esempio:

```
let r2 = &s; // Errore: prestito immutabile mentre esiste un prestito mutabile
```

Questa regola rigida garantisce che non vi sia mai più di un mutatore attivo su un dato, evitando accessi concorrenti non sicuri. Tuttavia, una volta che il prestito mutabile non è più in uso, è possibile crearne altri.
Il concetto di prestiti temporanei entra in gioco quando i riferimenti vengono presi in prestito solo per una breve durata, come all'interno di una singola espressione. Rust è in grado di gestirli senza impattare l'intero ciclo di vita del dato:

```
let s = String::from("ciao");
let len = s.len();
println!("{}", s);
```

Abbiamo visto che *s.len()* prende un prestito temporaneo a *s* per determinare la lunghezza della stringa. Il prestito dura solo per il tempo della chiamata alla

funzione *len(),* dopo di che *s* è di nuovo completamente accessibile. Questo prestito temporaneo non impedisce l'uso successivo di *s* perché il riferimento è valido solo per la durata della singola espressione.
Un concetto strettamente legato ai prestiti è quello degli *alias di riferimenti*, dove più riferimenti puntano allo stesso dato. Ovviamente devono essere tutti immutabili. Se esistesse un prestito mutabile in presenza di alias immutabili, ci sarebbe il rischio che una modifica attraverso il prestito mutabile invalidi gli alias, *portando a comportamenti indefiniti*. Il linguaggio previene questo problema a livello di compilatore, bloccando l'esistenza simultanea di alias immutabili e prestiti mutabili.
Un'evoluzione significativa di Rust è rappresentata dal meccanismo di *Non-Lexical Lifetimes* (NLL), introdotto per migliorare la flessibilità nella gestione dei riferimenti. Prima dell'introduzione degli NLL, i riferimenti e i loro cicli di vita erano strettamente legati ai blocchi lessicali (cioè ai confini di scope definiti dal codice). Ciò significava che un prestito sarebbe stato considerato attivo fino alla fine del blocco, anche se non era più utilizzato.
Con NLL, il compilatore è diventato più intelligente nella gestione dei lifetimes, permettendo di determinare il termine effettivo di un prestito in base all'utilizzo reale, e non solo alla struttura lessicale del codice. Ad esempio, senza NLL, il seguente codice avrebbe generato un errore:

```
let mut s = String::from("ciao");
let r1 = &s;
println!("{}", r1);  // Uso di r1 termina qui
let r2 = &mut s;     // Prestito mutabile dopo il prestito immutabile
```

Prima di NLL, il compilatore avrebbe bloccato *r2* perché avrebbe considerato *r1* attivo fino alla fine dello scope, nonostante *r1* non fosse più utilizzato. Con NLL, il compilatore riconosce che *r1* non viene più usato dopo il *println!* e permette l'esistenza di *r2* immediatamente dopo.
Il meccanismo di NLL aumenta la flessibilità nel gestire i riferimenti in contesti

complessi, consentendo prestiti mutabili e immutabili in sequenza, purché non siano effettivamente attivi nello stesso momento. Questo permette di scrivere codice più conciso e naturale senza compromettere la sicurezza.
In sintesi, Rust gestisce i prestiti multipli e la mutabilità garantendo che possa esistere solo un prestito mutabile alla volta, oppure che vi siano multipli prestiti immutabili senza conflitti. I prestiti temporanei permettono di accedere ai dati solo per la durata di una singola espressione, facilitando la gestione dei cicli di vita senza inibire l'accesso successivo. Grazie al meccanismo di *Non-Lexical Lifetimes*, il compilatore è in grado di determinare con maggiore precisione quando i riferimenti smettono di essere utilizzati, migliorando la flessibilità nella gestione dei prestiti, senza sacrificare la sicurezza della memoria.

### *Pattern di borrowing in funzioni e strutture dati*

Il prestito all'interno di funzioni permette di passare riferimenti a dati senza trasferirne la proprietà, consentendo di mantenere l'ownership del dato originale e garantendo che esso rimanga utilizzabile dopo la chiamata della funzione. Quando una funzione prende in prestito un dato tramite un riferimento, sia esso mutabile o immutabile, non si altera la proprietà del dato. Questo consente di evitare duplicazioni o trasferimenti non necessari, ottimizzando l'efficienza del programma. Se passiamo una stringa in prestito immutabile a una funzione, ad esempio:

```
fn stampa_messaggio(messaggio: &String) {
    println!("{}", messaggio);
}

let saluto = String::from("ciao");
stampa_messaggio(&saluto);
println!("{}", saluto); // saluto è ancora accessibile
```

Otteniamo che la funzione *stampa_messaggio* prende in prestito la stringa *saluto* tramite un riferimento immutabile, il che significa che può leggere il dato ma non

modificarlo. Poiché la proprietà del dato non è trasferita, la variabile *saluto* rimane utilizzabile anche dopo l'esecuzione della funzione.

Nel caso del ritorno di riferimenti da una funzione, è importante considerare i *lifetimes*. Rust richiede che quelli dei riferimenti restituiti siano chiaramente definiti per garantire non siano mai invalidati. Se una funzione restituisce un riferimento, il suo ciclo di vita deve essere uguale o inferiore a quello del dato originale a cui fa riferimento. Ad esempio:

```
fn prendi_primo_elemento(vettore: &Vec<i32>) -> &i32 {
    &vettore[0]
}

let numeri = vec![1, 2, 3];
let primo = prendi_primo_elemento(&numeri);
println!("{}", primo);
```

Qui, la funzione *prendi_primo_elemento* restituisce un riferimento al primo elemento di *numeri*, garantendo che il ciclo di vita del riferimento sia legato a quello del vettore originale. Il compilatore gestisce questo tramite i *lifetimes* per evitare che il riferimento venga utilizzato oltre il tempo in cui il vettore è ancora valido.

Il borrowing all'interno di strutture dati complesse rappresenta una sfida particolare, soprattutto quando si devono gestire dati a lunga vita o puntatori interni. Rust evita gli alias mutabili per garantire la sicurezza della memoria, ma ci sono situazioni in cui è necessario mantenere riferimenti interni a una struttura dati per un periodo esteso senza trasferire la proprietà dell'intera struttura. Questo scenario è complesso perché i riferimenti interni devono rispettare i lifetimes definiti dal compilatore.

Ad esempio, supponiamo di voler creare una struttura che mantenga un riferimento a una stringa esterna:

```
struct Contenitore<'a> {
    contenuto: &'a String,
```

```
}

let stringa = String::from("contenuto");
let contenitore = Contenitore { contenuto: &stringa };
println!("{}", contenitore.contenuto);
```

Qui, il ciclo di vita del riferimento contenuto è legato a quello della stringa esterna, e *Rust garantisce che il riferimento sia valido solo finché lo è anche il dato originale*. Questa gestione rigorosa dei lifetimes è cruciale quando si trattano dati a lunga vita, poiché permette al compilatore di prevenire errori di memoria e accessi a dati non validi.

Tuttavia, nelle strutture dati più complesse, è spesso necessario manipolare dati mutabili e prestiti immutabili in modo più sofisticato. Perciò esistono strumenti come *RefCell* e *Rc* per gestire questa complessità. Ad esempio, il primo permette la mutabilità interna, consentendo di modificare dati intrinseci di una struttura *anche se il contenitore è immutabile*:

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

struct Contenitore {
    contenuto: RefCell<String>,
}

let contenitore = Contenitore {
    contenuto: RefCell::new(String::from("iniziale")),
};

contenitore.contenuto.borrow_mut().push_str(" aggiornato");
println!("{}", contenitore.contenuto.borrow());
```

Quindi *RefCell* gestisce i prestiti mutabili dinamicamente a runtime, anziché a compile-time, permettendo mutabilità in contesti dove normalmente non sarebbe possibile. Questo è particolarmente utile in strutture dati complesse dove la mutabilità deve essere regolata in modo flessibile.

Quando i trait interagiscono con borrowing e ownership, Rust utilizza il concetto di *trait bounds* per assicurarsi che i parametri soddisfino certi requisiti in termini di

ownership o borrowing. Un *trait bound* è una restrizione imposta sui tipi generici per garantire che essi implementino specifici trait. Ad esempio, si può richiedere che un parametro generico implementi il trait *Borrow* per poter prendere in prestito i suoi dati:

```
fn mostra_elemento<T: AsRef<str>>(elemento: T) {
    println!("{}", elemento.as_ref());
}

let stringa = String::from("ciao");
mostra_elemento(&stringa);
mostra_elemento(stringa);
```

La funzione *mostra_elemento* accetta un parametro generico *T* che deve implementare il trait *AsRef<str>.* Questo significa che la funzione può prendere in prestito un riferimento a una stringa (*&str*) o accettare una stringa completa. La flessibilità offerta dai *trait bounds* permette di gestire scenari complessi in cui ownership e borrowing si combinano.
Infine, i trait lavorano con ownership in modo molto specifico. Se si richiede che un tipo possieda i dati, la funzione che utilizza quel trait potrebbe trasferire la proprietà del dato. Se invece il trait si basa su un riferimento, la funzione lavorerà con prestiti senza trasferire l'ownership. Ecco un esempio che mostra come un trait possa influenzare il comportamento di borrowing:

```
trait Mostrabile {
    fn mostra(&self);
}

impl Mostrabile for String {
    fn mostra(&self) {
        println!("{}", self);
    }
}

let messaggio = String::from("ciao");
messaggio.mostra(); // Borrowing, non si trasferisce la proprietà
```

In questo caso, il trait *Mostrabile* richiede un riferimento immutabile (*&self*), il che significa che la chiamata al metodo mostra non trasferisce la proprietà del dato. La stringa messaggio viene semplicemente presa in prestito per la durata della chiamata, e può essere utilizzata nuovamente in seguito.

### *Smart Pointers*

Gli smart pointer sono strumenti avanzati che estendono il concetto di puntatori offrendo funzionalità aggiuntive, come la gestione automatica della memoria, il conteggio dei riferimenti, e la mutabilità interna. A differenza dei puntatori semplici, gestiscono automaticamente l'allocazione e la deallocazione della memoria, rispettando le regole di ownership e borrowing. Alcuni dei più comuni sono *Box<T>, Rc<T>, Arc<T>, Cell<T>* e *RefCell<T>,* ognuno con caratteristiche e utilizzi specifici.

Il *Box* è lo smart pointer più semplice, utilizzato per allocare un dato dinamicamente sullo heap anziché sullo stack. Quando utilizzi *Box<T>,* il tipo di dato viene spostato dallo stack all'heap, ma il modello di ownership e borrowing rimane lo stesso. La proprietà del dato è mantenuta da *Box<T>,* che garantisce l'unico accesso esclusivo al dato. Questo permette di utilizzare *Box* per implementare strutture ricorsive o quando si ha bisogno di allocazioni dinamiche:

```
let x = Box::new(5);
println!("{}", x);
```

Perciò, il valore 5 è allocato dinamicamente nell'heap, e *x* ne possiede l'ownership. Quando quest'ultimo esce dallo scope, Rust dealloca automaticamente la memoria associata.

L'Rc (*Reference Counting*) è uno smart pointer che consente di condividere l'ownership di un dato tra più parti del programma, mantenendo il conteggio di quanti riferimenti esistono a quel dato. È utilizzabile solo in contesti single-threaded, poiché il conteggio dei riferimenti non è thread-safe. Quando l'ultimo

riferimento a un dato viene rimosso, la memoria viene deallocata. Questo permette di gestire dati condivisi in maniera sicura, senza incorrere in condizioni di concorrenza non sicura. Ad esempio:

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

let a = Rc::new(String::from("ciao"));
let b = Rc::clone(&a); // Incrementa il conteggio dei riferimenti
println!("Riferimenti a: {}", Rc::strong_count(&a));
```

In questo codice, sia *a* che *b* condividono lo stesso dato, e Rust mantiene il conteggio dei riferimenti in modo che la memoria venga deallocata solo quando non ve ne sono più di attivi. Il conteggio dei riferimenti viene aggiornato automaticamente con *Rc::clone*.

L'Arc (*Atomic Reference Counting*) è simile a *Rc*, ma è thread-safe, il che lo rende utilizzabile in contesti multi-threaded. *Arc* utilizza *operazioni atomiche* per garantire che il conteggio dei riferimenti sia aggiornato correttamente anche in un ambiente concorrente, prevenendo condizioni di corsa. NB. Le operazioni atomiche, quando si utilizza *Arc*, sono operazioni che vengono eseguite in modo indivisibile, senza interruzioni, garantendo che nessun thread possa accedere o modificare lo stato condiviso contemporaneamente. *Arc* consente la condivisione sicura di dati tra più thread, gestendo il conteggio dei riferimenti in modo atomico, ossia garantendo che ogni incremento o decremento del contatore di riferimento avvenga senza interferenze da parte di altri thread. A differenza di *Rc*, che non è thread-safe, *Arc* utilizza operazioni atomiche per evitare condizioni di gara (race conditions) durante l'accesso concorrente ai dati, assicurando che il conteggio sia sempre corretto. Tuttavia, mentre *Arc* protegge il conteggio dei riferimenti, i dati interni non sono automaticamente thread-safe e devono essere protetti separatamente, spesso usando *Mutex* o altri meccanismi di sincronizzazione. Anche se leggermente meno efficiente rispetto a *Rc* a causa del sovraccarico delle operazioni atomiche, *Arc* è essenziale quando si ha bisogno di condividere dati tra più thread:

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

let dato = Arc::new(String::from("ciao"));
let dato_clonato = Arc::clone(&dato);
```

```
let handle = thread::spawn(move || {
    println!("{}", dato_clonato);
});

handle.join().unwrap();
```

Come detto, *Arc* permette di condividere la stringa *dato* tra il thread principale e uno nuovo. Il conteggio dei riferimenti è gestito in modo sicuro, garantendo che la memoria venga deallocata solo quando tutti i thread hanno terminato di usare il dato.
Quindi, in poche parole, sia *Rc* che *Arc* permettono di condividere dati tra più proprietari, tuttavia questi tipi sono immutabili per impostazione predefinita. Se è necessaria la mutabilità interna, ovvero la possibilità di modificare i dati anche quando si ha un riferimento immutabile, si può utilizzare *Cell* o *RefCell* per *Rc*, o *Mutex* per *Arc*. *RefCell* permette la mutabilità interna tramite il borrowing dinamico, verificando a runtime che non vi siano violazioni delle regole di borrowing. In contesti multithreaded, *Arc* può essere combinato con *Mutex*, che fornisce mutabilità sicura e sincronizzata, consentendo a più thread di accedere e modificare i dati condivisi senza rischiare condizioni di gara.
Essenzialmente gli smart pointer *Cell* e *RefCell* offrono meccanismi per aggirare le regole di borrowing e permettono di mutare dati attraverso riferimenti immutabili, seppur con alcune differenze. *Cell* è utilizzabile per dati semplici come tipi scalari (es. numeri), mentre *RefCell* è adatto per dati più complessi.
*Cell* consente di utilizzare la mutabilità interna per tipi *Copy*. Non permette prestiti di riferimenti ai dati contenuti, ma gestisce internamente i dati senza rispettare il borrowing esterno. Ad esempio:

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

let cella = Cell::new(5);
cella.set(10); // Modifica il valore
println!("{}", cella.get());
```

In questo caso, *Cell* permette di mutare il valore anche se la variabile cella è immutabile. La mutabilità è gestita internamente e il dato è copiato per lettura e scrittura.
*RefCell* è più flessibile di *Cell*, permettendo la mutabilità interna per dati *non Copy,* come stringhe o strutture complesse. La differenza chiave è che *RefCell* impone un controllo a runtime sul borrowing, mantiene infatti il conteggio di quanti prestiti mutabili o immutabili esistono, e genera un *panic* se si cerca di violare le regole di borrowing a runtime. Questo permette di gestire la mutabilità in modo dinamico, anche quando si hanno riferimenti immutabili:

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

let x = RefCell::new(String::from("ciao"));
x.borrow_mut().push_str(", mondo");
println!("{}", x.borrow());
```

Qui, *RefCell* permette di prendere in prestito mutabilmente la stringa per modificarla, anche se il dato è stato dichiarato come immutabile. Il compilatore non blocca questo comportamento, ma il runtime verificherà che non ci siano violazioni alle regole di borrowing (ad esempio, non ci possono essere più prestiti mutabili contemporaneamente).
*Cell* e *RefCell* sono strumenti molto utili in contesti in cui è necessario gestire la mutabilità di un dato all'interno di strutture più complesse, come le strutture dati immutabili che richiedono mutabilità interna. Tuttavia, l'uso di *RefCell* introduce un rischio a runtime, poiché un errore di borrowing produce un *panic*, richiedendo una maggiore attenzione nella gestione delle risorse.

### *Lifetimes avanzati*

I lifetimes rappresentano un meccanismo essenziale per garantire che i riferimenti siano sempre validi durante l'esecuzione del programma, prevenendo i classici errori di gestione della memoria come i dangling pointers. Sebbene Rust possa

spesso inferire automaticamente i lifetimes, ci sono casi in cui è necessario dichiararli esplicitamente per gestire riferimenti complessi e scenari di borrowing che coinvolgono dati con cicli di vita differenti.

L'introduzione dei *lifetimes espliciti* diventa essenziale quando il compilatore non riesce a dedurre correttamente il ciclo di vita dei dati. Questo accade, ad esempio, quando una funzione restituisce un riferimento che dipende da uno dei suoi parametri. Dichiarare manualmente i lifetimes in questi casi consente di legare esplicitamente la durata dei riferimenti a quella dei dati da cui essi dipendono, garantendo che il compilatore possa verificarne la validità. Consideriamo una funzione che accetta due riferimenti e restituisce uno di essi:

```
fn scegli_primo<'a>(x: &'a str, _y: &str) -> &'a str {
    x
}

let a = String::from("prima");
let b = String::from("seconda");
let risultato = scegli_primo(&a, &b);
```

In questo esempio, il lifetime *'a* indica che il riferimento restituito dalla funzione deve vivere almeno quanto il riferimento passato come parametro *x*. Questa esplicita indicazione evita che il programma faccia riferimento a dati non più validi dopo la chiamata alla funzione.

Le interazioni complesse tra più lifetimes sorgono quando una funzione o una struttura dati deve gestire più riferimenti con durate diverse. In tali casi, Rust richiede di dichiarare i lifetimes per evitare ambiguità. Ad esempio, supponiamo di averne due distinti per altrettanti parametri e un riferimento di ritorno che dipende dal lifetime più breve:

```
fn scegli<'a, 'b>(x: &'a str, y: &'b str) -> &'a str {
    x
}
```

Qui, *x* ha un lifetime *'a* e *y* ha un lifetime *'b*. Il riferimento restituito dipende dal

lifetime *'a*, il che significa che Rust garantisce che *x* rimanga valido almeno fino alla fine dell'uso del riferimento di ritorno. Se *y* avesse una durata più breve e fosse stato restituito invece di *x*, questo avrebbe potuto portare a un errore di riferimento a un dato non valido. In questi casi abbiamo il *lifetime subtyping* a supporto, che appunto consente di stabilire gerarchie tra *lifetimes*. Se *'a* è "più lungo" di un altro lifetime *'b*, si dice che *'a* è un *supertipo* di *'b*, e i riferimenti con *'a* possono essere utilizzati in contesti che richiedono *'b*. Questo avviene perché *'a* garantisce che il dato vivrà almeno quanto *'b*. In pratica, questo concetto consente di gestire meglio situazioni in cui i riferimenti passano attraverso contesti con diverse durate di vita. Un esempio comune si verifica con funzioni generiche che lavorano su riferimenti con lifetimes differenti:

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

Quindi, *'a: 'b* indica che *'a* deve vivere almeno quanto *'b*, e la funzione può restituire un riferimento con il lifetime *'b*. Questo tipo di subtyping permette di gestire riferimenti con durate di vita diverse mantenendo la coerenza del modello di ownership.

Quando i trait entrano in gioco, i lifetimes devono essere gestiti con attenzione, soprattutto nel caso dei trait object. Uno come *&dyn Trait*, rappresenta un riferimento dinamico a un oggetto che implementa un certo trait, ma comporta implicazioni sui lifetimes. Quando si crea un trait object, il ciclo di vita del riferimento deve essere esplicitamente gestito, poiché un trait object può implicare riferimenti a dati con cicli di vita differenti:

```
fn usa_trait_obj<'a>(dato: &'a dyn Mostrabile) {
    dato.mostra();
```

```
}

trait Mostrabile {
    fn mostra(&self);
}
```

Il trait *Mostrabile* ha un lifetime *'a*, che garantisce che il riferimento a *dato* resti valido per l'intera durata in cui viene utilizzato. Questo è particolarmente importante quando si utilizzano trait object con dati di durata variabile, poiché evita che i dati a cui fanno riferimento vengano invalidati prematuramente.
Anche i *trait bounds* con riferimenti possono diventare complessi quando coinvolgono lifetimes. Quando se ne dichiara uno su un tipo generico che prende in prestito un riferimento, è possibile limitare il ciclo di vita del riferimento utilizzando un lifetime esplicito:

```
fn mostra_con_trait<T: Mostrabile + 'static>(dato: T) {
    dato.mostra();
}
```

In questo caso, il trait bound *'static* assicura che qualsiasi dato passato alla funzione *mostra_con_trait* viva per l'intera durata del programma, prevenendo potenziali problemi con dati deallocati prematuramente.
Un'area in cui i lifetimes possono risultare complessi è nella gestione delle strutture dati annidate. Queste contengono riferimenti ad altre strutture e richiedono una chiara gestione dei lifetimes per garantire che i dati interni rimangano validi finché sono in uso. Supponiamo di avere una struttura che contiene un riferimento a una stringa:

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

impl<'a> Contenitore<'a> {
    fn nuovo(dato: &'a str) -> Self {
        Contenitore { dato }
```

```
    }
}
```

Se si annidano strutture simili, è necessario assicurarsi che i lifetimes dei riferimenti interni siano compatibili con quelli esterni. La difficoltà sta nel gestire correttamente i cicli di vita di ogni livello di riferimento, evitando che un dato venga deallocato mentre un riferimento ad esso è ancora in uso. Rust verifica rigorosamente i lifetimes in questi contesti, ma la corretta gestione richiede un'attenta dichiarazione esplicita dei cicli di vita.

Le limitazioni dei lifetimes emergono soprattutto quando si tenta di combinare dati con cicli di vita incompatibili o quando si lavora con strutture dati complesse che ne richiedono diversi. Per esempio, non è possibile creare strutture che contengano riferimenti che non rispettano le regole di borrowing, il che rende impossibili certi tipi di strutture dati senza usare tecniche avanzate come i *puntatori smart*.

Tuttavia, le potenzialità offerte dai lifetimes sono notevoli. Rust consente di gestire in modo sicuro e automatico la durata dei riferimenti, evitando errori di memoria senza dover ricorrere a garbage collector. Questo sistema di lifetimes permette di ottimizzare l'uso della memoria, garantendo che i dati vengano deallocati esattamente quando non sono più necessari, pur mantenendo la sicurezza dell'accesso ai dati attraverso riferimenti multipli e complessi.

### *Errori comuni e come evitarli*

Quando si utilizza il sistema di ownership e borrowing, è piuttosto abituale imbattersi in alcuni errori tipici, soprattutto durante le prime fasi di apprendimento o quando si gestiscono situazioni più complesse. Un errore comune riguarda il tentativo di utilizzare un dato dopo che è stato "spostato" (*move*) a un'altra variabile. Infatti, il *move* trasferisce l'ownership di un valore a una nuova variabile, *invalidando quella originale*. Ad esempio:

```
let a = String::from("ciao");
```

```
let b = a; // "a" viene spostato in "b"
println!("{}", a); // ERRORE: "a" non è più valido
```

Qui, dopo il *move*, *a* non è più utilizzabile poiché la sua ownership è stata trasferita a *b*. Questo è un meccanismo che evita il doppio uso della stessa risorsa e garantisce che i dati non vengano accidentalmente modificati o liberati due volte. Per risolvere problemi come questi, si può scegliere *di clonare il valore* anziché trasferirne la proprietà, garantendo che entrambi i valori siano utilizzabili:

```
let a = String::from("ciao");
let b = a.clone(); // "a" non viene spostato, ma cloniamo il dato
println!("{}", a); // Funziona, poiché "a" è ancora valido
```

Un altro errore comune legato al sistema di borrowing è tentare di avere più prestiti mutabili contemporaneamente o cercare di mutare un dato mentre è ancora in uso come riferimento immutabile. Rust impone regole rigorose: *non puoi avere sia un prestito immutabile che uno mutabile attivo nello stesso tempo*. Ad esempio:

```
let mut x = String::from("ciao");
let y = &x; // Prestito immutabile
let z = &mut x; // ERRORE: non è possibile avere un prestito mutabile
```

Questo errore riflette il principio fondamentale di Rust per non incorrere in condizioni di race o modifiche impreviste ai dati condivisi. Per evitarlo, assicurati di non mescolare prestiti immutabili e mutabili nello stesso contesto. In molti casi, può essere utile rilasciare il prestito immutabile prima di iniziare a mutare il dato:

```
let mut x = String::from("ciao");
{
    let y = &x; // Prestito immutabile in uno scope limitato
    println!("{}", y);
} // "y" esce dallo scope, il prestito viene rilasciato
let z = &mut x; // Ora possiamo mutare "x"
```

Quando si gestiscono cicli di vita complessi, il compilatore potrebbe non essere in

grado di dedurre correttamente i lifetimes, causando errori che indicano che il compilatore non può determinare se i riferimenti rimarranno validi. In questi casi, una strategia comune è quella di dichiararli esplicitamente. Ad esempio, in una funzione che restituisce un riferimento che dipende dai suoi parametri, è necessario indicare chiaramente la relazione tra i lifetimes degli argomenti e del risultato. Questo può essere fatto, appunto, dichiarando i *lifetimes espliciti*:

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

In questo modo, il compilatore può verificare che il riferimento restituito da *scegli_primo* abbia lo stesso ciclo di vita del parametro *x*, prevenendo errori in cui un dato potrebbe essere deallocato mentre è ancora in uso. Se si ha difficoltà con il sistema di lifetimes, *una buona pratica consiste nel semplificare il contesto dei riferimenti, magari dividendo le funzioni in più parti, o isolando il ciclo di vita dei dati in scope più piccoli per aiutare il compilatore a inferire correttamente i lifetimes*.

Un altro errore frequente si verifica quando si usano strutture dati complesse o cicli di vita annidati, dove può diventare difficile per il compilatore stabilire i corretti lifetime, come ad esempio nelle strutture che contengono riferimenti, dove è facile dimenticare di dichiararli, causando errori di compilazione:

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

impl<'a> Contenitore<'a> {
    fn nuovo(dato: &'a str) -> Self {
        Contenitore { dato }
    }
}
```

Se il compilatore non riesce a inferire correttamente i lifetimes, un approccio può

essere *dichiarare esplicitamente ogni ciclo di vita* in modo chiaro e definire la relazione tra i diversi riferimenti. Nel caso di strutture dati complesse, considerare l'uso di *smart pointers* come *Rc* o *Arc* può semplificare la gestione dei lifetimes, soprattutto quando si ha bisogno di condividere la proprietà di un dato tra più strutture.
Per scrivere codice più efficiente e sicuro, sfruttare al massimo la semantica del borrowing può portare a miglioramenti significativi. Ad esempio, usare riferimenti anziché trasferire dati può ridurre la quantità di operazioni di clonazione e spostamento dei dati, migliorando le prestazioni. Invece di trasferire l'ownership di grandi quantità di dati, è spesso più efficiente passare riferimenti immutabili, specialmente in contesti dove i dati non devono essere modificati:

```
fn stampa(dato: &str) {
    println!("{}", dato);
}

let s = String::from("ciao");
stampa(&s); // Passiamo un riferimento, non trasferiamo l'ownership
```

Così la stringa *s* non viene spostata né copiata, ma solo presa in prestito temporaneamente dalla funzione *stampa*, che la utilizza in modo sicuro senza modificarla. Questo approccio riduce il carico computazionale e previene la creazione di copie inutili di dati.
Un altro esempio di miglioramento dell'efficienza si trova nell'uso dei smart pointer come *Rc* e *Arc* per condividere dati tra più parti del programma senza spostarne l'ownership, l'abbiamo visti pocanzi. Utilizzando *Rc* (per contesti single-threaded) o *Arc* (per contesti multi-threaded), è possibile ridurre la quantità di duplicazione dei dati e garantire che la memoria venga liberata solo quando l'ultimo riferimento a quel dato viene rilasciato. Ad esempio, in un ambiente multi-threaded, possiamo utilizzare *Arc* per condividere in sicurezza un dato tra più thread:

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

```
let dato = Arc::new(String::from("ciao"));
let dato_clonato = Arc::clone(&dato);

let handle = thread::spawn(move || {
    println!("{}", dato_clonato);
});

handle.join().unwrap();
```

In questo scenario, *Arc* consente di condividere la stringa *dato* tra thread senza doverla clonare o trasferire. Questo riduce l'uso inefficiente di risorse e garantisce la sicurezza grazie al conteggio atomico dei riferimenti.
Proseguendo la discussione sugli errori comuni nel sistema di ownership, borrowing e lifetimes, possiamo esplorare ulteriormente le sfide legate alla mutabilità interna, al passaggio di riferimenti tra funzioni, e agli errori legati ai trait e agli smart pointer. Tutti questi aspetti, già accennati in precedenza, meritano una trattazione più approfondita per comprenderne appieno le implicazioni e come evitare problemi nelle fasi più avanzate dello sviluppo.
Uno degli errori comuni legati alla mutabilità interna si verifica quando si ha un riferimento immutabile a una struttura dati ma si ha comunque la necessità di modificarla. Rust impone che un dato preso in prestito come immutabile non possa essere modificato, ma ci sono situazioni in cui è necessario alterare lo stato interno di una struttura senza violare questa regola. Per gestire questo scenario, esistono strumenti come *Cell* e *RefCell*, che consentono di avere mutabilità interna anche quando un dato è preso in prestito come immutabile.
Un errore comune è l'utilizzo di mutabilità interna senza considerare i costi e i rischi di runtime che ciò comporta. Ad esempio, *RefCell* permette di eseguire mutazioni interne di dati che sono immutabili all'esterno, ma viola le garanzie di sicurezza di Rust a livello di compilazione e introduce errori a runtime se si tentano di avere più mutazioni simultanee o mutazioni mentre ci sono riferimenti immutabili attivi. Osserviamo un esempio:

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

struct Contatore {
    valore: RefCell<i32>,
}

let contatore = Contatore {
    valore: RefCell::new(0),
};

*contatore.valore.borrow_mut() += 1; // Mutazione permessa tramite RefCell
```

L'uso di *RefCell* permette di mutare *valore* anche se la struttura è immutabile. Tuttavia, un errore comune si verifica se si tenta di prendere in prestito il dato in modo mutabile e immutabile contemporaneamente:

```
let mutabile = contatore.valore.borrow_mut();
let immutabile = contatore.valore.borrow(); // ERRORE a runtime: borrow multiplo
```

Per evitare questo tipo di errori, è importante utilizzare *RefCell* con attenzione e solo quando è assolutamente necessario. Una buona pratica è limitare lo scope delle mutazioni e garantire che non ci siano aliasing di riferimenti mutabili e immutabili. Inoltre, è utile documentare chiaramente l'introduzione di *RefCell* in modo che gli sviluppatori successivi siano consapevoli dei rischi di runtime.
Un altro errore comune è legato al passaggio di riferimenti tra funzioni senza prestare attenzione ai *lifetimes*. Quando si lavora con questo tipo di funzioni, Rust verifica che il *ciclo di vita* dei riferimenti sia correttamente gestito, prevenendo l'uso di dati che potrebbero essere deallocati. Tuttavia, un errore ricorrente è non dichiarare esplicitamente i *lifetimes* quando necessario, causando errori di compilazione difficili da interpretare. Ad esempio, consideriamo il caso di una funzione che restituisce un riferimento basato su uno dei suoi parametri:

```
fn prendi_riferimento<'a>(x: &'a str) -> &'a str {
    x
}
```

Qui, il lifetime *'a* dichiara esplicitamente che il riferimento restituito avrà lo stesso ciclo di vita del parametro *x*, evitando errori in cui il compilatore non riesce a determinarne la validità. Se si omette il lifetime, Rust può generare errori di inferenza, soprattutto quando la funzione coinvolge più parametri con cicli di vita diversi. Una buona pratica è dichiarare i lifetimes esplicitamente quando si ha a che fare con funzioni che gestiscono riferimenti, soprattutto in contesti più complessi.

Un'altra area in cui si possono commettere errori è l'integrazione dei trait con il sistema di ownership e borrowing. Quando si utilizzano *trait bounds* su funzioni generiche o si lavora con *trait object*, i lifetimes devono essere gestiti con particolare attenzione. Ad esempio, se si ha una funzione che accetta un *trait bound* su un tipo che prende in prestito un riferimento, è necessario specificare esplicitamente il ciclo di vita del riferimento per evitare violazioni delle regole di borrowing:

```
fn stampa_con_trait<T: Mostrabile + 'static>(dato: T) {
    dato.mostra();
}
```

In questo caso, il trait bound *'static* garantisce che qualsiasi dato passato alla funzione viva per l'intera durata del programma. Tuttavia, un errore comune è dichiarare lifetimes troppo restrittivi, limitando inutilmente la flessibilità della funzione. Una buona pratica consiste nel dichiararli esplicitamente solo quando necessario, e lasciare che il compilatore inferisca i lifetimes nei casi più semplici.

Gli errori legati ai *smart pointer* come *Rc* e *Arc* sono altrettanto comuni, specialmente quando si passa da contesti single-threaded a multi-threaded. *Rc* è adatto per la condivisione di dati in contesti single-threaded, ma se si tenta di utilizzarlo in un contesto multi-threaded, Rust genererà errori di compilazione poiché *Rc* non è sicuro per l'accesso concorrente. In questi casi, è necessario utilizzare *Arc*, che garantisce la sicurezza in contesti multi-threaded tramite un conteggio di riferimenti atomico:

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

let dato = Arc::new(String::from("ciao"));
let dato_clonato = Arc::clone(&dato);

let handle = thread::spawn(move || {
    println!("{}", dato_clonato);
});

handle.join().unwrap();
```

Un errore comune in questo contesto è dimenticare di usare *Arc* quando si condivide un dato tra thread, causando problemi di concorrenza e violazioni di sicurezza. Per evitare questo tipo di errori, è utile prestare attenzione a quale smart pointer utilizzare in base al contesto, e documentare chiaramente le decisioni relative all'uso di *Rc* o *Arc*.

Infine, è importante evitare transferimenti non desiderati dei dati, specialmente quando si lavora con grandi strutture. Il *move* implicito può risultare in comportamenti inaspettati se non si è consapevoli di come funziona il trasferimento di ownership. Una strategia comune per evitare spostamenti indesiderati è l'uso di clonazione esplicita o di riferimenti immutabili. Ad esempio, nel caso di grandi strutture dati, può essere più efficiente clonare solo una parte del dato o passare riferimenti immutabili anziché trasferire l'intera struttura:

```
let grande_dato = String::from("molti dati");
let clone = grande_dato.clone(); // evita il move e conserva l'originale
```

Utilizzare la clonazione può sembrare costoso, ma in alcuni casi è preferibile rispetto al trasferimento di proprietà, specialmente quando si ha bisogno di accedere al dato originale in seguito. Una buona pratica è quindi bilanciare l'uso di clonazione e borrowing, ottimizzando le risorse e prevenendo transferimenti indesiderati.

In conclusione, evitare errori nel sistema di ownership, borrowing e lifetimes

richiede una profonda comprensione del comportamento di questi meccanismi. Dichiarare *lifetimes* espliciti quando il compilatore non riesce a inferirli correttamente, limitare l'uso di *RefCell* per mutabilità interna e scegliere attentamente tra *Rc* e *Arc* sono tutte strategie fondamentali per prevenire problemi comuni. Scrivere codice sicuro ed efficiente in Rust implica sfruttare al massimo la semantica del borrowing e comprendere appieno le interazioni tra ownership e gestione della memoria.

### *Threading sicuro con ownership e borrowing*

Il sistema di ownership e borrowing si estende in modo coerente anche alla gestione della concorrenza, garantendo la sicurezza dei dati attraverso il compilatore e prevenendo errori comuni, come le race conditions, che potrebbero verificarsi in ambienti multi-threaded. La base di questa sicurezza deriva dalla proprietà e dal controllo sui riferimenti che Rust impone a livello di compilazione, ma quando si entra nel dominio della concorrenza, è essenziale comprendere come funzionano i meccanismi di threading sicuro attraverso l'uso di trait specializzati come *Send* e *Sync*, e strutture come *Arc* e *Mutex* che facilitano l'accesso sicuro ai dati condivisi tra thread.

Nel modello di ownership, un thread è il proprietario dei dati che utilizza, e per condividerli, è necessario spostarli nel thread scelto, oppure farlo in modo sicuro attraverso due trait chiave: *Send* e *Sync*. Un tipo è *Send* se può essere trasferito in modo sicuro tra thread, mentre un tipo è *Sync* se può essere condiviso tra thread in modo sicuro tramite *riferimenti*. La maggior parte dei tipi primitivi in Rust è automaticamente *Send* e *Sync*, ma alcuni tipi che gestiscono risorse come Rc (*Reference Counting*) non lo sono per mancanza di sicurezza in contesti concorrenti senza ulteriori meccanismi di sincronizzazione.

Ad esempio, se si tenta di usare un Rc in un contesto multi-threaded, il compilatore genererà un errore, poiché questo non è sicuro da usare con accessi concorrenti. In questi casi, è necessario utilizzare *Arc*, ovvero *Atomic Reference*

*Counting,* che garantisce la sicurezza in contesti multi-threaded attraverso l'uso di operazioni atomiche. *Arc* può essere condiviso tra thread, ma per accedere in modo sicuro ai dati contenuti all'interno, è spesso combinato con una struttura come *Mutex* per garantire che un solo thread possa modificare i dati alla volta. Ecco un esempio che mostra l'uso di *Arc* e *Mutex* insieme per gestire i dati condivisi:

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

let dato_condiviso = Arc::new(Mutex::new(0));

let mut handle_vec = vec![];

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

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

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

Usiamo *Arc* per consentire la condivisione sicura del dato tra più thread, mentre *Mutex* garantisce che un solo thread possa accedere al valore alla volta. *Arc* fornisce il conteggio atomico dei riferimenti, evitando che il dato venga deallocato prematuramente, e *Mutex* assicura che le operazioni di modifica siano sicure e non concorrenti, evitando condizioni di race. Quando un thread vuole accedere al dato, acquisisce un *lock* sul *Mutex*, garantendo che nessun altro thread possa accedere a quel dato fino a quando il *lock* non viene rilasciato. Questo meccanismo assicura che anche se il dato è condiviso tra più thread, le modifiche avvengano in modo

sicuro e ordinato.
Un errore comune in questo contesto è non gestire correttamente i *lock* del *Mutex*. Quando un thread lo acquisisce, è possibile che questo processo fallisca, quindi è necessario gestire l'errore utilizzando *unwrap()* o un'altra strategia per trattare il fallimento. Inoltre, è importante rilasciare il *lock* il prima possibile per evitare il *deadlock*, una situazione in cui due o più thread rimangono bloccati perché stanno aspettando di acquisire il lock su risorse che l'altro possiede. *Una buona pratica è limitare lo scope dei lock,* evitando di tenere bloccato il dato per troppo tempo.
Il trait *Send* garantisce che i dati possano essere trasferiti tra thread senza causare problemi di memoria. Ad esempio, se un tipo contiene riferimenti a dati che non vivono abbastanza a lungo o che non sono sicuri da condividere tra thread, non implementerà *Send*, prevenendo così errori. Allo stesso modo, il trait *Sync* indica che è sicuro accedere a un tipo attraverso riferimenti immutabili da più thread. La combinazione di *Send* e *Sync* permette di creare modelli di concorrenza sicuri, dove i dati possono essere condivisi o trasferiti tra thread senza violare le regole di ownership.
Un altro strumento fondamentale per gestire la concorrenza in modo sicuro è il sistema di borrowing. Nel contesto della concorrenza, il borrowing garantisce che i riferimenti a dati condivisi siano utilizzati in modo coerente tra i vari thread. Anche quando i dati sono condivisi con *Arc*, il borrowing di quei dati avviene in base alle stesse regole viste nel contesto single-threaded: un riferimento mutabile esclusivo o più riferimenti immutabili, mai entrambi contemporaneamente. L'uso di *Mutex* e *RwLock* (un'altra struttura di sincronizzazione che consente letture concorrenti e scritture esclusive) rispetta queste regole, mantenendo la sicurezza anche nel multithreading.
Il lifetime gioca un ruolo importante nella concorrenza poiché Rust deve garantire che ogni dato condiviso o trasferito tra thread rimanga valido per tutta la durata necessaria. Quando si passano riferimenti a thread, è fondamentale che i cicli di vita dei dati siano ben definiti e corrispondano alla durata dell'esecuzione del

thread stesso. Se un riferimento non vive abbastanza a lungo, Rust genererà errori a livello di compilazione per prevenire il rischio di accesso a dati non validi. Ad esempio, se si tenta di passare un riferimento con un ciclo di vita limitato a un thread che lo utilizzerà oltre quel ciclo di vita, Rust impedirà la compilazione del programma.
Un errore comune in questo contesto è cercare di condividere riferimenti diretti tra thread senza utilizzare *Arc.* Poiché il dato potrebbe essere deallocato prematuramente o non essere thread-safe, il compilatore impedirà il trasferimento di riferimenti a dati non sicuri. La soluzione in questi casi è utilizzare *Arc* o, se necessario, *Arc* combinato con *Mutex* o *RwLock* per garantire che il dato sia valido e accessibile in modo sicuro per tutta la durata del thread.
Tuttavia, è fondamentale comprendere bene questi strumenti e usarli con cautela per evitare errori come deadlock, blocchi prolungati o errori di borrowing in contesti concorrenti.

***Gestione delle risorse***

Il modello di Rust per la gestione delle risorse è un'estensione naturale del suo sistema di ownership, utilizzando i concetti di proprietà e borrowing per gestire non solo la memoria ma anche risorse più complesse come *file, socket, connessioni di rete* e altre risorse di sistema. In questo contesto, Rust adotta un modello basato su RAII (*Resource Acquisition Is Initialization*), che lega la gestione delle risorse al ciclo di vita degli oggetti, consentendo il rilascio sicuro e automatico delle risorse quando queste escono dallo scope.
Quando si lavora con risorse come *file* o *socket,* Rust sfrutta il sistema di ownership per garantire che una risorsa sia posseduta da un’unica entità alla volta, prevenendo accessi non sicuri o condizioni di gara (race conditions). Ad esempio, quando si apre un file, la risorsa viene posseduta dall'oggetto *File*. Quando l'oggetto esce dallo scope, Rust invoca automaticamente il suo *drop* per chiudere il file e liberare la risorsa. Questo comportamento si basa sul concetto di *destructor*,

che è gestito tramite il trait *Drop*.
Un esempio semplice di gestione delle risorse con un file potrebbe essere il seguente:

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

fn apri_file() {
    let file = File::open("example.txt").unwrap();
    // Il file viene automaticamente chiuso quando esce dallo scope
}
```

In questo esempio, il file viene aperto e posseduto dall'oggetto file. Quando l'esecuzione della funzione *apri_file* termina, l'oggetto file esce dallo scope, e Rust rilascia la risorsa chiamando il *Drop* implementato su *File,* che si occupa di chiudere il file in modo sicuro. Questo approccio evita che i programmatori debbano gestire manualmente la chiusura di risorse come i file, riducendo il rischio di bug legati alla gestione errata di queste risorse.
In contesti più avanzati, Rust permette di utilizzare costrutti RAII per gestire risorse con cicli di vita complessi, come risorse che devono essere condivise o modificate da più entità. Un esempio di risorsa condivisa potrebbe essere una connessione di rete che viene utilizzata da più thread. Per gestire questa complessità, sappiamo già che esistono strumenti come *Arc* e *Mutex* per garantire la sicurezza e il controllo sull'accesso concorrente.
Consideriamo un caso in cui più thread devono accedere a un socket condiviso. Si può usare un *Arc<Mutex<Socket>>* per garantire che solo un thread alla volta possa accedere allo socket:

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

fn main() {
    let socket = TcpStream::connect("127.0.0.1:8080").unwrap();
    let socket = Arc::new(Mutex::new(socket));
```

```
    let thread_socket = Arc::clone(&socket);
    thread::spawn(move || {
        let mut sock = thread_socket.lock().unwrap();
        // Utilizza il socket in modo sicuro
    });

    let mut sock = socket.lock().unwrap();
    // Usa il socket anche qui in modo sicuro
}
```

Nell'esempio, lo socket TCP è protetto da un *Mutex* e condiviso tra più thread attraverso *Arc*. Il *Mutex* garantisce che solo un thread possa accedere allo socket in un dato momento, evitando condizioni di gara. Il ciclo di vita dello socket è gestito automaticamente, e la risorsa viene rilasciata in modo sicuro una volta che tutte le referenze ad *Arc* escono dallo scope.

Un altro aspetto avanzato della gestione delle risorse è la possibilità di implementarne pattern propri. Questo può essere fatto adottando il trait *Drop* per tipi personalizzati. Viene chiamato automaticamente quando un'istanza del tipo esce dallo scope, permettendo di rilasciare risorse associate come file, connessioni di rete o blocchi di memoria personalizzati. Vediamo subito un esempio in cui implementiamo *Drop* per gestire una connessione simulata:

```
struct Connessione {
    id: u32,
}

impl Connessione {
    fn new(id: u32) -> Connessione {
        Connessione { id }
    }

    fn invia_dati(&self) {
        println!("Invio dati sulla connessione {}", self.id);
    }
}

impl Drop for Connessione {
```

```
    fn drop(&mut self) {
        println!("Chiudo connessione {}", self.id);
    }
}

fn main() {
    let conn = Connessione::new(1);
    conn.invia_dati();
    // Quando `conn` esce dallo scope, `Drop` viene chiamato automaticamente
}
```

In questo listato, la struttura *Connessione* simula una connessione di rete, e implementiamo *Drop* per assicurarci che questa venga chiusa correttamente quando esce dallo scope. La funzione *drop* viene chiamata automaticamente da Rust, e l'utente non deve preoccuparsi di chiudere manualmente la connessione.
Quando si adottano pattern di gestione delle risorse, è essenziale tenere conto dei *lifetimes*, *i quali assicurano che una risorsa non venga rilasciata prematuramente mentre ci sono ancora riferimenti attivi ad essa*. Se si lavora con risorse complesse che devono essere condivise o utilizzate in contesti con cicli di vita complicati, Rust offre strumenti per gestire esplicitamente i lifetimes, garantendo che i riferimenti siano validi per tutto il tempo necessario.
Un esempio di utilizzo dei lifetimes in un contesto di gestione delle risorse può essere visto nel passaggio di riferimenti a risorse in funzioni che operano su di esse senza prendere la proprietà. Consideriamo una funzione che accetta un riferimento a un file e legge i dati da esso senza trasferire la proprietà:

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

fn leggi_file(file: &File) -> io::Result<String> {
    let mut contenuto = String::new();
    file.take(100).read_to_string(&mut contenuto)?;
    Ok(contenuto)
}

fn main() {
```

```
    let file = File::open("example.txt").unwrap();
    let dati = leggi_file(&file).unwrap();
    println!("Dati letti: {}", dati);
}
```

La funzione *leggi_file* prende in prestito un riferimento immutabile al file e legge i primi 100 byte senza trasferire la proprietà del file. Il file rimane di proprietà del chiamante e può essere utilizzato anche dopo che la funzione ha restituito il controllo. Questo schema di borrowing è sicuro e garantisce che la risorsa venga gestita correttamente senza rischi di accesso simultaneo o perdita di dati.
In sintesi, Rust fornisce un sistema robusto per gestire risorse complesse come *file, socket* e altre risorse di sistema attraverso il suo modello di ownership e borrowing. L'uso del RAII assicura che le risorse vengano rilasciate automaticamente quando escono dallo scope, prevenendo problemi di gestione manuale. Rust consente anche di implementare pattern avanzati di gestione delle risorse tramite il trait *Drop*, garantendo che ogni risorsa venga trattata in modo sicuro e idiomatico. I lifetimes garantiscono che i riferimenti a risorse siano validi per tutto il tempo necessario, e strumenti come *Arc* e *Mutex* rendono possibile la gestione sicura di risorse condivise in contesti concorrenti.

### *Performance e ottimizzazioni relative a ownership*

Il sistema di ownership e borrowing ha un impatto significativo sulle performance del programma, rendendolo più efficiente rispetto a linguaggi che si affidano a sistemi automatici. Questo approccio consente una gestione più diretta e prevedibile della memoria, riducendo il sovraccarico tipico della gestione automatica di risorse attraverso il garbage collection. Infatti, in linguaggi come Java o Go, la gestione della memoria avviene in modo dinamico durante l'esecuzione, con il meccanismo di pulizia che interviene per liberarne le aree non più utilizzate. In Rust, invece, questo processo è determinato in fase di compilazione grazie al sistema di ownership e ai lifetimes, il che permette un controllo più fine sulle risorse, riducendo overhead e garantendo una maggiore

prevedibilità.
Un elemento chiave per l'efficienza è che, una volta che un'istanza viene spostata (tramite un *move*), non ci sono duplicazioni inutili dei dati. Il trasferimento di proprietà avviene a livello di memoria senza dover effettuare una copia dei dati stessi. *Questo comportamento riduce al minimo l'allocazione e la deallocazione della memoria*, evitando il sovraccarico dovuto a operazioni di copia superflue, come può accadere in linguaggi che non distinguono tra spostamenti e copie. Vediamolo in un esempio di *move* implicito:

```
fn main() {
    let s1 = String::from("ciao");
    let s2 = s1; // Move
    println!("{}", s2); // s1 non è più utilizzabile
}
```

La stringa *s1* viene spostata in *s2* senza essere copiata. L'efficienza di questo meccanismo dipende dal fatto che *s1* perde il controllo della memoria e *s2* ne diventa il nuovo proprietario. *Nessuna duplicazione dei dati avviene in memoria*, *poiché Rust trasferisce direttamente la proprietà*. Questa ottimizzazione è resa possibile dal fatto che il linguaggio garantisce che *s1* non venga più utilizzato dopo il *move*, eliminando ogni rischio di accesso a memoria non valida.
Tuttavia, come già detto, ci sono casi in cui il *move* non è desiderabile o efficiente, specialmente quando si vuole mantenere più copie valide di un dato oggetto. In questi casi, può essere necessario usare la clonazione esplicita tramite il metodo *clone*, che crea una copia completa dei dati. La clonazione ha un costo in termini di performance, poiché implica l'allocazione di nuova memoria e la copia dei contenuti. Rust incoraggia l'uso di *borrowing* al posto del *cloning* quando possibile, poiché il primo consente di accedere ai dati senza duplicarli o trasferirne la proprietà:

```
fn main() {
    let s1 = String::from("ciao");
    let s2 = &s1; // Borrowing
```

```
    println!("{}", s1); // s1 può ancora essere usato
    println!("{}", s2); // s2 è un riferimento a s1
}
```

In questo caso, *s2* è un riferimento immutabile a *s1*. Nessun dato viene copiato o spostato, ma si accede direttamente ai dati originali. Questo evita la duplicazione della memoria e mantiene le performance elevate, specialmente quando si trattano dati di grandi dimensioni.
Le ottimizzazioni per ridurre la necessità di *clone* o *move* inefficaci sono particolarmente importanti. Quando si gestiscono strutture dati complesse o grandi, è possibile sfruttare i riferimenti immutabili e mutabili per evitare copie non necessarie. Ad esempio, nelle funzioni che prendono in input strutture dati, è preferibile passare riferimenti ai dati piuttosto che trasferire la proprietà, a meno che non sia strettamente necessario. Questo permette di ridurre le allocazioni di memoria e mantenere elevata l'efficienza. Ecco un esempio:

```
fn somma_vettore(v: &Vec<i32>) -> i32 {
    v.iter().sum()
}

fn main() {
    let v = vec![1, 2, 3, 4, 5];
    let risultato = somma_vettore(&v);
    println!("La somma è {}", risultato);
}
```

In questo esempio, la funzione *somma_vettore* prende un riferimento a *Vec<i32>,* evitando di trasferire la proprietà del vettore. Questo approccio riduce il costo computazionale di una copia completa e consente alla funzione chiamante di mantenere il controllo sui dati originali, migliorando l'efficienza.
Un altro vantaggio di questo sistema è che consente una migliore gestione delle risorse rispetto ad altri linguaggi, dove la memoria viene rilasciata automaticamente dopo alcuni fatidici passaggi, il che può portare a pause impreviste e rallentamenti, specialmente in applicazioni ad alte prestazioni o in

tempo reale. *In Rust, invece, la memoria e le risorse vengono rilasciate non appena l'oggetto esce dallo scope.* Questo comportamento predeterminato consente di evitare il costo del garbage collection e di gestire le risorse in modo più efficiente. L'allocazione e la deallocazione delle risorse seguono un modello deterministico, riducendo la latenza e garantendo performance stabili.
In applicazioni ad alte prestazioni come i sistemi embedded o i giochi, questa prevedibilità è cruciale. Per esempio, in un gioco, la gestione della memoria deve essere immediata per evitare rallentamenti improvvisi.
Un altro aspetto fondamentale delle performance è legato all'uso di *stack* e *heap*. Poiché Rust permette di gestire direttamente dove e come allocare i dati, è possibile ottimizzare il codice *in modo che la maggior parte dei dati venga allocata sullo stack, che è significativamente più veloce rispetto all'allocazione sul heap.* Per le strutture dati più complesse che devono vivere a lungo, Rust offre meccanismi per gestire in modo efficiente le allocazioni sul heap tramite tipi come *Box, Rc* o *Arc*. Tuttavia, il sistema di ownership permette di evitare il sovrautilizzo di questi strumenti, allocando sull'heap solo quando strettamente necessario e gestendo in modo esplicito il ciclo di vita degli oggetti.
In conclusione, il sistema di ownership e borrowing ha un impatto positivo sulle performance del programma poiché riduce la necessità di allocazioni e deallocazioni dinamiche. Il *move* è efficiente poiché non copia i dati, ma ne trasferisce semplicemente la proprietà, e il *borrowing* consente di accedervi senza duplicarli. Le ottimizzazioni derivano dalla possibilità di evitare copie inutili, utilizzare *stack* e *heap* in modo efficiente e sfruttare riferimenti per ridurre il sovraccarico di memoria.

### *Gli attributi*

Gli attributi sono metadati che possiamo applicare a vari elementi del codice come funzioni, strutture, enumerazioni, moduli e *crate* per influenzarne il comportamento durante la compilazione o l'esecuzione. Consentono al

programmatore di fornire al compilatore delle istruzioni aggiuntive o di abilitare funzionalità specifiche, come l'ottimizzazione del codice, la gestione di warning, o l'esecuzione di test. Gli attributi sono preceduti dal simbolo # e racchiusi tra parentesi quadre. Ad esempio, *#[inline]* suggerisce al compilatore di "inserire" il codice di una funzione al momento della sua chiamata, per migliorare le prestazioni. Vengono generalmente utilizzati per scopi come:

- Ottimizzazione delle performance (es. #[inline])
- Automazione di implementazioni (es. #[derive] per generare trait comuni come Debug o Clone)
- Debugging o test (es. #[test] per dichiarare test unitari)
- Controllo delle configurazioni di compilazione condizionale (es. #[cfg] per includere codice solo su piattaforme specifiche)

| Metodi | Utilizzo | Esempio |
|---|---|---|
| `#[inline]` | *Suggerisce al compilatore di "inlineare" la funzione, ovvero sostituirla con il suo codice al momento della chiamata.* | `#[inline]`<br>`fn somma(a: i32, b: i32) -> i32 {`<br>` a + b\n}` |
| `#[derive]` | *Genera automaticamente implementazioni di trait come Debug, Clone, Eq, PartialEq, ecc. per una struttura o enumerazione.* | `#[derive(Debug, Clone)]`<br>`struct Punto {`<br>` x: i32,`<br>` y: i32,`<br>`}` |
| `#[allow]` | *Disabilita specifici warning del compilatore all'interno di una funzione o blocco di codice.* | `#[allow(dead_code)]`<br>`fn funzione_non_usata() {`<br>` println!(\"Questa funzione non è usata.\");`<br>`}` |
| `#[cfg]` | *Condiziona la compilazione di una parte di codice a specifiche configurazioni (come sistemi operativi o feature).* | `#[cfg(target_os = \"windows\")]`<br>`fn piattaforma() {\n println!(\"Codice per Windows\");`<br>`}` |
| `#[test]` | *Indica che una funzione è un test unitario e viene eseguita durante l'esecuzione dei test.* | `#[test]`<br>`fn test_somma() {`<br>` assert_eq!(somma(2, 2), 4);`<br>`}` |
| `#[panic]` | *Imposta il comportamento da adottare in caso di panic, come terminare il programma o stampare un messaggio di errore.* | `#[panic_handler]`<br>`fn gestore_panic(info:`<br>`&core::panic::PanicInfo) -> ! {`<br>` println!(\"Errore: {}\", info);`<br>` loop {}`<br>`}` |
| `#[expect]` | *Utilizzato per annotare test che potrebbero generare errori o warning, per gestire attese specifiche di errori o risultati in fase di test.* | `#[test]`<br>`#[expect(panics)]`<br>`fn test_panico() {`<br>` panic!(\"Questo test genera un panic.\");`<br>`}` |
| `#[repr]` | *Specifica come una struttura o enumerazione dovrebbe essere rappresentata in memoria (utile per compatibilità con altri linguaggi).* | `#[repr(C)]`<br>`struct PuntoC {`<br>` x: i32,`<br>` y: i32,`<br>`}` |
| `#[macro_use]` | *Permette di importare macro da altri* | `#[macro_use]`<br>`extern crate mia_crate;` |

| | | |
|---|---|---|
| | *moduli o crate senza doverle esplicitamente importare con use.* | |
| `#[non_exhaustive]` | *Impedisce ai client di un crate di esaurire le possibili varianti di un'enumerazione o di una struttura, utile per future estensioni.* | `#[non_exhaustive]`<br>`struct Configurazione {`<br>`param1: i32,`<br>`param2: i32,`<br>`}` |

## *Quiz & Esercizi*

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

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

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

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

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.

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.

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

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.

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

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.

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

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

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

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

15) Scrivere una funzione che utilizzi il concetto di *move* esplicito, restituendo una variabile da una funzione per trasferire l'ownership indietro.
16) Scrivere una funzione che accetti un *trait bound* con un riferimento immutabile, dimostrando l'interazione tra trait bounds e borrowing.
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.
18) Scrivere una funzione che sposti (*move*) un valore in una struttura e dimostri come i *move* funzionano con le struct.
19) Scrivere una funzione generica che accetti un *trait bound* con riferimento e che implementi un'operazione di borrowing su una struct generica.
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.
21) Creare un puntatore intelligente *Box* per gestire un valore su heap e dimostrare come l'ownership è trasferita.
22) Utilizzare *Rc* per gestire la condivisione di dati in più parti del programma, dimostrando come avviene il conteggio di riferimenti.
23) Utilizzare *Arc* per gestire la condivisione di dati tra thread e dimostrare come si differenzia da *Rc* per la gestione concorrente.
24) Utilizzare *RefCel*l per permettere mutabilità interna in una struttura, anche quando è referenziata in modo immutabile.
25) Dimostrare l'uso di *Cell* per la mutabilità interna quando si trattano tipi primitivi copiabili.
26) Utilizzare *Mutex* per garantire accesso esclusivo a una variabile condivisa tra thread, prevenendo condizioni di gara.
27) Utilizzare *Arc* e *Mutex* insieme per condividere in sicurezza una variabile tra thread multipli e prevenire condizioni di gara
28) Utilizzare Rc e RefCell insieme per gestire mutabilità interna in una struttura condivisa senza usare multi-threading.
29) Utilizzare *Arc, Mutex* e una funzione asincrona per gestire la concorrenza con un contatore condiviso che viene modificato da più task.
30) Utilizzare *Cell* per mutare un valore copiabile in una funzione che richiede l'accesso immutabile alla struttura, dimostrando la mutabilità interna.

***Riassunto***

In queste pagine abbiamo esplorato in dettaglio il meccanismo di ownership e borrowing, concentrandoci sugli aspetti avanzati e sulle interazioni più complesse tra lifetimes, mutabilità e gestione della memoria. Il concetto di *move*, sia implicito che esplicito, è stato trattato, sottolineando

l'impatto di questi trasferimenti di proprietà sul comportamento del programma. È emerso come il *move* prevenga la duplicazione non necessaria di dati, mentre il borrowing consente di accedere ai dati senza trasferirne la proprietà, con esempi che evidenziano l'efficienza di questi meccanismi rispetto alla clonazione. Abbiamo poi analizzato la mutabilità e i prestiti multipli, osservando come Rust consenta di prendere in prestito dati in modo sicuro, imponendo regole rigorose sull'immutabilità e mutabilità per prevenire condizioni di gara e violazioni della memoria.

Un punto centrale è stato il ruolo dei lifetimes nella gestione dei riferimenti e come i lifetimes espliciti aiutino a risolvere situazioni in cui il compilatore non riesce a inferire correttamente i cicli di vita dei dati. Abbiamo esplorato come il sistema di Rust gestisca funzioni che accettano e restituiscono riferimenti, senza trasferimenti di proprietà, concentrandoci anche sulla gestione di dati complessi con lifetimes estesi. È stata discussa l'integrazione di ownership e borrowing con strutture più avanzate, come i trait, con particolare attenzione all'uso di trait bounds con riferimenti.

Abbiamo poi approfondito i puntatori intelligenti, concentrandoci su *Box, Rc,* e *Arc,* con particolare attenzione alla gestione della memoria condivisa sicura. È emerso come *Rc* e *Arc* utilizzino il conteggio dei riferimenti per gestire risorse condivise, con *Arc* specificamente pensato per contesti multithread. Il concetto di mutabilità interna è stato collegato all'uso di *Cell* e *RefCell* per gestire dati modificabili anche quando sono presenti riferimenti immutabili.

Infine, abbiamo visto come Rust consenta di scrivere codice più efficiente e sicuro, sfruttando il suo sistema di ownership per ottimizzare la gestione della memoria rispetto ai linguaggi con garbage collector. Le performance migliorano grazie alla riduzione di operazioni di clonazione e spostamenti inutili, e Rust permette un controllo fine sull'allocazione e deallocazione delle risorse. In contesti multithreaded, abbiamo analizzato l'uso di *Arc* e *Mutex* per garantire la sicurezza nella gestione dei dati condivisi, evidenziando le operazioni atomiche come fondamentali per evitare condizioni di gara. Tutto questo dimostra come Rust riesca a combinare sicurezza e performance, senza sacrificare l'efficienza nella gestione delle risorse e del ciclo di vita dei dati.

# # 6 - Le espressioni regolari

*La sintassi di base*
*Le sequenze speciali*
*Metodi e Regex*

Eccoci giunti all'argomento probabilmente più amato, odiato e ostico dei linguaggi di programmazione. Parliamo delle espressioni regolari, le quali, tuttavia, sono un potente strumento e una necessità nello sviluppo dei programmi. Le espressioni regolari sono spesso considerate un argomento complesso e difficile nei linguaggi di programmazione, ma sono anche uno strumento potente e indispensabile nello sviluppo di programmi. Sebbene la loro sintassi possa sembrare complicata a prima vista, con l'uso appropriato, le espressioni regolari possono condensare intere porzioni di codice in una sola affermazione, semplificando la scrittura del codice e accelerando l'analisi delle stringhe.
Le espressioni regolari, o regex, costituiscono uno strumento potentissimo per la manipolazione e la ricerca di stringhe di testo, che trova applicazione in numerosi ambiti della programmazione. L'integrazione in Rust si allinea con i principi fondamentali del linguaggio, mantenendo il focus su prestazioni elevate e sicurezza. Il *crate regex*, disponibile nella libreria standard, è costruito per garantire una sintassi familiare per chi ha esperienza con le regex in altri linguaggi, come Perl o Python, ma con un'implementazione che tiene conto della necessità di ottimizzare la gestione della memoria e prevenire inefficienze nell'esecuzione.
Uno degli aspetti principali dell'approccio di Rust è l'assenza di *backtracking* nel

motore regex. Questo impedisce che espressioni regolari complesse o malformate possano condurre a esecuzioni lente o a fenomeni di "catastrofe da backtracking". Ne risulta un comportamento più prevedibile e stabile in fase di esecuzione, particolarmente vantaggioso quando si lavora con input di grandi dimensioni o in contesti critici dal punto di vista delle prestazioni.

La sintassi e le funzionalità delle espressioni regolari in Rust sono ampiamente compatibili con quelle utilizzate in altri linguaggi, includendo operatori come caratteri jolly, quantificatori, ancore, gruppi di cattura e classi di caratteri. Tuttavia, è importante sottolineare che il crate regex non supporta funzionalità come il lookahead o il lookbehind, presenti in alcune implementazioni avanzate di regex. Questa limitazione è il risultato di una precisa scelta progettuale, volta a preservare le caratteristiche di efficienza e prevedibilità del motore regex di Rust.

Dal punto di vista della sicurezza, il linguaggio, fedele alla sua filosofia di "memory safety", evita l'uso di puntatori grezzi o manipolazioni dirette della memoria nell'implementazione delle *regex*. Il crate è stato progettato per essere utilizzato in modo sicuro anche in ambienti concorrenti, senza incorrere in race condition o altri problemi legati alla gestione condivisa delle risorse. Questo lo rende particolarmente adatto per applicazioni multi-threaded o asincrone, dove l'integrità e la coerenza delle operazioni devono essere garantite.

Rust offre inoltre strumenti per la gestione efficiente delle regex in termini di compilazione e caching. Le espressioni regolari vengono compilate in automi a stati finiti prima dell'esecuzione, un processo che può risultare relativamente costoso in termini di tempo. Tuttavia, è possibile ridurre l'overhead associato alla compilazione ripetuta delle stesse regex attraverso l'uso di caching o tecniche di pre-compilazione. Questi meccanismi permettono di riutilizzare regex compilate precedentemente, riducendo il costo computazionale nelle esecuzioni successive.

In sintesi, le espressioni regolari rappresentano un esempio di equilibrio tra potenza espressiva e ottimizzazione delle prestazioni, all'interno di un ecosistema linguistico che privilegia la sicurezza e la prevedibilità. Rust implementa le regex in

modo sicuro, efficiente e altamente performante, mantenendo al contempo una sintassi familiare e funzionalità robuste, pur limitando alcune caratteristiche per garantire coerenza e prevedibilità nell'esecuzione.

### *Qualificatori, caratteri e operatori nelle regex*

Il crate *regex* in Rust rappresenta una delle librerie principali per l'utilizzo delle espressioni regolari. Progettato per offrire un equilibrio tra potenza espressiva e prestazioni, come anticipato precedentemente, questo *crate* implementa un motore basato su *automi a stati finiti* (DFA), garantendo una complessità temporale lineare rispetto alla lunghezza dell'input, indipendentemente dalla complessità del pattern utilizzato. Tale approccio si distingue dall'uso di motori di *backtracking*, presenti in altri linguaggi, che possono soffrire di comportamenti non deterministici o di fenomeni relativi.

L'uso di regex in Rust permette la definizione e l'esecuzione di pattern sofisticati per il matching delle stringhe, sfruttando una sintassi standard e ben consolidata, ma con l'efficienza e la sicurezza. Qui le espressioni regolari sono progettate per essere thread-safe e possono essere utilizzate in ambienti concorrenti senza incorrere in problemi legati alla gestione condivisa delle risorse. Ciò avviene grazie all'attenzione del linguaggio alla gestione della memoria e all'integrità dei dati in contesti multi-threaded.

Una delle caratteristiche principali è la gestione dei qualificatori, che regolano la quantità di occorrenze richieste per un determinato pattern. Questi consentono di specificare con precisione se una determinata sequenza debba apparire un numero esatto di volte, almeno una volta o anche zero volte. Perciò sono essenziali per creare pattern flessibili e per adattare le espressioni regolari a una varietà di scenari, dove la quantità di ripetizioni gioca un ruolo cruciale.

I caratteri speciali costituiscono il cuore delle espressioni regolari, consentendo di esprimere operazioni complesse con una sintassi compatta. Ogni carattere ha un significato specifico: alcuni indicano posizioni particolari all'interno di una stringa,

come l'inizio o la fine, mentre altri rappresentano caratteri jolly che possono corrispondere a un intervallo indefinito di caratteri. Questi metacaratteri permettono di definire pattern che si adattano a variabili condizioni all'interno delle stringhe di input.

Gli operatori svolgono una funzione altrettanto importante nell'elaborazione delle espressioni regolari. Sono utilizzati per definire l'alternanza tra più espressioni, per raggruppare parti di un'espressione o per creare classi di caratteri. L'operatore di alternanza, ad esempio, consente di scegliere tra due o più possibilità, mentre i raggruppamenti servono a strutturare in modo ordinato le espressioni complesse, permettendo anche la cattura di sottostringhe per un'analisi più approfondita. Le classi di caratteri, che possono essere positive o negative, permettono di specificare insiemi di caratteri accettabili o inaccettabili, rispettivamente.

Le espressioni regolari, tuttavia, presentano anche alcune limitazioni rispetto a quelle implementate in altri linguaggi. In particolare, il crate *regex* non supporta *lookahead* e *lookbehind*, caratteristiche avanzate spesso utilizzate per eseguire matching basati su contesti specifici senza includere quei contesti nel risultato. Questa scelta progettuale riflette l'attenzione di Rust all'efficienza e alla prevedibilità del tempo di esecuzione, poiché queste operazioni tendono a introdurre comportamenti complessi e meno prevedibili.

Un altro aspetto rilevante del crate *regex* è la gestione della compilazione delle espressioni regolari. Poiché la compilazione di una regex può essere costosa, è importante tenere conto del fatto che è possibile precompilare le espressioni e riutilizzarle in modo efficiente. La creazione di una regex è una fase separata dalla sua esecuzione, e una volta compilata, può essere eseguita ripetutamente senza dover essere ricostruita ogni volta. Ciò riduce il carico computazionale nelle applicazioni che richiedono l'uso intensivo di espressioni regolari.

Per iniziare a utilizzare le espressioni regolari in Rust, è necessario aggiungere il *crate regex* al tuo progetto. Puoi farlo aggiungendo la seguente riga nel file *Cargo.toml* (di Cargo parleremo approfonditamente nel prossimo capitolo):

```
 [dependencies]
regex = "1"
```

Nel codice Rust, importa il *crate*:

```
extern crate regex;
use regex::Regex;
```

Una volta fatto ciò, puoi iniziare a lavorare con le espressioni regolari. Ora esaminiamo i principali costrutti:

*Il carattere jolly* **.** punto corrisponde a qualsiasi carattere eccetto il carattere di nuova linea. Per verificare se una stringa corrisponde a un pattern, si può utilizzare il metodo *is_match*:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new("c.t").unwrap();
    let text = "cat";
    println!("{}", pattern.is_match(text)); // True, "c.t" corrisponde a "cat"
}
```

*Il quantificatore* ***** equivale a zero o più occorrenze del carattere precedente. Ciò significa che la parte del pattern prima di * può apparire nessuna o più volte nella stringa:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new("a*b").unwrap();
    let text = "aaab";
    println!("{}", pattern.is_match(text)); // True, "a*b" corrisponde a "aaab"
}
```

*Il quantificatore* **+** rappresenta una o più occorrenze del carattere precedente. In altre parole, quest'ultimo deve apparire almeno una volta:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new("a+b").unwrap();
    let text = "aaab";
    println!("{}", pattern.is_match(text)); // True, "a+b" corrisponde a "aaab"
}
```

*Il quantificatore* ***?*** indica che il carattere precedente può apparire zero o una volta. Questo è utile per gestire variazioni minime in una stringa:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new("colou?r").unwrap();
    let text = "color";
    println!("{}", pattern.is_match(text)); // True, "colou?r" corrisponde a "color"
}
```

*L'operatore di alternanza* **|** consente di scegliere tra due o più alternative. Si può usare per creare pattern che corrispondono a una delle espressioni separate:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new("cane|gatto").unwrap();
    let text = "gatto";
    println!("{}", pattern.is_match(text)); // True, "gatto" corrisponde a "cane|gatto"
}
```

*Le classi di caratteri* ***[ ]*** definiscono una classe di caratteri, corrispondente a uno qualsiasi di quelli contenuti al suo interno. Possono essere utilizzate anche per specificare intervalli di caratteri:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new("[abc]").unwrap();
    let text = "a";
    println!("{}", pattern.is_match(text)); // True, "[abc]" corrisponde a "a"
}
```

*I gruppi* ***( )*** servono per raggruppare parte di un'espressione regolare e per creare gruppi di cattura. Questi gruppi permettono di isolare porzioni specifiche di una stringa:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new("(abc)+").unwrap();
    let text = "abcabc";
    println!("{}", pattern.is_match(text)); // True, "(abc)+" corrisponde a "abcabc"
}
```

*L'ancora* ***^*** corrisponde all'inizio di una stringa. Si usa per verificare se una stringa inizia con un determinato pattern:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new("^Inizio").unwrap();
    let text = "Inizio di una stringa";
    println!("{}", pattern.is_match(text)); // True, "Inizio" corrisponde all'inizio della
stringa
}
```

*L'ancora* ***$*** invece, corrisponde alla fine della stringa:

```
extern crate regex;
use regex::Regex;
```

```
fn main() {
    let pattern = Regex::new("fine$").unwrap();
    let text = "Questa è la fine";
    println!("{}", pattern.is_match(text)); // True, "fine$" corrisponde alla fine della stringa
}
```

*Il metacarattere* **** viene utilizzato per sfuggire ai metacaratteri e trattarli come caratteri letterali. Ad esempio, per corrispondere a un punto letterale, si deve usare \\..:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new("\\.").unwrap();
    let text = "file.txt";
    println!("{}", pattern.is_match(text)); // True, "\\." corrisponde al punto letterale nel testo
}
```

*Intervalli* ***{}*** definiscono un intervallo per i quantificatori. Ad esempio, a{2,4} corrisponde a due, tre o quattro occorrenze della lettera *a*:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new("a{2,4}").unwrap();
    let text = "aaa";
    println!("{}", pattern.is_match(text)); // True, corrisponde a "aaa"
}
```

*Classi negate* ***[^ ]*** corrisponde a qualsiasi carattere che non sia all'interno delle parentesi quadre. Questo crea una classe di caratteri negata:

```
extern crate regex;
use regex::Regex;

fn main() {
```

```
    let pattern = Regex::new("[^abc]").unwrap();
    let text = "d";
    println!("{}", pattern.is_match(text)); // True, corrisponde a "d" (qualsiasi carattere tranne a, b, o c)
}
```

### *Sequenze di escape per caratteri predefiniti*

Rust supporta alcune delle sequenze di escape comuni come \d, \w, \s, ecc. Tuttavia, come detto, non supporta \b (confine di parola) e \B (non confine di parola). Non esistono le espressioni per i lookahead e lookbehind ((?=expr), (?<=expr), (?<!expr), (?:)), poiché non sono implementate queste funzionalità, come in altri linguaggi. Ecco un esempio per le sequenze di escape che sono supportate:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"\d\D\w\W\s\S").unwrap();
    let text = "1a A";
    println!("{}", pattern.is_match(text)); // True, corrisponde a "1a A"
}
```

Per quanto riguarda i qualificatori {n}, {n,} e {n,m}, questi sono pienamente supportati. Per il pattern che cerca esattamente tre occorrenze di un carattere, possiamo scrivere:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"a{3}").unwrap();
    let text = "aaa";
    println!("{}", pattern.is_match(text)); // True, "a{3}" corrisponde a "aaa"
}
```

Analogamente, per {n,} (n o più occorrenze) e {n,m} (tra n e m occorrenze), il

codice sarebbe:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"a{2,}").unwrap();
    let text = "aaaa";
    println!("{}", pattern.is_match(text)); // True, "a{2,}" corrisponde a "aaaa"
}

extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"a{2,4}").unwrap();
    let text = "aaa";
    println!("{}", pattern.is_match(text)); // True, "a{2,4}" corrisponde a "aaa"
}
```

Rust supporta anche gli intervalli di caratteri e le combinazioni di classi di caratteri. Per esempio, un pattern che corrisponde a una qualsiasi lettera minuscola può essere scritto così:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"[a-z]").unwrap();
    let text = "m";
    println!("{}", pattern.is_match(text)); // True, corrisponde a "m"
}
```

Allo stesso modo, possiamo combinare intervalli di caratteri per cercare lettere maiuscole, minuscole o cifre:

```
extern crate regex;
use regex::Regex;

fn main() {
```

```
    let pattern = Regex::new(r"[a-zA-Z0-9]").unwrap();
    let text = "A";
    println!("{}", pattern.is_match(text)); // True, corrisponde a "A"
}
```

Il crate *regex* in Rust non supporta il *lookahead* ((?=expr)), il *lookbehind* ((?<=expr)), né i loro equivalenti negativi ((?!expr), (?<!expr)). Questo è dovuto al fatto che il motore delle espressioni regolari è progettato per essere più efficiente e prevedibile, evitando costrutti che potrebbero introdurre complessità non lineare. Di conseguenza, questi pattern devono essere riscritti utilizzando altre tecniche, come il post-processing dei risultati del matching o frammentando le espressioni regolari.
Vediamo adesso alcuni esempi pratici con le sequenze di escape, che come detto, permettono di rappresentare insiemi di caratteri comuni per caratteri predefiniti:

**\d**: corrisponde a una cifra (0-9).
**\D**: corrisponde a qualsiasi carattere che non sia una cifra.
**\w**: corrisponde a un carattere alfanumerico o underscore (a-z, A-Z, 0-9, _).
**\W**: corrisponde a qualsiasi carattere che non sia alfanumerico o underscore.
**\s**: corrisponde a un carattere di spazio bianco (spazio, tabulazione, nuova linea).
**\S**: corrisponde a qualsiasi carattere che non sia uno spazio bianco.

Codice:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"\d\D\w\W\s\S").unwrap();
    let text = "3a_! bC";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

Nel codice, \d: corrisponde a 3.
\D: corrisponde a a (non cifra).
\w: corrisponde a _ (underscore).

\W: corrisponde a ! (non alfanumerico).
\s: corrisponde a uno spazio.
\S: corrisponde a b (non spazio bianco).

- Qualificatori {n}, {n,}, {n,m}, che specificano il numero di occorrenze di un carattere o gruppo:

**{n}**: esattamente n occorrenze.
**{n,}**: almeno n occorrenze.
**{n,m}**: almeno n e al massimo m occorrenze.

a) Esattamente n occorrenze {n}:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"a{3}").unwrap();
    let text = "aaa";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

Il pattern a{3} cerca esattamente tre a consecutive.

b) Almeno n occorrenze {n,}:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"a{2,}").unwrap();
    let text = "aaaaa";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

Il pattern a{2,} cerca almeno due *a* consecutive. Il testo ne contiene cinque, quindi corrisponde.

c) Tra n e m occorrenze {n,m}:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"a{2,4}").unwrap();
    let text = "aaaa";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

Il pattern a{2,4} corrisponde a un minimo di due e un massimo di quattro a consecutive. Il testo ha quattro a, quindi è valido.

- Intervalli e classi di caratteri, i quali definiscono insiemi specifici di caratteri da corrispondere.

a) Intervallo di lettere minuscole [a-z]

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"[a-z]").unwrap();
    let text = "m";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

Il pattern [a-z] corrisponde a qualsiasi lettera minuscola da *a* a *z*.

b) Combinazione di intervalli [a-zA-Z0-9]

```
extern crate regex;
use regex::Regex;

fn main() {
```

```
    let pattern = Regex::new(r"[a-zA-Z0-9]").unwrap();
    let text = "9";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

Il pattern [a-zA-Z0-9] corrisponde a qualsiasi lettera maiuscola, minuscola o cifra.

- Caratteri speciali e metacaratteri, eseguire l'escape con \.

a) Punto . come carattere letterale

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"\.").unwrap();
    let text = "file.txt";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

Il pattern \. corrisponde a un punto letterale. Senza l'escape, . corrisponderebbe a qualsiasi carattere.

- Gruppi di cattura, permettono di estrarre parti della stringa che corrispondono a un sotto-pattern:

a) Cattura di gruppi numerici

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"(\w+)@(\w+\.\w+)").unwrap();
    let text = "email@example.com";
    if let Some(caps) = pattern.captures(text) {
        let username = &caps[1];
        let domain = &caps[2];
```

```
        println!("Username: {}, Dominio: {}", username, domain);
    }
}
```

(\w+): cattura una sequenza di uno o più caratteri alfanumerici (il nome utente).
@: corrisponde al carattere @.
(\w+\.\w+): cattura il dominio (esempio example.com).

Il crate regex non supporta *lookahead* ((?=...)) e *lookbehind* ((?<=...)), sia positivi che negativi. Questo significa che pattern come *foo(?=bar)* non funzioneranno. In alternativa, se devi corrispondere *foo* solo se seguito da *bar*, puoi cercare *foobar* e utilizzare un gruppo di cattura per estrarre *foo*:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"(foo)bar").unwrap();
    let text = "foobar";
    if let Some(caps) = pattern.captures(text) {
        println!("Trovato: {}", &caps[1]); // Stampa "Trovato: foo"
    }
}
```

Il pattern *(foo)bar* corrisponde a *foobar*, *foo* viene catturato nel gruppo 1.

- Uso di ancore ^ e $, servono per specificare l'inizio ^ e la fine $ di una stringa:

a) Inizio della stringa ^

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"^Inizio").unwrap();
    let text = "Inizio di una frase";
    println!("{}", pattern.is_match(text)); // Stampa "true"
```

```
}
```

Il pattern ^*Inizio* corrisponde se la stringa inizia con *Inizio*.

b) Fine della stringa $

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"fine$").unwrap();
    let text = "Questo è la fine";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

Il pattern *fine$* corrisponde se la stringa termina con *fine.*

- Negazione nelle classi di caratteri [^ ], possono essere negate utilizzando ^ all'interno delle parentesi quadre:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"[^aeiou]").unwrap();
    let text = "b";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

Il pattern [^aeiou] corrisponde a qualsiasi carattere che non sia una vocale minuscola.

Ripetizioni con *, +, ?

*: zero o più occorrenze.
+: una o più occorrenze.
?: zero o una occorrenza.

a) Zero o più occorrenze *

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"ba*r").unwrap();
    let text = "bar";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

a* corrisponde a zero o più *a*. Quindi bar, baar, baaar, ecc.

b) Una o più occorrenze +

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"ba+r").unwrap();
    let text = "baar";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

a+ corrisponde a una o più *a*. Quindi bar non corrisponderebbe, ma baar sì.

c) Zero o una occorrenza ?

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"colou?r").unwrap();
    let text = "color";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

u? indica che *u* può apparire zero o una volta. Quindi sia *color* che *colour*

corrispondono.
I raggruppamenti possono essere utilizzati per applicare qualificatori a un'intera espressione:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = Regex::new(r"(ab){2}").unwrap();
    let text = "abab";
    println!("{}", pattern.is_match(text)); // Stampa "true"
}
```

Il pattern (ab){2} cerca due occorrenze consecutive di *ab*.

Nonostante alcune limitazioni rispetto ad altre implementazioni, il crate *regex* offre strumenti potenti per la manipolazione e l'analisi delle stringhe, mantenendo al contempo l'efficienza e la sicurezza tipiche del linguaggio.

### *Le sequenze speciali o complesse*

Le sequenze di escape come \n, \r, \t, ecc. possono essere utilizzate direttamente nel pattern. Ad esempio, per trovare una nuova linea (\n):

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"\n";
    let text = "prima riga\nseconda riga";
    let re = Regex::new(pattern).unwrap();

    if re.is_match(text) {
        println!("Trovato: nuova linea");
    }
}
```

- *Sequenze di escape Unicode e ASCII*: Rust supporta le espressioni regolari Unicode. Puoi usare \u per caratteri Unicode e \x per ASCII. Esempio per il carattere Unicode é (U+00E9):

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"\u{00E9}";
    let text = "café";
    let re = Regex::new(pattern).unwrap();

    if re.is_match(text) {
        println!("Trovato: carattere 'é'");
    }
}
```

Gli Unicode si scrivono con \u{xxxx}, dove xxxx è l'esadecimale.

- *Ancoraggi*: inizio e fine della stringa, confini di parola:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"\bword\b";
    let text = "word in a sentence";
    let re = Regex::new(pattern).unwrap();

    if re.is_match(text) {
        println!("Trovato: parola isolata 'word'");
    }
}
```

- *Cattura di gruppi e riferimenti retrospettivi*:

```
extern crate regex;
use regex::Regex;

fn main() {
```

```
    let pattern = r"(\w+)\s\1";
    let text = "hello hello";
    let re = Regex::new(pattern).unwrap();

    if re.is_match(text) {
        println!("Trovata parola ripetuta");
    }
}
```

- *Gruppi di cattura non numerati*: puoi definire un gruppo che non cattura con (?:...):

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"(?:abc)+";
    let text = "abcabc";
    let re = Regex::new(pattern).unwrap();

    if re.is_match(text) {
        println!("Corrispondenza trovata: 'abcabc'");
    }
}
```

- *Alternanza*: operatore | :

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"cane|gatto";
    let text = "gatto";
    let re = Regex::new(pattern).unwrap();

    if re.is_match(text) {
        println!("Corrisponde a 'gatto' o 'cane'");
    }
}
```

- *Intervalli di caratteri*:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"[a-z]";
    let text = "g";
    let re = Regex::new(pattern).unwrap();

    if re.is_match(text) {
        println!("Corrisponde a una lettera minuscola");
    }
}
```

- *Quantificatori Greedy e Lazy*:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"a+?";
    let text = "aaaa";
    let re = Regex::new(pattern).unwrap();

    if re.is_match(text) {
        println!("Trovata la prima 'a' (lazy)");
    }
}
```

- *Righe e parole*: sono supportati ancoraggi come \A (inizio stringa), \Z (fine stringa):

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"\AInizio";
    let text = "Inizio di una stringa";
    let re = Regex::new(pattern).unwrap();

    if re.is_match(text) {
        println!("Corrisponde all'inizio della stringa");
```

```
    }
}
```

- *Regex Anchors*: oltre agli ancoraggi standard come ^ (inizio della stringa) e $ (fine della stringa), ci sono alcune specificità in Rust relative a come queste ancore funzionano con input multilinea, o *Multiline Matching* ((?m)) si può abilitare con (?m) per far sì che ^ e $ corrispondano all'inizio e alla fine di ogni riga, piuttosto che dell'intera stringa:

```
let pattern = r"(?m)^\w+";  // Corrisponde a parole all'inizio di ogni riga
```

- *Unicode Handling*: tramite il crate *regex*, abbiamo supporto avanzato per operare su stringhe Unicode. Esistono diverse classi di caratteri che vi funzionano:

**\p{}**: supporta classi Unicode generali, come \p{L} (lettere) e \p{N} (numeri). È utile per lavorare con alfabeti non latini:

```
let pattern = r"\p{L}+";
```

**\P{}**: è l'operatore opposto di \p{}, e corrisponde a qualsiasi carattere che non appartiene alla classe Unicode specificata:

```
let pattern = r"\P{L}+";
```

- *Flag di compilazione e opzioni avanzate*: Rust permette di utilizzare vari flag che modificano il comportamento del matching:

- *Case Insensitive ((?i))*: come in molti altri linguaggi, puoi rendere il confronto case-insensitive.

```
let pattern = r"(?i)rust";  // Corrisponde sia a "rust" che a "RUST"
```

- *Dot Matches Newline ((?s))*: normalmente il punto . non corrisponde ai caratteri di nuova linea. Questo flag fa sì che . corrisponda anche a \n.

```
let pattern = r"(?s).+";  // Corrisponde a tutto, inclusi i caratteri newline
```

- *Free-spacing mode ((?x))*: permette di inserire spazi e commenti nel pattern per renderlo più leggibile, ignorando gli spazi bianchi:

```
let pattern = r"(?x) \d+ \s* # Numero e spazi";
```

- *Performance e Streaming*: il crate *regex* è stato progettato per essere estremamente efficiente e supporta la compilazione in tempo reale, ma esistono altre tecniche di ottimizzazione, specialmente per operazioni su grandi dataset, ad esempio può essere applicata in modalità "streaming", che consente di eseguire il matching su input molto grandi senza dover caricare tutto in memoria. Questa funzionalità è utile per file di grandi dimensioni o flussi di dati continui.
Possiamo anche usare le funzioni *captures_iter()* e *find_iter()* per ottenere risultati mentre elabori parti della stringa:

```
let text = "foo123bar456";
let re = Regex::new(r"\d+").unwrap();
for cap in re.find_iter(text) {
    println!("Trovato: {}", cap.as_str());
}
```

- *Gestione degli Errori*: essendo Rust un linguaggio fortemente tipizzato, gestisce molto bene gli errori. La creazione di un pattern regex può fallire (ad esempio, se contiene errori di sintassi), quindi è comune gestire questi errori in modo esplicito:

```
match Regex::new(r"(\d{3") {
    Ok(re) => println!("Regex compilata correttamente"),
    Err(err) => println!("Errore nella regex: {}", err),
}
```

- *Capturing Group Named*: oltre ai classici gruppi numerici, Rust supporta anche i gruppi nominati. Questo è utile quando hai più gruppi e vuoi riferirti a loro tramite nomi invece di numeri:

```
let pattern = r"(?P<num>\d+)";
let text = "Il numero è 123";
let re = Regex::new(pattern).unwrap();
if let Some(caps) = re.captures(text) {
    println!("Trovato: {}", &caps["num"]);
}
```

- *Substitutions con replace*: il metodo *replace* permette di sostituire i *match* di una regex con una stringa o il risultato di una *closure*:

Sostituzione semplice:

```
let re = Regex::new(r"\d+").unwrap();
let result = re.replace("123 foo", "#");
println!("{}", result);  // "# foo"
```

Sostituzione con una closure:

```
let re = Regex::new(r"\d+").unwrap();
let result = re.replace_all("123 foo 456", |caps: &regex::Captures| {
    format!("[{}]", &caps[0])
});
println!("{}", result);  // "[123] foo [456]"
```

- *Handling Large Datasets*: Rust è ben noto per la sua gestione della memoria sicura ed efficiente. Nel caso di grandi dataset, le espressioni regolari possono essere ottimizzate per lavorare con *&str* invece di *String* per evitare inutili allocazioni. Inoltre, è possibile lavorare su buffer di dati o flussi per evitare di caricare grandi quantità di dati in memoria contemporaneamente.

***Metodi e Regex***

In Rust, non esiste il concetto di "regex compilata" come in altri linguaggi, ma il *crate regex* è già ottimizzato per prestazioni elevate e si comporta in modo simile.

- *re.find*: puoi utilizzarlo per ottenere la prima corrispondenza di un pattern:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"\d+";
    let input = "Ci sono 123 numeri qui";
    let re = Regex::new(pattern).unwrap();

    if let Some(mat) = re.find(input) {
        println!("Trovato: {}", mat.as_str());
    }
}
```

*find* restituisce la prima corrispondenza del pattern nella stringa *input*.

- *re.find_iter*: per ottenere tutte le corrispondenze, con questo metodo si può iterare su di esse:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"\d+";
    let input = "Ci sono 123 numeri, 456 e 789 qui";
    let re = Regex::new(pattern).unwrap();

    for mat in re.find_iter(input) {
        println!("Trovato: {}", mat.as_str());
    }
}
```

*find_iter* restituisce un iteratore su tutte le corrispondenze del pattern.

- *Regex.Replace*: per sostituire tutte le corrispondenze, utilizza il metodo *replace_all*:

```
extern crate regex;
use regex::Regex;
```

```
fn main() {
    let pattern = r"\d+";
    let input = "Ci sono 123 numeri qui";
    let replacement = "#";
    let re = Regex::new(pattern).unwrap();

    let result = re.replace_all(input, replacement);
    println!("{}", result); // Output: "Ci sono # numeri qui"
}
```

*replace_all* sostituisce tutte le corrispondenze del pattern con la stringa il rimpiazzo definito.

- *re.split*: il metodo *split* suddivide una stringa in base a un pattern:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"\d+";
    let input = "Uno1Due2Tre3";
    let re = Regex::new(pattern).unwrap();

    for part in re.split(input) {
        println!("{}", part); // Output: "Uno", "Due", "Tre"
    }
}
```

*split* divide la stringa input in base al pattern.

- *Regex.IsMatch*: verifica se una stringa corrisponde a un pattern, puoi usare il metodo *is_match*:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"\d+";
```

```
    let input = "Ci sono 123 numeri qui";
    let re = Regex::new(pattern).unwrap();

    let is_match = re.is_match(input);
    println!("{}", is_match); // Output: true
}
```

*is_match* restituisce *true* se il pattern corrisponde alla stringa, altrimenti *false*.

- *"Regex compilata"*: in Rust non esiste un'opzione diretta per la compilazione come in altri linguaggi, ma puoi ottimizzare il riutilizzo della regex dichiarandola una sola volta all'inizio del programma. In ogni caso, il *crate regex* è già progettato per essere molto efficiente:

```
extern crate regex;
use regex::Regex;

fn main() {
    let pattern = r"\d+";
    let input = "Ci sono 123 numeri qui";
    let re = Regex::new(pattern).unwrap();

    // Esegui la regex più volte
    for _ in 0..1000 {
        if let Some(mat) = re.find(input) {
            println!("Trovato: {}", mat.as_str());
        }
    }
}
```

In questo esempio, la regex viene compilata una volta con *Regex::new* e riutilizzata più volte all'interno di un ciclo.

- *Confronto delle prestazioni*: non c'è una distinzione esplicita tra regex "compilata" e "non compilata", ma possiamo misurare le prestazioni con un ciclo che esegue la regex molte volte:

```
extern crate regex;
```

```
use regex::Regex;
use std::time::Instant;

fn main() {
    let pattern = r"\d+";
    let input = "Ci sono 123 numeri qui";
    let re = Regex::new(pattern).unwrap();

    // Misura il tempo di esecuzione per regex normale
    let start = Instant::now();
    for _ in 0..100000 {
        let _ = re.find(input);
    }
    let duration = start.elapsed();
    println!("Tempo di esecuzione: {:?}", duration);
}
```

In questo esempio, usiamo *Instant::now()* per misurare il tempo di esecuzione di una regex eseguita più volte. Sebbene non ci sia un'opzione di compilazione, il crate *regex* è altamente ottimizzato.

Abbiamo dunque visto come vengono gestite le espressioni regolari in modo sicuro ed efficiente, permettendo operazioni come la corrispondenza, sostituzione e suddivisione delle stringhe. Tuttavia, è importante notare che non sono supportate tutte le funzionalità avanzate presenti in altre implementazioni (come *lookbehind*), ma per la maggior parte delle operazioni comuni il linguaggio offre un'ottima soluzione.

### *Quiz & Esercizi*

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

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

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

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

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

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

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).

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

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

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

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

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

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

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

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

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

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

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

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

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

21) Scrivere un programma che verifichi se una stringa rappresenta un indirizzo MAC valido.

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

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.

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.

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.

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.

# # 7 - Moduli, Crate e Cargo

*Moduli e Crate*
*Cargo*
*Workspace*

I moduli e la gestione delle dipendenze tramite *crate* (unità di compilazione, collegamento e distribuzione) sono essenziali per l'organizzazione del codice e la costruzione di progetti scalabili e manutenibili. Infatti servono a organizzare e strutturare il codice in unità logiche. Ogni modulo definisce un contesto di visibilità e incapsulamento, consentendo di separare diverse funzionalità all'interno di un progetto. Permettono di raggruppare funzioni, strutture, tipi e costanti, facilitando la suddivisione del codice in blocchi più gestibili. Al loro interno è possibile definire delle sotto-gerarchie, chiamate sotto-moduli, che aiutano a riflettere una struttura logica più profonda e complessa. Questo sistema permette di controllare la visibilità degli elementi: quelli all'interno di un modulo sono, di default, privati al modulo stesso, ma possono essere resi pubblici, se necessario, utilizzando delle apposite dichiarazioni. La modularità in Rust consente quindi di creare una chiara separazione delle responsabilità e una gestione controllata delle dipendenze interne tra le diverse parti di un programma.
Un aspetto strettamente correlato ai moduli è il concetto di crate, che in Rust è l'unità fondamentale di compilazione e di distribuzione. Può essere di due tipi: un crate binario, che rappresenta un programma eseguibile, oppure un crate di libreria, che contiene del codice riutilizzabile e condivisibile. *I crate permettono di riutilizzare codice esterno e di organizzare le dipendenze in modo efficiente*,

fornendo un meccanismo standardizzato per includere e gestire librerie di terze parti o sviluppare codice in moduli separati e poi integrarli. Ogni crate ha il proprio spazio dei nomi, che garantisce l'isolamento e ne previene le collisioni, favorendo una maggiore manutenibilità e modularità del codice.
Un crate può essere composto da più moduli, che ne formano la struttura interna, ma dall'esterno appare come una singola entità. Il sistema di moduli all'interno del crate consente di mantenere organizzato il codice, mentre il crate stesso rappresenta un'unità di distribuzione che può essere facilmente condivisa e integrata in altri progetti.
La gestione delle dipendenze attraverso i crate segue un approccio rigoroso. Ognuno può dichiarare delle dipendenze verso altri crate. Questo meccanismo consente di evitare la duplicazione del codice e favorisce la riusabilità di soluzioni consolidate. Tuttavia, Rust si assicura che le dipendenze siano ben definite e che non ci siano ambiguità o conflitti nella risoluzione di esse. Il sistema dei crate promuove una loro gestione sia robusta che scalabile, poiché include strumenti per il controllo delle versioni, la risoluzione dei conflitti e la gestione delle dipendenze transitive, ovvero le dipendenze di altre dipendenze.
Tutto ciò trova il suo pieno potenziale attraverso Cargo, lo abbiamo brevemente anticipato nel primo capitolo. Come visto, non è semplicemente un gestore di pacchetti, ma un vero e proprio strumento integrato che si occupa della gestione dell'intero ciclo di vita di un progetto, dall'impostazione iniziale alla distribuzione, passando per la compilazione, la gestione delle dipendenze e l'esecuzione di test.
In questo contesto, Cargo è fondamentale per una gestione completa. Ogni progetto viene inizializzato tramite il tool, che genera la struttura di base, inclusi i file di configurazione necessari per definire le dipendenze e i metadati. Il file chiave è il *Cargo.toml*, che specifica le dipendenze del progetto, le versioni e altre informazioni cruciali come i dettagli sul crate stesso (ad esempio, nome, descrizione e licenza). Questo file funge da contratto formale tra il progetto e le sue dipendenze, garantendo che il codice sia sempre costruito in un ambiente

coerente e ripetibile.
Quando un progetto dichiara una dipendenza da un crate, Cargo si occupa automaticamente di recuperarlo, gestire la versione corretta e risolvere eventuali dipendenze transitive. Questo significa che, qualora un crate dipenda a sua volta da altri, *Cargo risolverà automaticamente tali dipendenze*, assicurandosi che siano compatibili tra loro. Inoltre, Cargo fornisce un sistema di versionamento semantico, che consente di specificare in modo preciso le versioni dei crate con cui il progetto è compatibile. Questo approccio permette di evitare conflitti di versione, rendendo la gestione delle dipendenze molto affidabile.
Un altro aspetto chiave di Cargo è la gestione della compilazione. Rust ne ha uno estremamente rigoroso, che esegue controlli statici approfonditi, e Cargo semplifica la gestione di questo processo, orchestrando la compilazione dei moduli e dei crate secondo un ordine determinato dalle dipendenze. Ottimizza anche il processo di compilazione con l'uso di una cache, ricompilando solo le parti del progetto che hanno subito modifiche, riducendo così i tempi di build.
Cargo gestisce anche i profili di compilazione, permettendo di configurarne diversi per scenari differenti, come la modalità di sviluppo o quella di rilascio. Questi profili offrono un controllo preciso su aspetti come le ottimizzazioni e i controlli di debug, garantendo che il progetto sia sempre costruito nel modo più efficiente possibile in base alle esigenze.
Oltre alla gestione delle dipendenze e della compilazione, Cargo fornisce funzionalità avanzate come il supporto integrato per i test. Ogni crate può includerne, ed esiste un meccanismo semplice per eseguirli e verificarne l'accuratezza. Questa integrazione tra codice, test e gestione delle dipendenze è un altro esempio della filosofia di Rust, che mette al centro la robustezza e la sicurezza del codice, riducendo il rischio di errori e garantendo la coerenza del progetto nel tempo.
In sintesi, Cargo rappresenta il cuore operativo di Rust, consentendo agli sviluppatori di concentrarsi sullo sviluppo senza preoccuparsi degli aspetti tecnici

legati alla gestione delle dipendenze, alla compilazione o all'esecuzione dei test. In questo modo, è facilitata la creazione di progetti ben strutturati e modulari, che possono facilmente sfruttare la potenza del sistema di moduli e crate per realizzare software affidabile e scalabile.

### *I moduli*

Come detto, i moduli rappresentano il meccanismo fondamentale per organizzare e suddividere il codice in blocchi logici separati. In pratica raggruppa funzioni, strutture, tipi, costanti e altri elementi in un contesto separato, permettendo di isolare concetti differenti in modo chiaro e mantenibile. L'utilizzo dei moduli consente di evitare collisioni di nomi e di gestire la visibilità degli elementi, controllando l'accesso da altre parti del programma. Questo approccio permette di creare codice più organizzato, leggibile e sicuro, con una chiara separazione delle responsabilità.

Per comprendere l'utilità dei moduli, possiamo immaginare un progetto di grandi dimensioni in cui ogni parte svolge una funzione specifica. Ad esempio, un'applicazione potrebbe avere un modulo dedicato alla gestione dei dati, uno per l'interfaccia utente, e un altro per le operazioni di rete. I moduli aiutano a mantenere separato il codice relativo a ciascuna parte, migliorando la manutenibilità e facilitando la collaborazione su progetti complessi.

La visibilità degli elementi all'interno di un modulo è un aspetto cruciale. Di default, gli elementi dichiarati in un modulo sono privati: *ciò significa che possono essere utilizzati solo all'interno del modulo stesso e non sono accessibili dall'esterno*. Questa scelta riflette il principio dell'incapsulamento, in cui i dettagli interni di un modulo rimangono nascosti e solo le interfacce pubbliche vengono esposte. Quando si desidera rendere un elemento accessibile al di fuori del modulo, è necessario utilizzare la parola chiave *pub*. Un elemento dichiarato come pubblico può essere importato e utilizzato in altri moduli o parti del codice.

Ad esempio, possiamo dichiarare una funzione all'interno di un modulo e

specificarne la visibilità. Se la funzione è privata, sarà utilizzabile solo all'interno del modulo:

```
mod gestione_dati {
    fn salva_dati() {
        println!("Salvataggio dati in corso...");
    }
}
```

La funzione *salva_dati* è privata e non può essere invocata al di fuori del modulo *gestione_dati*. Se volessimo rendere questa funzione pubblica, così da poterla chiamare da altre parti del programma, dovremmo utilizzare la parola chiave *pub*:

```
mod gestione_dati {
    pub fn salva_dati() {
        println!("Salvataggio dati in corso...");
    }
}
```

In questo caso, la funzione è diventata pubblica e può essere utilizzata da qualsiasi altro modulo o parte del codice. Per accedere a *salva_dati* da un altro modulo, è necessario utilizzare il sistema di importazione di Rust, che si realizza con la parola chiave *use*. Ad esempio, se volessimo chiamare *salva_dati* da una funzione esterna al modulo, potremmo scrivere:

```
use gestione_dati::salva_dati;

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

I moduli possono essere dichiarati sia in-line, all'interno di un file, sia esternamente in file separati, mantenendo così il codice ben organizzato. Se un modulo diventa troppo grande o complesso, è possibile suddividerlo ulteriormente in sotto-moduli, che seguono la stessa logica di incapsulamento e visibilità. Ad esempio, possiamo creare un modulo esterno chiamato *gestione_dati.rs*, che

contiene la logica di gestione dei dati, e importarlo nel file principale.
La relazione tra file, directory e moduli è strettamente legata all'organizzazione del codice sorgente. Ogni file in un progetto può rappresentare un modulo, e le directory possono fungere da contenitori per moduli e sotto-moduli, permettendo di costruire gerarchie logiche che riflettono la complessità del programma. Questo meccanismo permette di suddividere progetti grandi in componenti ben definiti e di mantenere il codice organizzato in modo efficace.
Iniziamo con il concetto di modulo in un file. Ogni file .rs può rappresentare un modulo. Per esempio, supponiamo di avere un file chiamato *main.rs*, che contiene il codice principale del nostro programma. Se volessimo separare una parte della logica in un altro modulo, potremmo creare un nuovo file, come *gestione_dati.rs*. Questo file rappresenterebbe un modulo chiamato *gestione_dati* e, una volta creato, possiamo fare riferimento a esso nel file *main.rs*.
Nel file *main.rs,* potremmo importare il modulo *gestione_dati* dichiarandolo esplicitamente:

```
mod gestione_dati;

fn main() {
    gestione_dati::salva_dati();
}
```

Qui, *mod gestione_dati;* dichiara la sua esistenza e ne richiede la compilazione dal file *gestione_dati.rs*. In questo file possiamo definire funzioni, strutture e altre componenti come segue:

```
pub fn salva_dati() {
    println!("Salvataggio dati in corso...");
}
```

La funzione *salva_dati* è resa pubblica con pub affinché possa essere chiamata da *main.rs*. Questo sistema crea una relazione diretta tra i file e i moduli, dove ogni file .rs rappresenta un modulo distinto.

Quando un progetto cresce in complessità, è possibile suddividere un modulo in sotto-moduli. Questi sono definiti all'interno di un modulo genitore, e la loro struttura, come già detto, rispecchia spesso quella delle directory e dei file del progetto. Ad esempio, supponiamo di avere un modulo principale *gestione_dati* che a sua volta contiene due sotto-moduli, uno per lettura e uno per scrittura. Questa struttura potrebbe essere riflessa nella directory *gestione_dati*, che contiene due file: *lettura.rs* e *scrittura.rs.* Nel file *gestione_dati.rs*, possiamo dichiarare i sotto-moduli come segue:

```
pub mod lettura;
pub mod scrittura;
```

I sotto-moduli lettura e scrittura sono dichiarati pubblici e ciascuno di essi può contenere la propria logica. Ad esempio, nel file *lettura.rs* possiamo avere:

```
pub fn leggi_dati() {
    println!("Lettura dati in corso...");
}
```

Allo stesso modo, nel file *scrittura.rs* potremmo avere:

```
pub fn scrivi_dati() {
    println!("Scrittura dati in corso...");
}
```

Nel file *gestione_dati.rs*, il modulo principale, possiamo accedere ai sotto-moduli lettura e scrittura:

```
pub fn opera_sui_dati() {
    lettura::leggi_dati();
    scrittura::scrivi_dati();
}
```

Ora, in *main.rs*, possiamo accedere a *opera_sui_dati* attraverso l'importazione del modulo *gestione_dati*:

```
mod gestione_dati;

fn main() {
    gestione_dati::opera_sui_dati();
}
```

Il sistema di moduli consente quindi di rappresentarne la gerarchia attraverso la struttura delle directory e dei file. Ogni directory può rappresentare un modulo contenitore, e i file all'interno della directory possono rappresentare i sotto-moduli. Da qui Rust fornisce un meccanismo potente per semplificare l'accesso agli elementi dei moduli. L'uso della parola chiave *use* permette di evitare di scrivere ripetutamente il percorso completo di un elemento. Ad esempio, se volessimo utilizzare la funzione *leggi_dati* dal modulo *lettura* in *main.rs*, senza dover scrivere il percorso completo ogni volta, possiamo fare così:

```
mod gestione_dati;

use gestione_dati::lettura::leggi_dati;

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

In questo modo, possiamo accedere direttamente alla funzione *leggi_dati* senza dover specificare l'intero percorso *gestione_dati::lettura::leggi_dati* ogni volta. È anche possibile importare più elementi da un modulo utilizzando una sintassi abbreviata, ad esempio:

```
use gestione_dati::{lettura::leggi_dati, scrittura::scrivi_dati};

fn main() {
    leggi_dati();
    scrivi_dati();
}
```

Questa flessibilità nel sistema *use* permette di importare esattamente ciò di cui abbiamo bisogno, migliorando la leggibilità del codice e riducendo la necessità di

riferimenti ripetitivi.

### *Organizzazione del codice*

Abbiamo appena visto che la modularizzazione offre numerosi vantaggi, specialmente nei progetti di grandi dimensioni, dove l'organizzazione e la gestione del codice diventano fondamentali per mantenere la manutenibilità e la leggibilità. Un progetto modulare permette di suddividere il codice in parti più piccole e autonome, ognuna con una responsabilità specifica. Questo non solo migliora la chiarezza, ma facilita anche la collaborazione tra team, la risoluzione di bug e l'estensione delle funzionalità.

Un vantaggio chiave della modularizzazione è la separazione delle responsabilità. Invece di avere tutto il codice mescolato in un unico blocco, i moduli consentono di definire chiaramente compiti e ruoli distinti per ciascuna parte del progetto. Per esempio, un progetto potrebbe avere un modulo dedicato alla gestione dei dati e uno separato per l'interfaccia utente. Riprendendo gli esempi precedenti, immaginiamo di avere un modulo *gestione_dati* che si occupa esclusivamente di leggere e scrivere dati:

```
pub mod gestione_dati {
    pub mod lettura {
        pub fn leggi_dati() {
            println!("Lettura dati in corso...");
        }
    }

    pub mod scrittura {
        pub fn scrivi_dati() {
            println!("Scrittura dati in corso...");
        }
    }
}
```

Ciò mostra come la modularizzazione permetta di mantenere separata la logica relativa alla lettura e alla scrittura dei dati, rendendo più semplice la manutenzione

e l'evoluzione del codice. La separazione delle responsabilità consente, inoltre, di estendere ciascuna parte senza influenzare il funzionamento delle altre.

Un altro principio fondamentale reso possibile dalla modularizzazione è l'*incapsulamento*. Rust, per sua natura, favorisce il controllo rigoroso sulla visibilità degli elementi attraverso la parola chiave *pub*. Gli elementi privati in un modulo non sono accessibili dall'esterno, il che permette di proteggere i dettagli interni dell'implementazione, esponendo solo ciò che è necessario. Nell'esempio precedente, se le funzioni *leggi_dati* e *scrivi_dati* non fossero pubbliche, non potrebbero essere utilizzate da altri moduli. Questo controllo della visibilità garantisce che le parti interne del codice non possano essere accidentalmente modificate o invocate da altre parti del programma, proteggendo l'integrità del progetto.

Ad esempio, possiamo decidere che la funzione *opera_sui_dati* nel modulo *gestione_dati* possa chiamare *leggi_dati* e *scrivi_dati,* ma che queste funzioni siano visibili solo all'interno del modulo stesso:

```
pub mod gestione_dati {
    mod lettura {
        pub fn leggi_dati() {
            println!("Lettura dati in corso...");
        }
    }

    mod scrittura {
        pub fn scrivi_dati() {
            println!("Scrittura dati in corso...");
        }
    }

    pub fn opera_sui_dati() {
        lettura::leggi_dati();
        scrittura::scrivi_dati();
    }
}
```

In questo caso, *opera_sui_dati* è l'unica funzione pubblica esposta dal modulo

*gestione_dati*, mentre *leggi_dati* e *scrivi_dati* restano incapsulate e inaccessibili dall'esterno. Questo approccio protegge i dettagli di implementazione, permettendo di esporre solo un'interfaccia limitata e ben definita.
Quando i progetti diventano più grandi, la modularizzazione offre un chiaro vantaggio nell'organizzazione del codice. I moduli e sotto-moduli permettono di creare strutture gerarchiche che riflettono la logica del progetto, facilitando sia la navigazione che la comprensione del codice. Ogni componente del progetto può essere organizzata in base alla sua funzionalità, riducendo il rischio di duplicazione e conflitti tra nomi.
In un ipotetico progetto complesso possiamo avere una directory *src* con diverse sottodirectory che rappresentano i vari moduli. Nella directory *gestione_dati*, potremmo organizzare il nostro codice in base alle diverse operazioni sui dati:

```
src/
    gestione_dati/
        lettura.rs
        scrittura.rs
    main.rs
```

Nel file *main.rs*, possiamo importare il modulo *gestione_dati* e utilizzarne le funzionalità senza doverci preoccupare dei dettagli di implementazione dei sotto-moduli *lettura* e *scrittura*:

```
mod gestione_dati;

fn main() {
    gestione_dati::opera_sui_dati();
}
```

La modularizzazione consente quindi di gestire facilmente la complessità, specialmente in progetti con molteplici funzionalità o team di sviluppo. Ogni modulo può essere sviluppato, testato e mantenuto separatamente, riducendo il rischio di errori e migliorando la produttività.

### *Introduzione al Crate*

In Rust, il *crate* rappresenta l'unità fondamentale di compilazione e distribuzione. Ogni programma Rust, sia esso un'applicazione o una libreria, è contenuto all'interno di un *crate,* il quale non è altro che un pacchetto che contiene del codice organizzato in moduli. Durante la compilazione, Rust lo gestisce come un'unità indivisibile, generando un file eseguibile o una libreria a partire da esso.

Esistono due principali tipologie di *crate*: *binari e librerie*. Un *crate* binario è un'unità eseguibile autonoma che contiene un punto di ingresso, generalmente la funzione *main*. Questi sono usati per costruire applicazioni vere e proprie che possono essere eseguite dal sistema operativo. *Un crate di libreria, invece, non produce un eseguibile, ma contiene codice riutilizzabile che può essere importato e utilizzato da altri crate, offrendo funzioni, strutture o moduli predefiniti*.

Un crate binario tipicamente include il file *main.rs* che funge da punto di ingresso del programma. Supponiamo di avere il seguente file *main.rs*:

```
fn main() {
    println!("Esecuzione del programma!");
}
```

In questo caso, stiamo dichiarando un *crate binario*. Quando viene compilato, genera un eseguibile che può essere lanciato, in pratica quello che solitamente usiamo per costruire applicazioni cosiddette *standalone*.

Al contrario, un crate di libreria non ha un punto di ingresso e viene utilizzato per fornire codice da riutilizzare. Invece di avere un file *main.rs*, avrà un file *lib.rs* come punto di partenza. Qui, possiamo definire funzioni, strutture, e moduli che possono essere importati da altri progetti. Ad esempio, supponiamo di avere un file *lib.rs* con il seguente codice:

```
pub fn saluta() {
    println!("Ciao dalla libreria!");
}
```

In questo caso, abbiamo definito una funzione *saluta* pubblica, la quale può essere utilizzata da altri *crate*. Questa però, non verrà eseguita direttamente come in un *crate binario*, piuttosto, potrà essere importata in un altro progetto. Un crate di libreria può essere integrato in un progetto binario o in altri crate di libreria attraverso il sistema di pacchetti di Rust.

La struttura di un *crate* riflette spesso la complessità del progetto. In uno semplice, potrebbe essere composto da un singolo file, come *main.rs* per uno binario o *lib.rs* per una libreria. Tuttavia, i progetti più complessi possono utilizzare più file e directory per organizzare il codice in modo modulare. I moduli vengono utilizzati all'interno dei crate per suddividere il codice in unità logiche più piccole.

Riprendendo l'esempio della gestione dei dati, possiamo organizzare il nostro crate in più file. In un crate binario, possiamo avere un file *main.rs* che importa un modulo da un altro file:

```
mod gestione_dati;

fn main() {
    gestione_dati::opera_sui_dati();
}
```

Nel file *gestione_dati.rs*, possiamo definire il modulo *gestione_dati*:

```
pub mod lettura {
    pub fn leggi_dati() {
        println!("Lettura dati in corso...");
    }
}

pub mod scrittura {
    pub fn scrivi_dati() {
        println!("Scrittura dati in corso...");
    }
}

pub fn opera_sui_dati() {
    lettura::leggi_dati();
```

```
    scrittura::scrivi_dati();
}
```

Questo mostra come un *crate binario* possa essere strutturato utilizzando moduli per mantenere il codice organizzato. Ogni modulo può essere contenuto in un file separato, e tutti possono essere gestiti dal file principale *main.rs*.
In un crate di libreria, la struttura è simile, ma l'obiettivo è fornire moduli riutilizzabili. Ad esempio, in un crate di libreria con un file *lib.rs*, possiamo organizzare il codice come segue:

```
pub mod gestione_dati {
    pub mod lettura {
        pub fn leggi_dati() {
            println!("Lettura dati dalla libreria...");
        }
    }

    pub mod scrittura {
        pub fn scrivi_dati() {
            println!("Scrittura dati dalla libreria...");
        }
    }

    pub fn opera_sui_dati() {
        lettura::leggi_dati();
        scrittura::scrivi_dati();
    }
}
```

Quando un altro *crate* vuole utilizzare questa libreria, può importare il modulo *gestione_dati* e le sue funzioni in questo modo:

```
use crate::gestione_dati::opera_sui_dati;

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

La relazione tra moduli e *crate* è molto stretta: i primi forniscono la suddivisione

logica del codice all'interno dei secondi, mentre il *crate* stesso rappresenta l'unità di compilazione.

Anche per questo motivo, lo spazio dei nomi all'interno di un *crate* è uno degli aspetti fondamentali per gestire l'organizzazione e l'accesso alle varie componenti del codice. Esso agisce come un contesto che impedisce ai nomi definiti in un *crate* di entrare in conflitto con quelli di un altro. Ognuno ha il proprio spazio dei nomi separato, che include tutte le funzioni, strutture, moduli e tipi definiti al suo interno. Questo isolamento garantisce che diversi progetti possano definire elementi con lo stesso nome senza interferire tra loro.

Supponiamo di avere due *crate* differenti, entrambi con una funzione chiamata *processa_dati*. Se usassimo queste due funzioni nello stesso progetto, Rust garantisce che non ci siano conflitti grazie al fatto che ciascuna funzione è confinata nello spazio dei nomi del suo crate di origine. Nel caso in cui si volesse utilizzare entrambe le funzioni, sarebbe sufficiente riferirsi a esse attraverso i loro rispettivi percorsi, come ad esempio *crateA::processa_dati()* e *crateB::processa_dati().* Questo garantisce che non ci siano collisioni di nomi e che gli sviluppatori possano organizzare i propri *crate* in modo autonomo, senza preoccuparsi di interferenze con altri pacchetti o librerie esterne.

L'isolamento e l'indipendenza sono fondamentali per garantire la modularità e la sicurezza del codice. Ogni *crate* esiste in un contesto isolato, dove le sue componenti sono accessibili solo se esposte esplicitamente tramite la parola chiave *pub*. Ciò significa che una libreria può esporre solo le funzionalità che vuole rendere pubbliche, mantenendo private tutte le altre. Questo incapsulamento rende i *crate* molto sicuri, impedendo modifiche accidentali o utilizzi non previsti di funzioni o strutture interne. Inoltre, l'indipendenza facilita la creazione di moduli altamente riutilizzabili che possono essere distribuiti e utilizzati da progetti diversi senza preoccupazioni per la loro interazione interna.

Un altro aspetto importante dell'isolamento è che ciascuno può gestire le proprie dipendenze in modo indipendente. Se due *crate* utilizzano la stessa libreria

esterna, possono farlo senza alcun conflitto, poiché Rust gestisce le versioni e le dipendenze in modo isolato per ciascuno di essi.
Quando si crea un *crate di libreria*, l'obiettivo è costruire una raccolta di funzionalità che possono essere utilizzate da altri progetti. Per farlo si inizia definendo il file *lib.rs* invece di *main.rs.* Supponiamo di voler creare una libreria che gestisca operazioni sui dati. Il file *lib.rs* potrebbe contenere:

```
pub mod gestione_dati {
    pub fn salva_dati() {
        println!("Dati salvati!");
    }

    pub fn carica_dati() {
        println!("Dati caricati!");
    }
}
```

In questo caso, stiamo definendo un modulo *gestione_dati* che espone due funzioni pubbliche: *salva_dati* e *carica_dati*. Questa libreria potrebbe essere inclusa in altri progetti e le sue funzioni possono essere chiamate importandole nello spazio dei nomi del progetto che la utilizza.
Per includere questa libreria in un altro progetto, si può aggiungere il *crate* come dipendenza nel file *Cargo.toml* del progetto. Supponiamo che la nostra libreria si chiami *gestione_dati*, si può aggiungere come segue:

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

Una volta inclusa la libreria, il progetto può utilizzare le sue funzioni con il seguente codice:

```
use gestione_dati::salva_dati;

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

Rust offre un sistema robusto per la pubblicazione di *crate* attraverso la piattaforma *crates.io*, che funge da repository pubblico. Per pubblicare, il primo passo è creare un account e collegarlo al proprio ambiente di sviluppo. Una volta configurato, si può pubblicare la libreria con il comando *cargo publish*.
Prima di pubblicare un *crate*, è necessario assicurarsi che il file *Cargo.toml* sia configurato correttamente. Il file deve includere metadati come il nome, la versione, l'autore e altre informazioni utili per la distribuzione. Un esempio di configurazione per un crate di libreria potrebbe essere questo:

```
 [package]
name = "gestione_dati"
version = "0.1.0"
authors = ["Il tuo nome <tuo.email@example.com>"]
edition = "2021"

[dependencies]
```

Il comando *cargo publish* controllerà se il *crate* è configurato correttamente, lo compilerà e lo caricherà su *crates.io*. Una volta fatto, chiunque potrà utilizzare la libreria aggiungendola come dipendenza nel proprio progetto.

### *La gestione delle dipendenze tra Crate*

La gestione delle dipendenze tra *crate* è un processo fondamentale per garantire che un progetto possa utilizzare codice esterno in modo efficiente e sicuro. Rust adotta un sistema robusto attraverso il file *Cargo.toml,* che è al centro del sistema di build e gestione delle dipendenze del linguaggio. *Cargo*, lo strumento di build di Rust, permette di definire le dipendenze tra crate e di risolverle automaticamente, scaricandole e integrandole nel progetto.
L'introduzione alla gestione delle dipendenze in Rust inizia con la comprensione di come un progetto possa essere suddiviso in moduli riutilizzabili. I *crate* possono essere sviluppati come librerie, distribuiti e poi utilizzati in altri progetti. Per

includere una dipendenza, ci si affida a *Cargo.toml,* dove si specificano i dettagli, come nome, versione e altre informazioni opzionali. Il sistema di gestione si occupa di scaricare e configurare automaticamente i *crate* necessari da *crates.io*, il registro pubblico dei pacchetti.
Ad esempio, se si vuole utilizzare una libreria esterna che fornisce funzioni matematiche avanzate, come *nalgebra*, si può aggiungere la dipendenza nel file *Cargo.toml* in questo modo:

```
 [dependencies]
nalgebra = "0.29"
```

Con questa configurazione, Cargo scaricherà automaticamente la versione 0.29 del crate *nalgebra* e lo renderà disponibile all'interno del progetto. Dopo aver aggiunto la dipendenza, la si può utilizzare nel codice come segue:

```
extern crate nalgebra as na;

fn main() {
    let vettore = na::Vector3::new(1.0, 2.0, 3.0);
    println!("Il vettore è: {:?}", vettore);
}
```

Il listato utilizza una funzione di *nalgebra* per creare un vettore tridimensionale. Cargo si assicura che la dipendenza sia risolta correttamente, scaricata e resa disponibile per l'uso. La gestione delle dipendenze in Rust permette di includere facilmente librerie sviluppate da terze parti *senza dover gestire manualmente il download o l'integrazione del codice*.
Le dipendenze di crate di terze parti sono un aspetto essenziale dello sviluppo in Rust, poiché permettono agli sviluppatori di riutilizzare codice scritto da altri, riducendo così la necessità di reinventare la ruota. *Crates.io*, il repository centrale per le librerie Rust, facilita la ricerca e l'inclusione di queste dipendenze. Oltre a ciò, Cargo permette di gestire le dipendenze locali o specificare repository Git da cui recuperare il codice.

Se un progetto richiede una versione specifica di un crate che non è ancora stata pubblicata, oppure si vuole puntare a una versione di sviluppo, si può aggiungere una dipendenza da un repository Git direttamente nel file *Cargo.toml*. Supponiamo che si voglia utilizzare una versione non ancora rilasciata del crate *serde* (un crate comunemente utilizzato per la serializzazione e deserializzazione di dati):

```
 [dependencies]
serde = { git = "https://github.com/serde-rs/serde", branch = "master" }
```

In questo caso, Cargo clonerà il repository Git specificato e utilizzerà la versione del crate presente nel *branch master*. Questa flessibilità consente di testare nuove funzionalità o correzioni di bug che non sono ancora state incluse in una versione ufficiale.

Un altro aspetto della gestione delle dipendenze è la possibilità di gestire versioni multiple dello stesso crate in modo efficiente. Cargo utilizza un sistema di versioning semantico (*SemVer*), che permette di specificare esattamente quale versione di un crate si desidera utilizzare. Se due di essi dipendono da versioni differenti dello stesso crate, Cargo tenterà di risolvere le versioni compatibili. Se non è possibile trovare una versione comune, Cargo gestirà entrambe le versioni separatamente, mantenendo l'isolamento e l'integrità del progetto.

Per esempio, se un progetto dipende dalla versione 1.2 di un crate e un'altra dipendenza richiede la versione 1.1, Cargo tenterà di trovare una versione compatibile tra le due (ad esempio, 1.2.3 potrebbe funzionare per entrambi). Se non è possibile, le versioni 1.2 e 1.1 saranno mantenute separatamente senza creare conflitti.

Inoltre, Rust supporta le *feature flags*, che permettono di abilitare o disabilitare alcune funzionalità di un crate in base alle esigenze del progetto. Le feature flags sono definite nel file *Cargo.toml* e offrono un controllo fine-grained su quali componenti di un crate vengono compilati e inclusi. Questo approccio può ridurre il tempo di compilazione e le dimensioni del binario risultante, migliorando le

performance del progetto.
Per esempio, se si volesse utilizzare il crate *serde* con supporto per JSON, si potrebbe configurare la dipendenza in questo modo:

```
 [dependencies]
serde = { version = "1.0", features = ["serde_json"] }
```

In questo caso, si abilita esplicitamente il supporto per JSON all'interno del crate *serde*. Cargo gestirà la configurazione e assicurerà che il codice appropriato venga incluso durante la compilazione.
In Rust, oltre alle dipendenze da crate pubblicati su crates.io o repository Git, è possibile gestire dipendenze locali e dipendenze tra progetti interni. Questa funzionalità è particolarmente utile quando si sviluppano progetti modulari che condividono del codice all'interno di un'organizzazione o quando si lavora a più crate che devono interagire tra loro. Le dipendenze locali consentono di includere un crate che risiede in una directory locale senza la necessità di pubblicarlo o distribuirlo su un repository pubblico.
Per includere una dipendenza locale, si specifica il percorso del crate direttamente nel solito file *Cargo.toml*. Ad esempio, supponiamo di avere due crate: uno chiamato *gestione_dati,* che risiede in una directory locale, e uno chiamato *applicazione*, che necessita di utilizzare *gestione_dati.* La struttura del progetto potrebbe essere la seguente:

```
progetto/
├── applicazione/
│   ├── Cargo.toml
│   └── src/
│       └── main.rs
└── gestione_dati/
    ├── Cargo.toml
    └── src/
        └── lib.rs
```

Nel file *Cargo.toml* del progetto *applicazione*, si aggiunge la dipendenza dal crate

locale *gestione_dati*:

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

Questo specifica che applicazione dipende dal crate *gestione_dati* presente nella directory superiore. Cargo si occuperà di includere e compilare *gestione_dati* come parte del progetto principale. Nel file *main.rs* del crate *applicazione*, si può quindi utilizzare il modulo fornito da *gestione_dati*:

```
use gestione_dati::salva_dati;

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

Le dipendenze tra progetti interni possono essere particolarmente utili in contesti aziendali o di sviluppo collaborativo, dove diverse squadre lavorano su componenti modulari che vengono integrate in un unico progetto.
Un altro aspetto importante della gestione delle dipendenze è il versionamento semantico dei crate. Rust adotta un sistema che assegna a ciascun crate una versione composta da tre numeri: *MAJOR.MINOR.PATCH*. Tutto ciò ha lo scopo di indicare chiaramente la compatibilità tra le versioni di un crate, aiutando a gestire l'evoluzione del codice senza rompere le dipendenze tra progetti.
Il numero *MAJOR* cambia quando ci sono modifiche non retrocompatibili. Ad esempio, se una nuova versione di un crate introduce modifiche che rompono le API esistenti, la versione *MAJOR* viene incrementata. Il numero *MINOR* cambia quando vengono aggiunte nuove funzionalità compatibili con la versione precedente. Infine, il numero *PATCH* viene incrementato per correzioni di bug o modifiche minori che non alterano il comportamento esterno del crate.
In *Cargo.toml,* è possibile specificare quale versione di un crate si desidera utilizzare. Ad esempio:

```
 [dependencies]
serde = "1.0"
```

In questo caso, si richiede la versione 1.0 di *serde*, e Cargo scaricherà l'ultima versione disponibile che rientra nella serie 1.x.x, garantendo che nessuna modifica non retrocompatibile venga introdotta nel progetto.
Se si vuole fissare una versione più specifica, si può utilizzare una versione precisa o una gamma di versioni:

```
serde = "1.0.45"  # Usa esattamente la versione 1.0.45
serde = "^1.0"    # Usa una versione compatibile con 1.0 (qualsiasi versione 1.x.x)
serde = "~1.0.5"  # Usa la versione 1.0.5 o superiore, ma non oltre 1.1.0
```

Questo meccanismo aiuta a mantenere il progetto stabile e a evitare conflitti di versioni tra le dipendenze.
Le dipendenze transitive sono un concetto fondamentale nel sistema di gestione di Rust. Si verifica quando un crate su cui dipendiamo a sua volta dipende da altri. Ad esempio, supponiamo che il crate *applicazione* dipenda dal crate *gestione_dati*, che a sua volta dipende da un altro crate chiamato *logger*. Anche se *applicazione* non dipende direttamente da *logger*, durante la compilazione Cargo risolverà automaticamente tutte le dipendenze, incluse quelle transitive, e scaricherà *logger* insieme a *gestione_dati*.
Questo processo di risoluzione delle dipendenze transitive è completamente automatizzato da Cargo, che si assicura che tutte le versioni siano compatibili tra loro. Ad esempio, se gestione_dati richiede la versione 1.0 di logger e un altro crate nel progetto richiede la versione 1.1, Cargo tenterà di risolvere il conflitto, scaricando entrambe le versioni se necessario o cercando di trovare una versione compatibile:

```
 [dependencies]
gestione_dati = "0.1"
```

Supponiamo che *gestione_dati* abbia nel suo file *Cargo.toml* la seguente

dipendenza:

```
 [dependencies]
logger = "0.5"
```

Quando Cargo compila applicazione, scaricherà sia *gestione_dati* che *logger,* risolvendo così le dipendenze transitive in modo automatico.
Un altro aspetto importante è che Cargo supporta anche l'aggiornamento delle dipendenze in modo sicuro. Utilizzando il comando *cargo update*, si possono aggiornare alle versioni più recenti compatibili con le specifiche definite nel file *Cargo.toml*. Questo comando è utile quando si vuole ottenere le ultime correzioni di bug o miglioramenti nelle dipendenze senza modificare manualmente le versioni specificate.
In poche parole, grazie a Cargo, tutti questi aspetti appena visti vengono gestiti in modo automatico, riducendo il carico per gli sviluppatori e migliorando la stabilità e manutenibilità dei progetti, per questo motivo approfondiremo bene questo importante strumento di Rust.

***Cargo***

Cargo è lo strumento di build e gestione dei pacchetti ufficiale per il linguaggio Rust. Come sappiamo si occupa di diversi aspetti fondamentali dello sviluppo, tra cui la compilazione del codice, la gestione delle dipendenze, la generazione di documentazione e la pubblicazione dei pacchetti. La sua importanza è cruciale poiché, in un linguaggio dove la modularità e l'efficienza della compilazione sono essenziali, Cargo semplifica notevolmente l'organizzazione dei progetti e la distribuzione dei crate, sia come eseguibili binari che come librerie.
Quando si crea un nuovo progetto, la prima cosa da fare è utilizzare Cargo per inizializzarlo. Ciò stabilisce automaticamente una struttura di base per il progetto, riducendo la complessità dell'organizzazione manuale. Ad esempio, il comando *cargo new* crea un progetto con una configurazione predefinita, includendo già i

file fondamentali.
Un progetto Cargo è suddiviso in una struttura ben definita. La directory principale contiene il file *Cargo.toml*, che descrive il progetto e le sue dipendenze. Oltre a questo, c'è la directory *src* che contiene il codice sorgente del progetto. In un progetto binario, il punto di ingresso è il file *main.rs* all'interno della directory *src*, mentre in un progetto di libreria il file principale sarà lib.rs. Ad esempio, se si crea un nuovo progetto con il comando: *cargo new applicazione* Cargo genererà una struttura del genere:

```
applicazione/
├── Cargo.toml
└── src/
    └── main.rs
```

Questo file è fondamentale per la gestione del progetto, e un esempio di come potrebbe apparire è il seguente:

```
 [package]
name = "applicazione"
version = "0.1.0"
authors = ["Autore <autore@email.com>"]
edition = "2021"

[dependencies]
rand = "0.8"
```

Il progetto *applicazione* ha come dipendenza il crate *rand*, una libreria per la generazione di numeri casuali. Le dipendenze vengono dichiarate sotto la sezione [dependencies], e Cargo si occupa di scaricarle, compilarle e integrarle nel progetto in modo trasparente per lo sviluppatore.
Il file *main.rs* rappresenta il punto di ingresso del programma se si tratta di un progetto binario. Può contenere del codice di base come questo:

```
fn main() {
    println!("Ciao, mondo!");
```

```
}
```

Cargo semplifica la compilazione e l'esecuzione del progetto con un solo comando:

```
cargo run
```

Questo si occupa di compilare il codice, gestire le dipendenze e avviare l'eseguibile. Se il progetto contiene moduli o dipendenze complesse, Cargo si assicura che tutto venga compilato e collegato correttamente, riducendo l'onere per lo sviluppatore.
Per un progetto di libreria, la struttura generata da Cargo cambia leggermente. Utilizzando il comando:

```
cargo new gestione_dati --lib
```

Cargo genererà una struttura simile a questa:

```
gestione_dati/
├── Cargo.toml
└── src/
    └── lib.rs
```

In questo caso, il file *lib.rs* è il punto centrale del progetto, che espone le funzioni e i moduli della libreria. Il file potrebbe contenere qualcosa di simile:

```
pub fn salva_dati() {
    println!("Dati salvati correttamente.");
}
```

Quando altri progetti includono *gestione_dati* come dipendenza, potranno usare questa funzione importandola dal crate, grazie alla gestione delle dipendenze automatizzata da Cargo.
Un altro vantaggio è la gestione delle configurazioni di build. Un progetto può essere compilato in modalità debug o in modalità release. La modalità debug viene utilizzata durante lo sviluppo, con tempi di compilazione più rapidi e output di

debug inclusi. La modalità release, d'altra parte, produce un binario ottimizzato per le performance, adatto alla distribuzione. Per compilare un progetto in modalità release, basta digitare:

```
cargo build --release
```

Cargo genera un eseguibile ottimizzato nella cartella target/release. Questa separazione delle configurazioni permette agli sviluppatori di iterare rapidamente durante lo sviluppo e di ottimizzare il codice quando è pronto per essere distribuito.
Infine, uno degli aspetti più importanti è l'integrazione con l'ecosistema Rust attraverso *crates.io*, il repository centrale per i pacchetti. Cargo facilita la loro pubblicazione con comandi semplici. Una volta completato un progetto di libreria, lo si può pubblicare con la semplice istruzione:

```
cargo publish
```

Questa istruzione invia il crate al repository pubblico, rendendolo disponibile per altri sviluppatori. Prima della pubblicazione, Cargo effettua dei controlli per assicurarsi che tutto sia in ordine, come la correttezza della versione e delle dipendenze.
Il file *Cargo.toml* è il cuore di ogni progetto Rust gestito tramite Cargo. Si tratta di un file di configurazione che descrive in modo dettagliato il progetto, incluse le sue dipendenze, metadati, configurazioni di compilazione e altro. Il formato segue lo standard TOML (*Tom's Obvious, Minimal Language*), che è un linguaggio di configurazione leggibile e strutturato in sezioni.
La struttura di un file *Cargo.toml* è suddivisa in varie sezioni che servono a scopi specifici. All'inizio troviamo la sezione [package], che contiene informazioni generali sul progetto. Questa sezione descrive il nome del pacchetto, la versione, l'autore, e altre proprietà come l'edizione di Rust utilizzata. Ad esempio:

```
[package]
```

```
name = "applicazione"
version = "0.1.0"
authors = ["Nome Autore <email@esempio.com>"]
edition = "2021"
```

In questo caso, *name* rappresenta il nome del progetto, *version* la versione corrente, *authors* l'autore o gli autori, e *edition* specifica l'edizione di Rust utilizzata, che definisce il set di funzionalità e regole del linguaggio. La versione è un numero che segue le convenzioni del versionamento semantico (*SemVer*), che facilita la gestione delle versioni durante il ciclo di vita del progetto.
La sezione più importante di *Cargo.toml* riguarda la gestione delle dipendenze. *In pratica sono crate esterni, librerie o moduli di terze parti, che il progetto utilizza*. Cargo ne semplifica la gestione: si specificano i crate necessari e la loro versione desiderata, e il sistema si occupa di scaricarli e compilarli automaticamente. Le dipendenze vengono dichiarate nella sezione [dependencies]:

```
 [dependencies]
serde = "1.0"
rand = "0.8"
```

In queste righe, il progetto dipende da *serde*, una popolare libreria per la serializzazione, e da *rand*, una libreria per la generazione di numeri casuali. Le versioni specificate utilizzano il sistema *SemVer*, e Cargo sceglierà la versione più recente compatibile con queste specifiche.
Oltre alle dipendenze base, è possibile dichiarare dipendenze opzionali o specificare condizioni particolari. Ad esempio, se ne può dichiarare una che viene utilizzata solo durante la fase di test:

```
 [dev-dependencies]
criterion = "0.3"
```

Qui *criterion* è una libreria utilizzata solo per test e benchmarking. Questo approccio permette di mantenere separati i pacchetti necessari per lo sviluppo o il testing da quelli necessari per la produzione.

Cargo offre una gestione flessibile delle dipendenze, ne possiamo indicare alcune con caratteristiche specifiche. Ad esempio, se ne vogliamo una da un repository Git o da una directory locale, possiamo specificarlo nel file *Cargo.toml*:

```
 [dependencies]
gestione_dati = { git = "https://github.com/user/gestione_dati.git" }
```

Questa configurazione scaricherà il crate *gestione_dati* direttamente da un repository Git.
Se invece la dipendenza risiede in una directory locale, si utilizza l'attributo *path*:

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

Una volta configurate le dipendenze, Cargo si occupa di gestirle, scaricandole da *crates.io*, o da altre fonti come Git o file system locali.
Per quanto riguarda la creazione, l'aggiornamento e la rimozione delle dipendenze, Cargo offre comandi molto semplici e potenti. Per aggiungere una dipendenza a un progetto, si può usare il comando *cargo add*. Questo aggiorna automaticamente il file *Cargo.toml*, inserendo la dipendenza con la versione più recente compatibile:

```
cargo add serde
```

Viene così aggiunto il crate *serde* al progetto, includendo automaticamente la sua ultima versione compatibile nella sezione [dependencies] di *Cargo.toml*. La gestione automatizzata elimina la necessità di inserire manualmente dipendenze o versioni specifiche.
Per aggiornare le dipendenze, Cargo offre il comando *cargo update*. Questo comando aggiorna tutte le dipendenze alla versione più recente compatibile con le specifiche di *SemVer* presenti in *Cargo.toml.* Ad esempio, se la dipendenza di serde è configurata come 1.0, Cargo scaricherà la versione più recente di *serde* che inizia con 1.x.x, senza introdurre modifiche non retrocompatibili:

```
cargo update
```

Questo processo di aggiornamento garantisce che il progetto utilizzi sempre le versioni più recenti dei crate senza introdurre rotture. Se si desidera aggiornare solo una singola dipendenza, è possibile specificare il nome del crate:

```
cargo update -p serde
```

Nel caso in cui una dipendenza non sia più necessaria, si può rimuoverla con il comando *cargo remove*:

```
cargo remove rand
```

Questo rimuoverà la dipendenza *rand* dal file *Cargo.toml*, eliminando anche il crate dalle risorse utilizzate dal progetto.
Infine, Cargo si occupa anche della risoluzione delle dipendenze transitive, garantendo che tutte le versioni dei crate utilizzati nel progetto siano compatibili tra loro. Vengono effettuati una serie di controlli per assicurarsi che non ci siano conflitti tra le versioni delle dipendenze richieste da diversi crate.
Un'altra caratteristica avanzata di *Cargo.toml* è la possibilità di gestire feature flags. Le feature sono opzioni che possono essere abilitate o disabilitate per configurare la funzionalità del crate a seconda delle necessità. Ad esempio, un crate può avere funzionalità opzionali che possono essere abilitate nel momento in cui viene importato. Questa configurazione avviene aggiungendo una sezione [features] in *Cargo.toml:*

```
 [features]
default = ["logging"]
logging = []
```

In questo caso, la feature *logging* viene abilitata per default. Altre possono essere abilitate direttamente nel *Cargo.toml* di un progetto che dipende da questo crate, aggiungendo le feature specificate accanto al crate:

```
 [dependencies]
gestione_dati = { version = "0.1", features = ["logging"] }
```

Le feature permettono di mantenere il codice flessibile e personalizzabile, offrendo ai progetti che lo utilizzano la possibilità di configurare solo le funzionalità di cui hanno realmente bisogno.

Cargo gestisce l'intero processo di compilazione, semplificando il build system grazie a una serie di strumenti automatizzati e configurabili. Durante la compilazione di un progetto, Cargo si occupa di diverse fasi, tra cui il download e la gestione delle dipendenze, la compilazione del codice sorgente, l'ottimizzazione del binario finale e la gestione di file temporanei e di output. Questo permette agli sviluppatori di concentrarsi sullo sviluppo del codice senza preoccuparsi troppo dei dettagli tecnici del processo di compilazione.

Il processo di compilazione segue un percorso ben definito. Quando si esegue il comando *cargo build*, il sistema compila il progetto nella modalità predefinita di debug, creando una versione non ottimizzata dell'eseguibile. Questa modalità è utile per lo sviluppo attivo, in quanto la compilazione è più rapida e il codice contiene informazioni di debug. Tuttavia, il risultato non è ottimizzato in termini di performance. Se si esegue invece *cargo build --release*, il quale compila il progetto nella modalità di rilascio, ottimizzando il codice per le prestazioni e rimuovendo il codice di debug.

Le due modalità principali di compilazione sono dunque la modalità di sviluppo (debug) e la modalità di rilascio (release). Nella prima, il codice viene compilato rapidamente senza particolari ottimizzazioni, includendo le informazioni di debug che permettono di utilizzare strumenti come i debugger per individuare eventuali errori. Questo è il profilo di compilazione predefinito, e il binario risultante si trova nella cartella target/debug. Nella modalità release, invece, Cargo utilizza il compilatore di Rust per applicare una serie di ottimizzazioni al codice, come l'eliminazione del codice inutile, l'inlining delle funzioni e altre tecniche per ridurre le dimensioni del binario e migliorare la velocità di esecuzione. Il binario

ottimizzato si trova nella cartella target/release.
La differenza principale tra le due modalità riguarda le ottimizzazioni applicate al codice. Nella modalità debug, le informazioni di debug sono mantenute e le ottimizzazioni sono disabilitate per ridurre i tempi di compilazione. Nella modalità release, il compilatore applica ottimizzazioni aggressive per migliorare le prestazioni, ma questo aumenta significativamente il tempo di compilazione. Ad esempio, un programma compilato in modalità debug potrebbe contenere istruzioni aggiuntive per gestire l'arresto anomalo del programma o informazioni sulla mappatura del codice sorgente, mentre lo stesso programma compilato in modalità release sarà privo di tali informazioni e più efficiente dal punto di vista computazionale.
Un aspetto importante del sistema di build è la cache e la ricompilazione incrementale. Ogni volta che si compila un progetto, Cargo salva i risultati intermedi in una cache per evitare di ricompilare tutto da zero durante le successive compilazioni. Questo significa che se si modificano solo alcuni file, Cargo ricompilerà solo quelli, accelerando notevolmente il processo di compilazione. La cache si trova nella directory target, che contiene anche i file binari compilati e i risultati intermedi. Grazie alla ricompilazione incrementale, Cargo può gestire anche progetti di grandi dimensioni in modo efficiente, evitando di ricompilare i moduli o le dipendenze che non sono stati modificati.
La personalizzazione dei profili di compilazione consente agli sviluppatori di configurare i dettagli delle modalità di debug e release all'interno del file *Cargo.toml.* Rust permette di specificare vari parametri che influenzano la compilazione tramite la sezione [profile]. Ad esempio, si possono configurare ottimizzazioni specifiche, gestire il livello di debug o personalizzare altri aspetti della compilazione per ogni modalità. Un esempio di configurazione personalizzata potrebbe essere:

```
 [profile.dev]
opt-level = 0
```

```
debug = true
lto = false

[profile.release]
opt-level = 3
debug = false
lto = true
```

Per la modalità di sviluppo (dev), il livello di ottimizzazione è impostato a 0, il che significa nessuna ottimizzazione, e le informazioni di debug sono incluse. Per la modalità di rilascio (release), il livello di ottimizzazione è impostato a 3 (massima ottimizzazione), le informazioni di debug sono disabilitate e viene abilitata l'opzione LTO (*Link-Time Optimization*), una tecnica che ottimizza il codice durante il processo di linking, riducendo ulteriormente le dimensioni del binario e migliorando le prestazioni.

La personalizzazione dei profili di compilazione permette quindi di adattare il processo di build alle esigenze specifiche del progetto. Ad esempio, durante lo sviluppo, si può scegliere di disabilitare completamente le ottimizzazioni per ridurre i tempi di compilazione e velocizzare il ciclo di feedback, mentre per le build di produzione si può abilitare ogni forma di ottimizzazione possibile per ottenere il massimo dalle performance del codice.

Cargo fornisce anche strumenti per monitorare l'efficienza del processo di compilazione e ottimizzazione. È possibile utilizzare il comando *cargo check* per verificare il codice senza effettivamente compilarlo in un eseguibile. Questo è particolarmente utile durante lo sviluppo, in quanto permette di rilevare rapidamente errori di compilazione o problemi di tipo senza la necessità di attendere la generazione di un binario completo.

In sintesi, il sistema di build di Cargo è pensato per massimizzare l'efficienza e la flessibilità durante lo sviluppo e la distribuzione di applicazioni. Grazie alla modalità di sviluppo e rilascio, alla gestione della cache e alla ricompilazione incrementale, Rust riesce a bilanciare velocità e ottimizzazione, garantendo un'esperienza di sviluppo fluida. La possibilità di personalizzare i profili di compilazione offre inoltre

agli sviluppatori un controllo fine su come il loro codice viene gestito, permettendo loro di ottimizzare sia per la velocità di sviluppo che per le performance in produzione.
Oltre a ciò, abbiamo i test, i quali sono una parte fondamentale dello sviluppo, integrati direttamente nel linguaggio e gestiti attraverso Cargo. La filosofia alla base del testing è quella di garantire che il codice funzioni correttamente e sia robusto. Rust offre diversi strumenti per scrivere ed eseguire test in modo efficiente, tra cui test unitari e test di integrazione. I primi sono piccoli test che verificano il comportamento di singole funzioni o moduli. Questi sono generalmente inclusi nello stesso file del codice sorgente, e vengono definiti all'interno di un modulo speciale con l'attributo *#[cfg(test)],* che dice al compilatore di eseguire il codice solo durante la fase di testing. L'attributo *#[test]* per identificare una funzione come test, e generalmente utilizza la macro *assert!* o *assert_eq!* per verificare che il risultato di una funzione sia quello atteso.
Ad esempio, ecco un semplice test unitario che verifica il funzionamento di una funzione che somma due numeri:

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

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

    #[test]
    fn test_somma() {
        assert_eq!(somma(2, 3), 5);
    }
}
```

La funzione *somma* prende due interi e restituisce la loro somma. Il test unitario *test_somma* verifica che il risultato di somma(2, 3) sia effettivamente 5. Se il test fallisce, verrà mostrato un messaggio che indica il risultato atteso e quello

ottenuto.
I test di integrazione, invece, verificano il funzionamento di più componenti insieme. A differenza degli unitari, quelli di integrazione vengono scritti in file separati dal codice sorgente e si trovano all'interno di una directory chiamata *tests* nella radice del progetto. Ogni file nella cartella rappresenta un modulo di test separato, e Cargo eseguirà automaticamente tutti quelli che trova in questi file. Ovviamente ciò è utile per verificare che diverse parti del codice funzionino correttamente insieme.
Ad esempio, proviamo a verificare che la funzione *somma* funzioni correttamente quando viene chiamata da un altro modulo. Si può scrivere un test di integrazione in un file come *tests/integrazione.rs*:

```
extern crate nome_progetto;

use nome_progetto::somma;

#[test]
fn test_integrazione_somma() {
    assert_eq!(somma(10, 20), 30);
}
```

In questo caso, si verifica che la funzione *somma* del crate *nome_progetto* restituisca il valore corretto quando viene utilizzata in un contesto più ampio.
L'esecuzione dei test è gestita da Cargo attraverso il comando *cargo test*. Quando viene eseguito si compila il progetto in una modalità speciale per il testing, vengono eseguiti tutti i test definiti (sia unitari che di integrazione) e mostrati i risultati. Se tutti i test passano, Cargo restituisce un messaggio di successo. In caso di fallimento di uno o più test, vengono mostrati dettagliatamente quali test hanno fallito, insieme agli errori o alle asserzioni fallite.
I test sono automatici, non richiedono input manuale e vengono eseguiti in un ambiente controllato. Oltre alle macro di base come *assert!* e *assert_eq!,* esistono strumenti più avanzati, come la possibilità di marcare un test come

"*should_panic*", indicando che il test dovrebbe generare un errore per essere considerato valido. Ad esempio:

```
#[test]
#[should_panic]
fn test_panic() {
    panic!("Errore previsto");
}
```

Questo verificherà che il codice generi effettivamente un panico, e considererà il test come superato se l'errore viene sollevato.
Rust permette inoltre di filtrare i test eseguendo solo quelli che corrispondono a un certo nome, il che è particolarmente utile quando si sta lavorando su una funzione specifica e si desidera eseguire solo i test rilevanti. Per farlo, si può usare un comando come:

```
cargo test nome_test
```

Cargo fornisce anche un'opzione per eseguire i test in modalità parallela, sfruttando tutte le CPU disponibili, il che velocizza l'esecuzione su grandi codebase. Inoltre, supporta l'integrazione con strumenti di analisi della copertura del codice, che aiutano a valutare quale porzione viene effettivamente gestita dai test.
Oltre ai test automatizzati, Cargo consente anche di verificare i test a lungo termine con i test condizionali o test asincroni. Questi permettono di testare parti del codice che potrebbero dipendere da operazioni asincrone o da condizioni che richiedono un tempo di esecuzione prolungato. Si può, ad esempio, configurare test che si comportano in modo diverso a seconda del contesto di esecuzione, come test che verificano l'interazione con un database o un'API esterna.
Per quanto riguarda la gestione dei test a lungo termine o particolarmente complessi, possiamo marcarne alcuni come "ignorati", eseguendoli solo su richiesta:

```
#[test]
```

```
#[ignore]
fn test_lungo() {
    // test che richiede molto tempo
}
```

In questo modo, il test non verrà eseguito automaticamente durante la loro normale esecuzione, ma potrà essere eseguito manualmente con il comando *cargo test -- --ignored.*

In sintesi, Rust fornisce un sistema di testing integrato e potente, che rende semplice scrivere, eseguire e gestire test sia unitari che di integrazione. Grazie all'integrazione con Cargo, l'esecuzione e la gestione dei test diventano parte naturale del ciclo di sviluppo, garantendo un processo fluido e affidabile.

### *Pubblicazione di un Crate*

Preparare un *crate* per la pubblicazione su *crates.io*, il registro ufficiale di Rust per la distribuzione di pacchetti, richiede un'attenzione particolare a diversi aspetti. Oltre alla qualità del codice, è necessario garantire che il *crate* sia ben documentato, configurato correttamente e rispetti le best practice per la distribuzione.

Il primo passo per pubblicare è configurare correttamente il file *Cargo.toml.* Questo contiene metadati essenziali, come il nome, la versione, l'autore e la descrizione. Alcuni campi, come *description, license* e *repository*, sono fondamentali per garantire che il crate sia accettato. È importante fornire una descrizione chiara e completa, poiché aiuterà gli altri sviluppatori a comprendere lo scopo e l'utilizzo del pacchetto. Ecco un esempio di una sezione di *Cargo.toml* pronta per la pubblicazione:

```
 [package]
name = "mio_crate"
version = "0.1.0"
authors = ["Nome Autore <autore@email.com>"]
edition = "2021"
description = "Un crate che fa qualcosa di utile"
```

```
license = "MIT"
repository = "https://github.com/nomeutente/mio_crate"
homepage = "https://mio_crate_homepage.com"
keywords = ["utile", "rust", "crate"]
categories = ["algorithms", "data-structures"]
readme = "README.md"
```

La licenza (license) è un campo obbligatorio, specifica come viene distribuito il codice. È consuetudine usare licenze open source come la MIT o la Apache-2.0. Inoltre, includere campi come *keywords* e *categories* aiuta a rendere il *crate* più facile da trovare tramite il motore di ricerca di crates.io.
Prima di pubblicare un *crate*, è fondamentale fornire una documentazione chiara. Rust incoraggia la scrittura di documentazione tramite i commenti di documentazione, che iniziano con tre barre (///). Questi sono usati per generare automaticamente la documentazione tramite strumenti come *cargo doc*. Ogni funzione pubblica, struct o modulo dovrebbe avere una spiegazione chiara del suo scopo, del suo utilizzo e di eventuali particolari requisiti o limitazioni. Ecco un esempio di commento di documentazione per una funzione:

```
/// Somma due numeri interi.
///
/// # Esempi
///
/// ```
/// let risultato = mio_crate::somma(2, 3);
/// assert_eq!(risultato, 5);
/// ```
pub fn somma(a: i32, b: i32) -> i32 {
    a + b
}
```

La documentazione descrive brevemente la funzione *somma*, e include anche una sezione con esempi (# Esempi) che mostra come utilizzare la funzione. Gli esempi inseriti nei commenti di documentazione vengono eseguiti automaticamente come parte dei test quando si esegue *cargo test*, garantendo che la documentazione sia corretta e coerente con il codice.

Una volta che il *crate* è ben documentato, testato e pronto per la distribuzione, il passaggio successivo è caricarlo su *crates.io*. Per farlo, è necessario innanzitutto creare un account e generare un token di autenticazione. Questo viene utilizzato da Cargo per autenticarsi quando si esegue il comando di pubblicazione.
Dopo aver ottenuto il token, è possibile configurarlo nel proprio ambiente locale tramite la riga:

```
cargo login TOKEN_AUTENTICAZIONE
```

Una volta autenticati, è importante verificare che il *crate* sia pronto per la pubblicazione eseguendo *cargo package*. Questo comando crea un archivio del *crate* e verifica che tutti i metadati e le dipendenze siano configurati correttamente. Se ci sono problemi, Cargo fornirà messaggi di errore o avvertimenti. Se tutto è corretto, si può pubblicare con il seguente comando:

```
cargo publish
```

Cargo caricherà il *crate* su crates.io, rendendolo disponibile per tutti. Una volta pubblicato, potrà essere scaricato e utilizzato da altri sviluppatori semplicemente aggiungendo una dipendenza nel proprio file *Cargo.toml*.
Per garantire la qualità del *crate* e facilitare la sua adozione, ci sono alcune best practice da seguire. La prima è mantenere sempre aggiornata la documentazione. Qualsiasi modifica significativa dovrebbe essere accompagnata da un aggiornamento della documentazione e del file *README.md*, che spesso rappresenta la prima risorsa di riferimento per gli utenti. Includere una sezione con esempi pratici nel file aiuta gli sviluppatori a capire rapidamente come utilizzare il *crate*.
Un'altra best practice è seguire un versionamento semantico (*Semantic Versioning,* o *SemVer*) rigoroso nel formato *MAJOR.MINOR.PATCH*, dove:

Il numero *MAJOR* cambia quando si introducono modifiche incompatibili.
Il numero *MINOR* cambia quando si aggiungono funzionalità compatibili retroattivamente.

Il numero *PATCH* cambia per correzioni di bug compatibili retroattivamente.

Questo tipo di versionamento aiuta gli utenti a sapere se possono aggiornare il *crate* senza rischiare di avere problemi con il loro codice. Ad esempio, se un *crate* passa dalla versione 1.0.0 a 1.1.0, significa che sono state aggiunte nuove funzionalità, ma il codice esistente dovrebbe funzionare senza modifiche. Se invece passa alla versione 2.0.0, significa che ci sono cambiamenti significativi *che potrebbero richiedere adattamenti da parte degli utenti.*
Un altro punto cruciale è l'esecuzione di test automatici prima della pubblicazione. Ogni modifica dovrebbe essere testata localmente con *cargo test* e *cargo doc --no-deps* per assicurarsi che tutto funzioni come previsto e che la documentazione venga generata correttamente.
In sintesi, preparare un crate per la pubblicazione richiede una combinazione di attenzione alla configurazione, alla documentazione e alla qualità del codice. Seguendo queste best practice e sfruttando gli strumenti di Cargo, è possibile distribuire pacchetti di alta qualità, contribuendo all'ecosistema Rust e rendendo il proprio codice disponibile e facilmente utilizzabile da altri sviluppatori.

### *Workspace*

Un workspace è un insieme di progetti che condividono una configurazione comune, consentendo di organizzare e gestire più *crate* in un singolo progetto. I workspace sono particolarmente utili quando si lavora su progetti di grandi dimensioni che richiedono la divisione in più *crate* indipendenti, ma interconnessi. Questo approccio permette una maggiore modularità, facilitando la gestione delle dipendenze, il riutilizzo del codice e la compilazione coordinata.
La creazione di un workspace inizia con un file *Cargo.toml* nella directory principale del progetto. Questo definisce il workspace e indica i *crate* che ne fanno parte. In un workspace, ogni *crate* può avere il proprio *Cargo.toml*, *ma la configurazione globale è centralizzata a livello di workspace.* Esso infatti, coordina la

compilazione, la gestione delle dipendenze e le versioni dei *crate* che lo compongono. Ecco come potrebbe apparire un file *Cargo.toml* di un workspace:

```
 [workspace]
members = ["crate_primo", "crate_secondo"]
```

Il workspace contiene due *crate*, *crate_primo* e *crate_secondo*, che si trovano nelle rispettive directory all'interno della struttura del progetto. Ognuno di essi avrà il proprio file *Cargo.toml*, ma le dipendenze condivise o la gestione delle versioni possono essere centralizzate nel workspace principale.
Un esempio di struttura di un progetto che utilizza un workspace potrebbe essere la seguente:

```
progetto_workspace/
|
├── Cargo.toml
├── crate_primo/
|   └── Cargo.toml
└── crate_secondo/
    └── Cargo.toml
```

I vantaggi di organizzare un progetto in questo spazio sono numerosi. Uno dei più importanti è la possibilità di compilare tutti i *crate* inclusi in un'unica operazione. Usando il comando *cargo build* nella directory del workspace, Cargo si occuperà di compilare ognuno compreso nel workspace in modo efficiente, gestendo le dipendenze comuni e ricompilando solo ciò che è necessario. *Questo permette una compilazione incrementale, migliorando le prestazioni e riducendo i tempi di build in progetti complessi.*
Un altro vantaggio significativo è la condivisione delle dipendenze. Quando più *crate* ne utilizzano le stesse, il workspace consente a Cargo di scaricare e compilare una singola versione di una dipendenza, invece di farlo separatamente per ognuno. Questo non solo riduce i tempi di compilazione, ma evita anche conflitti di versione, garantendo una maggiore coerenza nell'intero progetto.

I workspace facilitano anche il coordinamento tra *crate*. Per esempio, se *crate_primo* dipende da *crate_secondo*, è possibile indicare questa dipendenza nel file *Cargo.toml* di *crate_primo* senza dover pubblicare *crate_secondo* su *crates.io*. Ciò è particolarmente utile durante lo sviluppo, poiché permette di testare e sviluppare i *crate* in parallelo senza doverli distribuire. Ecco un esempio di dipendenza tra *crate* all'interno di un workspace:

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

Come anticipato, *crate_primo* dipende da *crate_secondo*, ma invece di prendere la dipendenza da *crates.io*, questa è specificata tramite il percorso locale. Cargo risolve automaticamente queste dipendenze all'interno del workspace.
Tuttavia, i workspace hanno anche alcuni svantaggi. Uno dei principali è la complessità nella gestione delle versioni dei *crate* quando il progetto cresce. Se alcuni di questi all'interno di un workspace richiedono versioni diverse di una stessa dipendenza, *può essere difficile mantenere la coerenza*. Inoltre, nei progetti molto grandi, la compilazione di tutti i *crate* del workspace può diventare più lenta rispetto alla compilazione di singoli separati, soprattutto se i *crate* sono debolmente collegati tra loro e non condividono molte dipendenze.
Un altro potenziale svantaggio è che i workspace richiedono una maggiore attenzione nella gestione delle dipendenze. Quando si lavora su più *crate* all'interno dello stesso workspace, è facile introdurre dipendenze non necessarie o sovrapposizioni, il che può complicare il ciclo di sviluppo. È quindi importante mantenere una chiara distinzione tra le dipendenze di ciascun *crate* e le dipendenze globali del workspace.
Un'altra funzione utile dei workspace è la pubblicazione di *crate multipli.* Se si desidera pubblicare più *crate* appartenenti a un workspace, è possibile farlo in modo coordinato. Questo è utile, ad esempio, quando si sviluppa una libreria principale e una serie di *crate* ausiliari che la utilizzano. Cargo semplifica la

gestione delle versioni e delle dipendenze tra i vari *crate*, consentendo di pubblicare aggiornamenti coerenti.

### *Debugging con Cargo*

Rust fornisce strumenti avanzati di analisi del codice, linting e ottimizzazione delle performance, integrati direttamente in Cargo o accessibili attraverso di esso. Uno degli strumenti principali per il linting è *Cargo Clippy*, mentre per l'analisi delle performance c'è *Cargo Bench*. Viene anche offerto supporto per il debugging dei crate e l'integrazione con strumenti di terze parti.

Cargo *Clippy* è un linting tool molto potente che aiuta a identificare potenziali miglioramenti nel codice, come l'uso inefficiente delle risorse o pratiche di codifica non ottimali. Clippy fornisce suggerimenti per migliorare il codice Rust in termini di efficienza, leggibilità e mantenibilità. È particolarmente utile durante lo sviluppo di progetti, poiché aiuta a prevenire errori comuni e a mantenere uno stile di codifica consistente. Per utilizzarlo, basta eseguire il comando *cargo clippy* nella root del progetto.

Ad esempio, se si ha un codice in cui si fa un confronto non necessario o inefficiente, Clippy potrebbe suggerire una versione più ottimizzata:

```
fn main() {
    let x = vec![1, 2, 3];
    if x.len() > 0 {
        println!("Il vettore non è vuoto.");
    }
}
```

Clippy potrebbe suggerire di usare *!x.is_empty()* al posto di *x.len() > 0* per una maggiore leggibilità ed efficienza:

```
if !x.is_empty() {
    println!("Il vettore non è vuoto.");
}
```

*Cargo Bench* è lo strumento utilizzato per l'analisi delle performance. Permette di eseguire test di benchmark, ossia test che misurano le prestazioni del codice in termini di tempo di esecuzione e risorse utilizzate. Questo è fondamentale quando si lavora su progetti in cui l'efficienza è un fattore critico. Per utilizzare *cargo bench,* bisogna definire delle funzioni di benchmark nel proprio progetto. Questi sono generalmente scritti nella cartella *benches/,* e il codice può essere strutturato come segue:

```
#[macro_use]
extern crate criterion;
use criterion::Criterion;

fn somma_benchmark(c: &mut Criterion) {
    c.bench_function("somma", |b| b.iter(|| somma(2, 2)));
}

criterion_group!(benches, somma_benchmark);
criterion_main!(benches);
```

Eseguendo l'istruzione *cargo bench*, Cargo compila ed esegue i benchmark, fornendo misurazioni dettagliate sui tempi di esecuzione. Questo permette di identificare colli di bottiglia nelle prestazioni e ottimizzare il codice di conseguenza.
Per quanto riguarda il debugging, Rust fornisce strumenti di debugging integrati come *GDB* o *LLDB*, che possono essere utilizzati con Cargo per tracciare e risolvere errori durante lo sviluppo. L'esecuzione di un progetto in modalità di debug avviene tramite il comando *cargo run*, che compila il progetto con simboli di debug inclusi, facilitando l'analisi delle chiamate di funzione, delle variabili e della memoria. Inoltre, se si desidera una build in modalità di debug con un tool esterno come GDB, si può eseguire *cargo build --debug* e poi lanciare GDB sul binario generato:

```
cargo build
gdb target/debug/nome_binario
```

Questo consente di avviare una sessione di debugging interattiva, in cui si possono impostare breakpoints, analizzare variabili locali e seguire il flusso di esecuzione del programma. In alternativa, si possono utilizzare strumenti come lldb, integrati negli ambienti di sviluppo come Visual Studio Code o IntelliJ IDEA con il plugin Rust.

Cargo si integra inoltre facilmente con numerosi strumenti di terze parti, facilitando lo sviluppo, la testing e la distribuzione di progetti complessi. Oltre a *Clippy* e *Bench*, ci sono strumenti come *rustfmt*, che formattano automaticamente il codice secondo le convenzioni di stile di Rust. Per eseguire quest'ultimo su un progetto, basta digitare:

```
cargo fmt
```

Cargo può anche essere esteso con tool di terze parti per integrare analisi statiche, report di copertura del codice e integrazione continua (CI). Ad esempio, si possono configurare strumenti come *Tarpaulin* (https://crates.io/crates/cargo-tarpaulin/0.25.2) per la misurazione della copertura del codice. Utilizzandolo, è possibile ottenere una stima di quanto codice è coperto dai test automatici, un'informazione cruciale per progetti di grandi dimensioni.

L'integrazione di Cargo con piattaforme di CI come GitHub Actions o Travis CI è comune nei progetti open source Rust. Si possono configurare pipeline automatizzate che eseguono test, verificano la formattazione del codice con rustfmt e analizzano potenziali problemi con clippy, prima di accettare modifiche nel repository principale.

### *Considerazioni finali*

Cargo occupa un ruolo centrale nell'ecosistema Rust, fungendo da strumento essenziale per la gestione dei progetti, delle dipendenze e della compilazione del codice. La sua integrazione con *crates.io*, il repository centrale delle librerie e dei pacchetti Rust, permette agli sviluppatori di scaricare, gestire e condividere *crate*

in modo semplice e strutturato. Grazie alla stretta collaborazione con la community, è stato creato un ecosistema robusto, incentivando le best practice e promuovendo la qualità del codice, in particolare nei progetti open-source.
Il ruolo di Cargo non si limita alla gestione locale dei progetti. Grazie alla sua stretta integrazione con *crates.io,* gli sviluppatori possono pubblicare e distribuire *crate* in tutto l'ecosistema Rust, rendendo più semplice la condivisione e il riutilizzo del codice. Quando un progetto necessita di una libreria o di una funzionalità aggiuntiva, è sufficiente dichiarare la dipendenza nel file *Cargo.toml.* Ad esempio, per aggiungere la dipendenza da una libreria disponibile su *crates.io*, si inserisce il seguente blocco di codice:

```
 [dependencies]
rand = "0.8.4"
```

Cargo scaricherà automaticamente la libreria *rand* e la includerà nel progetto, garantendo che tutte le dipendenze necessarie siano risolte e compilate correttamente. Questo processo è reso possibile dalla stretta integrazione tra Cargo e *crates.io*, che semplifica enormemente la gestione delle dipendenze. Inoltre, quest'ultimo funge da punto di riferimento per la community, consentendo agli sviluppatori di trovare e utilizzare le librerie pubblicate da altri, oltre che contribuire a quelle esistenti.
Un aspetto chiave della gestione delle dipendenze nei progetti open-source è l'adozione delle best practice per garantire che il codice rimanga stabile, sicuro e facile da mantenere. Uno dei principi più importanti è il *versionamento semantico*. Quando si pubblica un *crate* su crates.io, è fondamentale seguirne le regole, che aiutano a comunicare chiaramente ai potenziali utilizzatori quali cambiamenti sono stati fatti e se questi comportano rotture con le versioni precedenti. Ad esempio, un incremento del numero di versione maggiore (come da 1.0.0 a 2.0.0) indica che sono stati introdotti cambiamenti incompatibili, mentre un aggiornamento della versione minore (da 1.0.0 a 1.1.0) segnala nuove funzionalità che sono

compatibili con le versioni precedenti.
Per evitare problemi legati all'aggiornamento delle dipendenze, è importante definire con attenzione le versioni dei crate nel file *Cargo.toml*. Un esempio comune è l'uso dei "range di versione", che permette a Cargo di scaricare aggiornamenti compatibili senza rompere la compatibilità con il codice esistente:

```
 [dependencies]
serde = "1.0"
```

Questa configurazione consente a Cargo di aggiornare serde a qualsiasi versione compresa tra 1.0.0 e 2.0.0, garantendo che quelle più recenti e compatibili siano sempre utilizzate, ma senza introdurre cambiamenti che potrebberocreare problemi al progetto.
Oltre alla gestione delle dipendenze, Cargo facilita il contributo e l'interazione con la community. Per contribuire a un progetto open-source su crates.io, questa incoraggia pratiche collaborative, come l'apertura di *pull request*, la segnalazione di bug e la partecipazione a discussioni tecniche sui repository GitHub. Prima di contribuire, è buona norma leggere i file *CONTRIBUTING.md* e *README.md* di un progetto, che solitamente contengono indicazioni su come farlo in modo efficace.
Per pubblicare un crate, si utilizza il comando cargo *publish*. Questo richiede che si abbia un account su *crates.io*, e che il progetto sia configurato correttamente con il file *Cargo.toml* contenente tutte le informazioni necessarie, come il nome del crate, la versione e l'autore. Un esempio di file pronto per la pubblicazione potrebbe essere:

```
 [package]
name = "mio_crate"
version = "0.1.0"
authors = ["Nome Autore <email@example.com>"]
edition = "2021"

[dependencies]
serde = "1.0"
```

Dopo aver eseguito cargo *publish*, il *crate* sarà disponibile per chiunque desideri utilizzarlo. Cargo garantisce che tutte le dipendenze siano risolte e che il *crate* venga distribuito con tutte le informazioni necessarie per il riutilizzo.
La community Rust ha un ruolo cruciale nello sviluppo e nella manutenzione dell'ecosistema. Oltre alla condivisione di librerie tramite *crates.io,* gli sviluppatori partecipano attivamente al miglioramento degli strumenti e del linguaggio. Contribuire a un progetto open-source non si limita solo al codice: la documentazione è altrettanto importante. Rust ha un sistema di documentazione incorporato, che permette di scrivere commenti strutturati direttamente nel codice utilizzando i *docstring*, che vengono poi compilati in documentazione HTML leggibile.
Per generare la documentazione di un crate, si utilizza il comando cargo *doc*. Questo crea una versione HTML della documentazione, che può essere consultata localmente o pubblicata su piattaforme pubbliche. La documentazione è una parte essenziale per rendere i crate open-source facilmente utilizzabili da altri sviluppatori. Un esempio di commento di documentazione potrebbe essere:

```
/// Questa funzione somma due numeri interi.
///
/// # Esempio
///
/// ```
/// let risultato = mio_crate::somma(2, 3);
/// assert_eq!(risultato, 5);
/// ```
pub fn somma(a: i32, b: i32) -> i32 {
    a + b
}
```

Quando esegui cargo *doc*, la documentazione verrà generata con questi commenti, rendendo la funzione facilmente comprensibile e pronta per essere consultata da altri utenti del crate.
In conclusione, abbiamo visto che Cargo è il fondamento dell'ecosistema Rust,

integrato perfettamente con *crates.io* e supportato da una community vibrante e attiva. Grazie a Cargo, è possibile gestire in modo efficiente le dipendenze, sviluppare progetti open-source e contribuire al miglioramento dell'ecosistema Rust. L'adozione delle best practice, come il versionamento semantico e la scrittura di documentazione chiara, garantisce che i progetti Rust rimangano manutenibili, sicuri e facilmente accessibili per la comunità.

### *Quiz & Esercizi*

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*).

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.

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.

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.

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.

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.

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*.

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.

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.

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.

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.

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

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

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

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

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

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

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

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

20) Modificare il file *Cargo.toml* per includere una dipendenza a una versione di pre-release di un crate.

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.

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*.

*NB. una cache LRU (Least Recently Used) è una struttura dati che memorizza una quantità limitata di elementi, rimuovendo quelli meno recentemente utilizzati quando la capacità è raggiunta. Utilizza una combinazione di una coda e una mappa hash per tracciare l'ordine di utilizzo e gestire gli accessi in modo efficiente.*

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.

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 estrarne tutti gli indirizzi IP e restituirli in un vettore.

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.

***Riassunto***

Abbiamo esplorato in profondità i concetti fondamentali dei moduli e dei crate, iniziando dalla relazione tra file, directory e moduli, evidenziando come Rust consenta di organizzare il codice in moduli e sotto-moduli attraverso un sistema gerarchico. La visibilità degli elementi, gestita tramite le parole chiave *pub* e *priv*, permette di controllare quali parti del codice siano accessibili dall'esterno di un modulo. La struttura dei progetti può essere divisa in file e directory che riflettono la gerarchia dei moduli, servendosi di *use* per importare moduli e funzioni, semplificando l'accesso alle diverse parti del progetto.

Abbiamo discusso anche dei vantaggi della modularizzazione, come la separazione delle responsabilità, l'incapsulamento e il controllo della visibilità, che contribuiscono a una migliore organizzazione dei progetti di grandi dimensioni. Questo approccio consente di scrivere codice più leggibile, manutenibile e riutilizzabile, evitando la complessità eccessiva in singoli file.

In seguito, ci siamo concentrati sui crate, che rappresentano l'unità fondamentale di compilazione e distribuzione. Ne esistono due tipologie principali: binari e librerie. I primi sono eseguibili, mentre i crate libreria contengono funzionalità che possono essere importate in altri progetti. La struttura di un crate è strettamente legata a quella dei moduli, poiché può contenere una gerarchia di moduli al suo interno. Il file *Cargo.toml* è fondamentale per la loro configurazione, in quanto specifica dipendenze, metadati e configurazioni varie per la compilazione e il rilascio.

Abbiamo discusso un aspetto cruciale, la gestione delle dipendenze, esplorando come Cargo consente di integrare crate di terze parti nei progetti tramite il file *Cargo.toml.* Le dipendenze locali e quelle tra progetti interni possono essere gestite con riferimenti diretti a percorsi locali, mentre il versionamento semantico aiuta a mantenere la stabilità del progetto garantendo compatibilità tra versioni differenti. Inoltre, Cargo risolve automaticamente le dipendenze transitive, ovvero quelle dipendenze che i crate importati utilizzano a loro volta.

La compilazione e il build system sono elementi centrali nello sviluppo. Cargo ne gestisce sia quest'ultima modalità (debug) che quella di rilascio (release), ottimizzando le performance nella seconda. La cache e il sistema di ricompilazione incrementale riducono i tempi di build evitando di ricompilare parti del progetto che non sono state modificate. I profili di compilazione possono essere personalizzati per adattarsi alle diverse fasi dello sviluppo, bilanciando tra debugging approfondito e prestazioni ottimali.

# # 8 - Multithreading

*Concorrenza e parallelismo*
*Arc, Mutex e RwLock*
*Programmazione asincrona e multithreading*

Il multithreading rappresenta un argomento centrale nell'ambito della programmazione concorrente, poiché sfrutta la capacità del linguaggio di gestire in modo sicuro ed efficiente l'esecuzione simultanea di più thread. Rust, infatti, è stato progettato con particolare attenzione alla sicurezza della memoria e alla concorrenza senza data race, che sono tra i problemi più critici nei linguaggi tradizionali come C o C++. Il multithreading permette di eseguire più compiti contemporaneamente, sfruttando le architetture multi-core moderne, migliorando così le prestazioni dei programmi.
In questo linguaggio i thread sono gestiti a livello di sistema operativo e possono essere pensati come processi leggeri che condividono lo stesso spazio di memoria. Tuttavia, la condivisione della memoria tra thread comporta un rischio significativo di *data race*, ovvero situazioni in cui due o più thread accedono contemporaneamente a una risorsa condivisa senza una corretta sincronizzazione. Rust affronta questo problema attraverso il suo sistema di proprietà e di borrowing, insieme alla verifica tramite il *borrow checker*, garantisce che non si verifichino situazioni di accesso concorrente non sicuro alla memoria, il che significa che i *data race* sono prevenuti a livello di compilazione.
Un concetto fondamentale del multithreading è la distinzione tra thread

"indipendenti" e la condivisione delle risorse. I thread possono essere lanciati in parallelo per eseguire compiti distinti e, al termine, possono unirsi al thread principale. Quando è necessario che più thread collaborino e accedano a dati condivisi, Rust richiede che tali dati siano gestiti in modo sicuro attraverso meccanismi come i *Mutex* e *RwLock*, che garantiscono un accesso esclusivo o concorrente regolato alle risorse. Questi strumenti, tuttavia, sono soggetti a possibili problemi di deadlock, che sono una forma di stallo dovuta a una cattiva gestione della sincronizzazione tra thread. Anche in questo caso, Rust cerca di fornire un approccio rigoroso attraverso l'uso di primitive thread-safe che incoraggiano un design corretto del sistema concorrente.

Un aspetto interessante del multithreading è l'utilizzo di "message passing", un paradigma di programmazione concorrente alternativo alla condivisione di stato. Rust, tramite il modulo mpsc (*multi-producer, single-consumer*), permette di far comunicare i thread attraverso l'invio di messaggi. Questo approccio riduce la necessità di gestire manualmente la sincronizzazione dell'accesso ai dati condivisi, spostando invece l'attenzione sul passaggio sicuro dei dati tra thread. *Message passing* è particolarmente utile per applicazioni che richiedono un'alta separazione delle responsabilità e la minimizzazione della condivisione diretta dello stato, migliorando così la manutenibilità e la sicurezza del codice.

L'esecuzione concorrente con thread può essere realizzata attraverso moduli della libreria standard, ma esistono anche librerie esterne come *Rayon* o *Tokio*. la prima fornisce un'astrazione di alto livello per la parallelizzazione automatica di operazioni su dati collettivi, come iterazioni parallele, mentre *Tokio* è utilizzato per la programmazione asincrona e la gestione di I/O in applicazioni ad alta concorrenza. Quest'ultimo introduce il concetto di "futures", un meccanismo che permette di scrivere codice asincrono non bloccante e scalabile.

Infine, è importante sottolineare che la scelta tra multithreading e programmazione asincrona dipende dalle specifiche esigenze dell'applicazione. Il multithreading è ideale quando si desidera sfruttare al massimo i core della CPU

per operazioni che richiedono un uso intensivo del calcolo, mentre l'asincronicità è più adatta per applicazioni I/O bound, come server web, dove è necessario gestire un elevato numero di richieste in modo efficiente senza bloccare l'esecuzione dei thread.

***Concorrenza e parallelismo***

La concorrenza e il parallelismo sono concetti distinti, ma spesso correlati, nella programmazione. La concorrenza si riferisce alla capacità di un programma di gestire più attività quasi contemporaneamente, senza necessariamente eseguire tutte in parallelo. È un concetto legato alla suddivisione del lavoro e alla gestione efficiente delle risorse, in cui le attività vengono eseguite in modo cooperativo, spesso tramite il multitasking, su un singolo processore o core. Il parallelismo, invece, implica l'esecuzione simultanea di più attività, di solito su più core o processori, con l'obiettivo di migliorare le prestazioni sfruttando le risorse hardware in modo più intensivo.

In Rust, la concorrenza può essere ottenuta con l'uso di thread, task asincrone o attraverso il passaggio di messaggi, mentre il parallelismo si manifesta in scenari dove più thread eseguono compiti simultaneamente su core diversi.

Un esempio di concorrenza potrebbe essere un programma che gestisce più richieste di rete, alternando il lavoro tra di esse senza necessariamente farle progredire contemporaneamente. Invece, un esempio di parallelismo è un programma che suddivide un compito computazionale tra più core della CPU, come sommare grandi serie di numeri distribuendo il lavoro a diversi thread che eseguono calcoli contemporaneamente.

In ambienti multithreaded, emergono problemi comuni come le *race condition,* i *deadlock* e i *livelock,* che possono compromettere l'affidabilità e la correttezza del programma.

Una *race condition* si verifica quando due o più thread accedono a una risorsa condivisa contemporaneamente e almeno uno di loro modifica tale risorsa. In Rust,

grazie al sistema di ownership e al borrow checker, queste situazioni vengono prevenute a compile-time. Ad esempio, se due thread tentano di modificare una variabile condivisa senza sincronizzazione adeguata, Rust non permetterà la compilazione del codice. Tuttavia, se si utilizzano strutture come *Arc<Mutex<T>>* per condividere risorse mutabili tra thread, è possibile implementare correttamente la concorrenza, come si dovrebbe fare in questo caso:

```
use std::thread;

fn main() {
    let mut counter = 0;

    let handle = thread::spawn(move || {
        for _ in 0..10 {
            counter += 1; // Race condition: l'accesso simultaneo non è sicuro
        }
    });

    handle.join().unwrap();
}
```

Senza sincronizzazione, se più thread modificassero counter contemporaneamente, il risultato sarebbe indefinito. Rust, grazie al suo sistema di tipi e borrowing, non permette questa operazione e il codice non compila. La soluzione sarebbe usare un *Mutex* per proteggere l'accesso a *counter*.

Il *deadlock* si verifica quando due o più thread rimangono bloccati in attesa l'uno dell'altro per accedere a risorse di cui entrambi hanno bisogno, creando un ciclo di dipendenze che non può essere risolto. Può verificarsi facilmente quando si utilizzano più lock (ad esempio, due *Mutex*), e i thread li acquisiscono in ordine diverso. Non esiste un meccanismo automatico per prevenire questo scenario, quindi il programmatore deve fare attenzione a come vengono gestiti i lock:

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

```
fn main() {
    let mutex1 = Arc::new(Mutex::new(0));
    let mutex2 = Arc::new(Mutex::new(1));

    let m1 = Arc::clone(&mutex1);
    let m2 = Arc::clone(&mutex2);

    let handle1 = thread::spawn(move || {
        let _lock1 = m1.lock().unwrap();
        let _lock2 = m2.lock().unwrap(); // Deadlock potenziale
    });

    let m1 = Arc::clone(&mutex1);
    let m2 = Arc::clone(&mutex2);

    let handle2 = thread::spawn(move || {
        let _lock2 = m2.lock().unwrap();
        let _lock1 = m1.lock().unwrap(); // Deadlock potenziale
    });

    handle1.join().unwrap();
    handle2.join().unwrap();
}
```

Qui, il thread *handle1* acquisisce il lock su *mutex1* e poi tenta di acquisire *mutex2*, mentre *handle2* acquisisce il lock su *mutex2* e tenta di acquisire *mutex1*. Entrambi i thread sono bloccati in attesa l'uno dell'altro, *causando un deadlock*. Per prevenirlo, è importante stabilire una gerarchia coerente di lock, assicurandosi che i thread acquisiscano i lock sempre nello stesso ordine.

Infine, il *livelock* si verifica quando i thread non sono bloccati ma continuano a modificare il loro stato in modo da non progredire mai. *In poche parole, i thread continuano a lavorare, ma le loro azioni si annullano a vicenda, impedendo al programma di avanzare*.

Un esempio di *livelock* potrebbe verificarsi in uno scenario in cui due thread rilasciano continuamente i lock nel tentativo di risolvere un conflitto di accesso, ma finiscono per ripetere lo stesso schema senza mai riuscire a completare il proprio lavoro. Rust non ha meccanismi intrinseci per prevenire *livelock*, ma, come con i

*deadlock*, il design attento delle logiche di accesso e sincronizzazione può ridurre il rischio.

### *Il modulo std::thread*

Il modulo *std::thread* è il principale strumento per gestire thread nativi, ovvero unità di esecuzione che operano in parallelo su uno o più core del processore. Rust offre un'interfaccia sicura e robusta per creare e gestire thread, mantenendo l'enfasi sulla sicurezza della memoria e sulla prevenzione delle race condition. Ogni thread creato tramite il modulo *std::thread* esegue un compito separato rispetto a quello principale, inizialmente avviato dal programma.

Per creare un thread, si utilizza la funzione *thread::spawn*. Questa accetta una *closure* come argomento, ovvero una funzione anonima, che contiene il codice da eseguire nel nuovo thread. Questo viene poi eseguito in parallelo rispetto al thread principale. Un aspetto importante da considerare è che, al momento della sua creazione, tutte le variabili utilizzate all'interno della *closure* devono rispettare le regole di proprietà e borrowing di Rust, per evitare problemi di sicurezza della memoria:

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("Dal thread separato: {}", i);
        }
    });

    for i in 1..5 {
        println!("Dal thread principale: {}", i);
    }

    handle.join().unwrap();
}
```

Viene quindi creato un nuovo thread che stampa una serie di numeri. La chiusura

passata a *thread::spawn* definisce il comportamento del thread separato. Quello principale continua la sua esecuzione, stampando anche i numeri. Questo dimostra come due thread possano eseguire operazioni in parallelo. Il thread separato viene avviato immediatamente, e il thread principale non è bloccato dall'esecuzione del secondario, almeno fino a quando non si utilizza la funzione *join*.
La funzione *join* è cruciale per la gestione dei thread. Quando si chiama su un thread, quello chiamante (in questo caso, il thread principale) si blocca fino a quando il secondario non ha terminato la sua esecuzione. Questa è una forma di sincronizzazione che permette di garantire che il thread secondario termini correttamente prima che il principale continui o termini l'esecuzione del programma. La funzione *join* restituisce un *Result*, che può essere utilizzato per gestire eventuali errori o panici che possono verificarsi nel thread secondario.
Un aspetto critico della creazione e gestione dei thread è la gestione della proprietà. Quando se ne crea uno e si trasferiscono variabili all'interno della chiusura, si applicano le regole di proprietà e borrowing per garantire che non ci siano problemi di concorrenza. Se una variabile è catturata dalla chiusura per trasferimento di proprietà, non sarà più disponibile nel thread chiamante. Se è necessario condividere dati tra thread, è possibile utilizzare il tipo *Arc* (*Atomic Reference Counted*), che permette di condividere dati immutabili tra thread senza violare le regole di proprietà. Detto ciò, proviamo a creare un thread che accede a una variabile esterna:

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Dal thread separato: {:?}", v);
    });

    handle.join().unwrap();
}
```

La variabile *v* viene trasferita al thread secondario grazie alla parola chiave *move*. Il suo utilizzo serve per trasferire la proprietà della variabile *v* al thread separato, poiché un thread non può accedere in modo sicuro a una variabile che è posseduta da quello principale. Il *move* garantisce che *v* appartenga esclusivamente al secondario, eliminando il rischio di accesso concorrente non sicuro.
La gestione della proprietà diventa più complessa quando si desidera che più thread accedano a una stessa variabile. In questo caso, è necessario utilizzare una struttura dati sicura per la concorrenza, come *Arc<Mutex<T>>.* Essa consente a più thread di possedere una variabile, mentre *Mutex* garantisce che l'accesso mutabile a tale variabile sia sincronizzato e sicuro.
La funzione *join* svolge anche un ruolo chiave nella gestione degli errori nei thread. Se uno di essi genera un panico, ad esempio per un errore irreversibile, il risultato di *join* sarà un *Err* che il programma può gestire in modo sicuro. Senza *join,* il thread secondario potrebbe terminare inaspettatamente senza che il principale ne sia consapevole.

### *Ownership e borrowing nel multithreading*

Il sistema di proprietà, uno dei pilastri del linguaggio, ha un impatto cruciale sulla gestione dei thread e sulla sicurezza della memoria in ambienti concorrenti. Rust è stato progettato con l'obiettivo di garantire la sicurezza della memoria senza la necessità di un garbage collector, utilizzando invece un rigoroso controllo a tempo di compilazione. Questo sistema, che ruota attorno ai concetti di proprietà, borrowing e lifetime, assicura che l'accesso alla memoria sia gestito correttamente, prevenendo problemi tipici della programmazione concorrente come le race condition e gli accessi concorrenti non sicuri.
In questo contesto, ogni variabile ha un proprietario unico. Il proprietario è responsabile della gestione del ciclo di vita della variabile e, quando il proprietario esce dallo scope, la variabile viene automaticamente deallocata. Quando si lavora

con thread, questa regola è particolarmente importante, poiché i thread spesso condividono o si trasferiscono dati. La gestione della proprietà è fondamentale per evitare che più thread accedano contemporaneamente a una stessa risorsa in maniera non sicura.
Un esempio tipico si verifica quando si tenta di creare un thread che accede a una variabile del thread principale. Senza un sistema di proprietà rigoroso, ciò potrebbe portare a situazioni in cui entrambi tentano di accedere o modificare la stessa variabile contemporaneamente, causando una race condition. Rust previene questo a livello di compilazione grazie al *borrow checker*, un meccanismo che verifica che ogni variabile abbia un solo proprietario mutabile o più riferimenti immutabili alla volta. Se il *borrow checker* rileva che due thread possono accedere contemporaneamente a una variabile in modo mutabile, il compilatore blocca l'esecuzione del programma:

```
use std::thread;

fn main() {
    let mut x = 0;

    let handle = thread::spawn(|| {
        x += 1; // Errore: non è permesso l'accesso mutabile concorrente
    });

    handle.join().unwrap();
}
```

Nel listato, il thread secondario tenta di modificare *x*, ma il compilatore rileva che essa appartiene al thread principale e non può essere modificata da un thread separato senza un'adeguata sincronizzazione. Questo garantisce che non si verifichino race condition, evitando che più thread modifichino lo stato condiviso in modo non sicuro. L'errore viene catturato a tempo di compilazione, piuttosto che a runtime, rendendo più facile individuare e correggere il problema.
Per permettere a più thread di accedere ai dati in modo sicuro, esistono diversi

strumenti, tra cui *Arc* e *Mutex*. Il primo consente di condividere dati immutabili tra più thread, gestendo automaticamente la proprietà tramite un conteggio atomico dei riferimenti. Questo significa che diversi thread possono leggere i dati contemporaneamente senza alcun rischio, poiché l'accesso immutabile è sicuro. Tuttavia, se è necessario modificare i dati condivisi, si utilizza un *Mutex*, che garantisce che un solo thread possa accedere a una risorsa alla volta.
Ad esempio, se volessimo condividere un contatore tra più thread e modificarlo in modo sicuro, potremmo usare *Arc<Mutex<T>>:*

```
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!("Risultato: {}", *counter.lock().unwrap());
}
```

*Arc* permette di condividere la proprietà del contatore tra più thread. *Mutex* assicura che solo un thread alla volta possa modificare il contatore, prevenendo le race condition. Ogni thread acquisisce il lock sul *Mutex* prima di accedere alla variabile *counter*, garantendo che l'accesso concorrente avvenga in modo sicuro.

Se un thread prova ad acquisire il lock mentre un altro lo detiene, esso rimarrà bloccato fino a quando non viene rilasciato. Questo garantisce che non ci siano scritture concorrenti non sicure.
Il *borrow checker* impedisce anche di creare situazioni in cui un thread può accedere a una variabile che è stata già deallocata o che si trova in uno stato non valido. Quando una variabile viene trasferita a un thread tramite la parola chiave *move*, la proprietà di quella variabile viene trasferita al thread secondario, e quello principale non può più accedere ad essa. Ciò impedisce situazioni in cui due thread tentano di usare una variabile che è stata deallocata dal thread che ne aveva il controllo. In questa situazione è utile ricordare che Rust utilizza il concetto di lifetime per gestire la durata delle variabili e garantire che esse non vengano utilizzate oltre il loro ciclo di vita. Vediamo un esempio:

```
use std::thread;

fn main() {
    let v = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Dal thread separato: {:?}", v);
    });

    handle.join().unwrap();
}
```

La parola chiave *move* trasferisce la proprietà della variabile *v* al thread separato. Senza di essa, Rust non permetterebbe al thread secondario di accedere a *v*, poiché la variabile è di proprietà del thread principale. Il sistema assicura che *v* non venga utilizzata dal thread primario una volta che è stata trasferita a quello secondario, prevenendo l'accesso alla memoria non valida o deallocata.

### *Condivisione dei dati tra thread*

Le strutture dati sicure per la concorrenza sono fondamentali per gestire

correttamente la condivisione di risorse tra thread, evitando problemi come race condition o accessi non sincronizzati. Rust, grazie al suo sistema di proprietà e al *borrow checker*, assicura che la gestione della memoria e delle risorse avvenga in modo sicuro. Tuttavia, quando è necessario condividere dati tra thread, abbiamo strumenti specifici come *Arc* (*Atomic Reference Counting*), *Mutex,* e *RwLock*, che permettono di gestire sia la condivisione immutabile che quella mutabile in modo sicuro.

*Arc* è una struttura che consente la condivisione di dati immutabili tra più thread. Funziona come un contatore di riferimenti atomico, garantendo che più thread possano accedere a un dato condiviso senza violare le regole di proprietà di Rust. *Arc* viene utilizzato per creare una proprietà condivisa di una risorsa, senza preoccuparsi della deallocazione prematura. Essendo "atomico", il conteggio dei riferimenti è gestito in modo thread-safe, il che significa che è possibile incrementare e decrementare il contatore senza temere race condition:

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

fn main() {
    let data = Arc::new(vec![1, 2, 3]);

    let mut handles = vec![];

    for _ in 0..3 {
        let data = Arc::clone(&data);
        let handle = thread::spawn(move || {
            println!("{:?}", data);
        });
        handles.push(handle);
    }

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

In questo esempio, *Arc* viene utilizzato per condividere il vettore *data* tra più thread. La funzione *Arc::clone* crea un nuovo riferimento a esso, incrementando il conteggio dei riferimenti. Poiché il dato è immutabile, ogni thread può accedere in sicurezza alla stessa risorsa senza violare le regole di proprietà. *Arc* garantisce che la risorsa non venga deallocata fino a quando non esiste più alcun riferimento a essa, gestendo la sua deallocazione in modo sicuro e automatico.

Quando però i thread devono modificare una risorsa condivisa, è necessario utilizzare strutture di sincronizzazione come *Mutex* o *RwLock*, che permettono di gestire l'accesso mutabile ai dati condivisi. Un *Mutex* (*Mutual Exclusion*) è una struttura che garantisce che solo un thread alla volta possa accedere in modo mutabile a una risorsa. Quando un thread vuole accedere a una risorsa protetta da un *Mutex*, deve acquisire un "lock" su di esso. Solo il thread che detiene il lock può accedere o modificare la risorsa; se un altro thread tenta di acquisire il lock mentre è già detenuto, verrà bloccato fino a quando non viene rilasciato:

```
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!("Risultato finale: {}", *counter.lock().unwrap());
}
```

Il *Mutex* protegge la variabile *counter*. Ogni thread acquisisce il lock chiamando *counter.lock(),* che restituisce un *Result* (da qui l'uso di *unwrap*). NB. la chiamata a *counter.lock()* restituisce un *Result* perché l'operazione di acquisire il lock potrebbe fallire. Questo può accadere, ad esempio, se un thread che deteneva il lock è andato in crash senza rilasciarlo correttamente. Il *Result* gestisce queste eventualità, restituendo un *Ok* se il lock è stato acquisito con successo o un *Err* se c'è stato un errore. Usare *unwrap()* serve a ottenere il valore contenuto in *Ok* o a causare un *panic* se si verifica un errore, poiché il codice presume che l'acquisizione del lock sia sempre sicura in quel contesto. Solo un thread alla volta può acquisire il lock e modificare il valore di counter, garantendo che non ci siano scritture concorrenti non sicure. Quando un thread rilascia il lock (il lock viene rilasciato automaticamente quando il valore che lo detiene esce dallo scope), un altro thread può acquisirlo. *Arc* viene utilizzato per consentire a più thread di condividere la proprietà del *Mutex*.

Oltre al *Mutex*, Rust offre un'altra struttura di sincronizzazione chiamata *RwLock* (*Read-Write Lock*), il quale permette di gestire accessi immutabili concorrenti e mutabili esclusivi a una risorsa. A differenza di un *Mutex*, che blocca completamente l'accesso ad una risorsa quando un thread detiene il lock, *RwLock* consente a più thread di accedere in modo immutabile (lettura) a una risorsa, purché nessun altro thread detenga un lock mutabile (scrittura). Se un thread desidera modificare la risorsa, deve acquisire un lock di scrittura, il che impedisce a qualsiasi altro thread di accedervi fino a quando non viene rilasciato:

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

fn main() {
    let data = Arc::new(RwLock::new(5));

    let mut handles = vec![];

    // Lettura concorrente
```

```
    for _ in 0..3 {
        let data = Arc::clone(&data);
        let handle = thread::spawn(move || {
            let read_data = data.read().unwrap();
            println!("Lettura: {}", *read_data);
        });
        handles.push(handle);
    }

    // Scrittura esclusiva
    let data = Arc::clone(&data);
    let write_handle = thread::spawn(move || {
        let mut write_data = data.write().unwrap();
        *write_data += 10;
        println!("Scrittura: {}", *write_data);
    });
    handles.push(write_handle);

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

Qui, tre thread acquisiscono il lock di lettura su *data*, consentendo loro di accedere in modo concorrente alla risorsa. Successivamente, un thread acquisisce il lock di scrittura, il che impedisce qualsiasi altro accesso alla risorsa fino a quando non viene rilasciato. *RwLock* è utile quando si desidera permettere letture concorrenti ma garantire che le scritture avvengano in modo esclusivo.
La scelta tra *Mutex* e *RwLock* dipende dal contesto applicativo: il primo è adatto per scenari dove sia necessario un accesso esclusivo mutabile, mentre l'altro è utile quando sono previste più letture concorrenti e rare modifiche.

### *Sincronizzazione e programmazione asincrona*

La sincronizzazione tra thread è un aspetto fondamentale della programmazione concorrente, e Rust fornisce diverse tecniche e strumenti per gestire questa sincronizzazione in modo sicuro. Essa è necessaria per coordinare l'accesso a risorse condivise tra thread in esecuzione parallela, prevenendo race condition,

deadlock e altri problemi di concorrenza. Il linguaggio, oltre a offrire strumenti di sincronizzazione basati su blocchi come *Mutex* e *RwLock*, adotta anche il *message passing* come meccanismo sicuro ed efficiente per la comunicazione tra thread.
Il *message passing* è una tecnica in cui *i thread comunicano scambiandosi messaggi anziché condividere direttamente risorse*. Questo modello riduce il rischio di errori di sincronizzazione perché i dati vengono trasferiti tra thread senza necessità di locking. Rust lo implementa attraverso il modulo *mpsc* (*multiple producer, single consumer*), che consente a più thread di inviare messaggi a un singolo thread ricevitore.
Il modulo *mpsc* permette di creare canali di comunicazione tra thread. Un canale è composto da un "trasmettitore" (sender) e un "ricevitore" (receiver). I thread possono inviare messaggi attraverso il trasmettitore, mentre un altro thread riceve i messaggi dal ricevitore. La gestione dei canali è thread-safe, e Rust garantisce che i dati vengano trasferiti in modo sicuro e senza accesso concorrente diretto:

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
        let msg = String::from("Ciao dal thread");
        tx.send(msg).unwrap();
    });

    let received = rx.recv().unwrap();
    println!("Messaggio ricevuto: {}", received);
}
```

Creiamo un canale con *mpsc::channel(),* che restituisce un *trasmettitore* (tx) e un *ricevitore* (rx). Un thread secondario invia un messaggio al thread principale utilizzando *tx.send(),* e il thread principale lo riceve con *rx.recv().* La funzione *recv()* si blocca fino a quando non viene ricevuto un messaggio, garantendo così

una forma di sincronizzazione tra i thread. Se il trasmettitore o il ricevitore vengono chiusi, *recv()* restituirà un errore.
Il modulo *mpsc* supporta anche la comunicazione tra più produttori, cioè più thread possono inviare messaggi allo stesso ricevitore. Questo è possibile clonando il *trasmettitore* (tx). Ognuno di essi può inviare messaggi indipendentemente, e il ricevitore li otterrà in ordine:

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();
    let tx1 = tx.clone();

    thread::spawn(move || {
        let msgs = vec![
            String::from("Messaggio 1"),
            String::from("Messaggio 2"),
            String::from("Messaggio 3"),
        ];

        for msg in msgs {
            tx.send(msg).unwrap();
            thread::sleep(Duration::from_millis(500));
        }
    });

    thread::spawn(move || {
        let msgs = vec![
            String::from("Messaggio A"),
            String::from("Messaggio B"),
            String::from("Messaggio C"),
        ];

        for msg in msgs {
            tx1.send(msg).unwrap();
            thread::sleep(Duration::from_millis(300));
        }
    });
```

```
    for received in rx {
        println!("Ricevuto: {}", received);
    }
}
```

Abbiamo due produttori che inviano messaggi a un singolo consumatore. Questi vengono ricevuti nell'ordine in cui arrivano al canale. La sincronizzazione tra i thread è gestita implicitamente attraverso quest'ultimo, eliminando la necessità di bloccare o coordinare manualmente l'accesso alle risorse.

Oltre alla sincronizzazione basata su *message passing*, Rust offre la programmazione asincrona tramite il modulo *async* e *await*. Questa metodologia è concettualmente diversa dal multithreading, mentre quest'ultimo coinvolge l'esecuzione simultanea di più thread su più core, la programmazione asincrona riguarda la sospensione e la ripresa delle attività in un singolo thread. L'obiettivo dell'asincronicità è evitare il blocco del thread principale mentre attende che operazioni lunghe (come I/O) vengano completate, utilizzando invece un modello basato sugli eventi.

Nella programmazione asincrona, un'operazione non bloccante viene avviata e il controllo ritorna immediatamente al chiamante. Quando l'operazione è pronta per essere completata, il runtime asincrono riprende l'esecuzione del task sospeso. Questo approccio è utile per gestire carichi di lavoro che richiedono un gran numero di operazioni I/O o di rete, riducendo il numero di thread necessari e migliorando l'efficienza. Vediamo un esempio di codice asincrono:

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

#[tokio::main]
async fn main() {
    let handle1 = tokio::spawn(async {
        sleep(Duration::from_millis(500)).await;
        println!("Task 1 completato");
    });
```

```
    let handle2 = tokio::spawn(async {
        sleep(Duration::from_millis(300)).await;
        println!("Task 2 completato");
    });

    handle1.await.unwrap();
    handle2.await.unwrap();
}
```

Utilizziamo il runtime *tokio* per gestire i task che vengono eseguiti in modo asincrono, senza bloccare il thread principale. Quando il task esegue l'operazione *sleep*, non blocca il thread ma lo sospende, permettendo ad altri task di essere eseguiti nel frattempo. La funzione *await* sospende l'esecuzione fino a quando l'operazione non viene completata. Questo è diverso dal multithreading, poiché entrambi i task possono essere lanciati nello stesso thread, grazie al runtime asincrono che gestisce la loro sospensione e ripresa.
La differenza principale tra multithreading e programmazione asincrona risiede nel modo in cui vengono gestite le operazioni che richiedono tempo. *Nel multithreading, si utilizzano più thread che eseguono operazioni in parallelo, mentre nella programmazione asincrona, un singolo thread esegue molte operazioni in maniera concorrente, sospendendo e riprendendo i task in base alla disponibilità delle risorse*. Il multithreading è più appropriato quando si ha bisogno di sfruttare al massimo i core della CPU, mentre la programmazione asincrona è utile per gestire carichi I/O pesanti o operazioni che coinvolgono attese lunghe, evitando di bloccare risorse preziose.

### *Quiz & Esercizi*

1) Spiega la differenza tra concorrenza e parallelismo e scrivi un programma che esegua due operazioni in parallelo su thread distinti.
2) Scrivi un programma che utilizzi *Mutex* per condividere una variabile tra thread in modo sicuro.
3) Modifica un programma che incrementa un contatore su più thread e dimostra come un Mutex prevenga le *race condition*.
4) Gestisci due variabili condivise tramite *Mutex* e dimostra come evitare un *deadlock*.

5) Utilizza *RwLock* per consentire accessi concorrenti immutabili a una risorsa condivisa.
6) Utilizza *Arc* e *Cell* per gestire dati condivisi tra thread con mutabilità interna.
7) Utilizza *Arc* e *Mutex* per garantire la sicurezza nell'accesso a dati condivisi tra thread.
8) Simula un server concorrente con accesso sicuro a dati condivisi tra thread utilizzando *Arc, Mutex* e operazioni atomiche.
9) Gestisci correttamente il *deadlock* su più risorse utilizzando *Mutex,* assicurando che i thread acquisiscano i lock in un ordine coerente.
10) Scrivi un programma che implementi una coda concorrente (queue) utilizzando *Arc, Mutex* e *Condvar* per sincronizzare produttori e consumatori su più thread.
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.
12) Crea un thread per inviare cinque stringhe a un canale. Il thread principale deve ricevere e stampare queste stringhe.
13) utilizza il pattern produttore-consumatore, dove un thread produce numeri e un altro li consuma.
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.
15) Scrivi un programma che crea tre thread, ognuno dei quali invia un messaggio al thread principale, che dovrà stampare tutti i messaggi ricevuti.

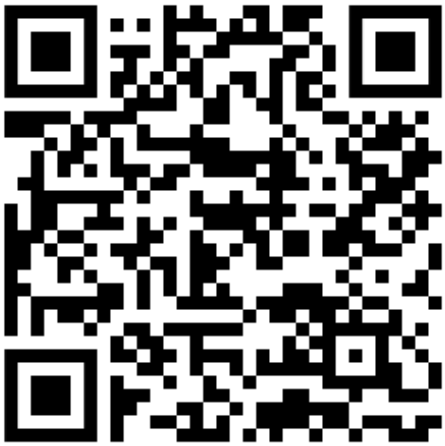
***Riassunto***

In quest'ultima tappa del nostro viaggio, abbiamo esplorato diversi concetti chiave relativi alla concorrenza e al multithreading, partendo dalla distinzione tra concorrenza e parallelismo, chiarendo come il multithreading in Rust permette l'esecuzione simultanea di operazioni su thread distinti per migliorare le prestazioni in ambienti multi-core. Abbiamo discusso dei problemi comuni come *race condition, deadlock* e *livelock*, spiegando come, attraverso il sistema di proprietà e il *borrow checker*, si prevengano tali errori. Si è evidenziata l'importanza del modulo *std::thread,* che gestisce i thread nativi, spiegando come creare, gestire e terminare i thread e come il sistema di proprietà di Rust ne influenzi la gestione. Abbiamo trattato strumenti di sincronizzazione come *Arc*, che consente la condivisione sicura di dati immutabili tra thread, e *Mutex* e *RwLock*, che permettono la condivisione mutabile e la protezione delle risorse critiche. Abbiamo spiegato come funziona il *message passing* utilizzando il modulo *mpsc*, che permette la comunicazione tra thread senza accesso diretto alle risorse condivise, riducendo il rischio di errori di sincronizzazione. Infine, abbiamo distinto la programmazione asincrona dal multithreading, chiarendo che mentre quest'ultimo si basa su thread fisici per eseguire operazioni in parallelo, la programmazione asincrona sospende e riprende i task in un singolo thread, migliorando l'efficienza in operazioni I/O-intensive.

## *Download gratuiti:*

I QR-code indicati sono indirizzati verso un server gratuito (*mega.nz*) da dove potrai scaricare l'ebook del libro e le soluzioni agli esercizi in formato pdf in modo da poter usufruire al meglio di tutti i listati presenti nel libro, senza quindi ogni volta doverli riscrivere da zero:

Ebook



Soluzioni

*Ti ringrazio per aver acquistato questo libro, spero sia stato utile e ti sia piaciuto.*
*Se vorrai lasciare una recensione su Amazon sarà molto gradita, grazie mille, ti voglio bene.*

*Tony Chan*

# Bibliografia

I sottoelencati siti internet hanno fornito ispirazione per diversi argomenti trattati nel libro:


*https://www.rust-lang.org/it/learn*

*https://qastack.it/*

*https://www.w3schools.com/spaces/spaces_rust.php*

*https://stackoverflow.com/*

*http://www.datrevo.com/*

*https://www.w3bai.com/*

*https://ichi.pro/*

*https://gamefromscratch.com/rust-for-game-development/*

*https://goessner.net/*

*https://gabrieleromanato.com/*

*https://www.marchettidesign.net/*

*https://yourinspirationweb.com/*

*https://it.wikipedia.org*

*https://www.codingcreativo.it/*


*E anche i testi:*

***C# Java, PHP, Python, la guida completa alla programmazione ad oggetti*** *di Tony Chan*
***HTML5 & CSS3, la guida completa*** *di Tony Chan*
***Javascript, la guida definitiva*** *di Tony Chan*
***Python, la guida completa*** *di Tony Chan*
***Python ed Ethical Hacking*** *di Tony Chan*
***Trading con Python****, di Tony Chan*
***PROGRAMMARE, Impara velocemente*** *di Tony Chan*

*← link ai libri.*