RUST Lang

Lorenzo Quadri

9 agosto 2025

1 Concetti comuni di programmazione

Rust usa molte parentesi graffe ({..}) che denotano gli scopes. Uno scope rappresenta un blocco di codice che viene eseguito insieme. Le variabili create all'interno di un scope rimangono all'interno dello scope: non fuoriescono e vengono ripuliti automaticamente quando si esce. In questo caso, la stampa di "Hello, World" avviene all'interno della funzione main.

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

Il punto esclamativo contrassegna una macro. Il sistema macro di Rust è molto potente e consente una sintassi che non funzionerebbe con le funzioni regolari. Le variabili in Rust sono per default *immutabili*. Una volta assegnata una variabile immutabile, non è possibile modificare la valore memorizzato nella variabile. Puoi creare più variabili, fare riferimento o copiare una variabile assegnata in precedenza, ma non è possibile modificare la variabile immutabile una volta assegnata. Puoi contrassegnare esplicitamente una variabile come modificabile con la parola chiave **mut**.

```
let mut your_name = String::new();
```

In Rust si usa combinare le funzioni La combinazione di funzioni in Rust è chiamata concatenamento di funzioni. A partire dal top, ogni funzione passa i suoi risultati alla funzione successiva. È comune formattare una catena di funzioni con ogni passaggio sulla riga sucessiva , rientrando per indicare che il blocco è unico.
Ad esempio:

Anteporre ad una variabile una e commerciale (”&”) crea un riferimento alla variabile. Un riferimento passa l'accesso alla variabile stessa, non a una copia della variabile. Questo è anche chiamato *borrowing - prestito* ossia stai prestando la variabile alla funzione che stai chiamando. Prestito con &mut consente alla funzione di prestito di mutare la tua variabile. Qualsiasi modifica apportata alla variabile viene scritta direttamente nella variabile che hai prestato.
Passare &mut your_name alla funzione *read_line* consente alla funzione *read_line* di scrivere direttamente sulla variabile.
Ti aspetti che la funzione read_line funzioni correttamente. In caso contrario, il tuo programma andrà in crash. Rust sta restituendo un oggetto **Result** e stai controllando che la funzione non generi eccezioni chiamando **expect**.
La macro println usa per indicare che il valore di una variabile deve essere stampato.
Fornisci quindi la variabile come secondo parametro alla chiamata macro. Rust

include un sistema di formattazione molto potente e può prendersi cura della maggior parte delle tue esigenze di formattazione delle stringhe.

Per fornire un output alle funzioni in Rust si usa la sintassi **->**.

```
fn what_is_your_name() -> String {
        let mut your_name = String::new();
        stdin()
        .read_line(&mut your_name)
        .expect("Failed to read line");
        your_name
}
```

1.1 Array

Il tipo di elenco più semplice di Rust è l'array. Un array contiene un elenco di valori con due regole: i valori devono essere dello stesso tipo e l'array non può cambiare dimensione.
Una volta deciso chi ammettere, non puoi modificare la dimensione dell'elenco senza ricompilare il tuo programma. Per dichiarare un array di stringhe letterali (tipi &str), puoi fare:

```
let visitor_list = ["bert", "steve", "fred"];
```

Rust deduce ancora una volta il tipo dell'array. Tutte le voci sono stringhe, quindi Rust presuppone che tu voglia un array dello stesso tipo. La creazione dell'array contiene tre voci, quindi Rust deduce che desideri un array di dimensione 3.
Se vuoi dichiarare il tipo, la sintassi completa è

```
let visitor_list : [&str;3] = ... .
```

Rust ha due tipi di stringhe, che possono essere fonte di confusione per i nuovi programmatori di Rust. Il primo tipo è *str*. Questi tipi sono generalmente usati come stringhe letterali, che sono stringhe immesse nel codice sorgente e generalmente immutabili. Il secondo tipo è *String*. Le stringhe sono dinamiche perché memorizzano una posizione, una lunghezza e una capacità. Puoi aggiungere stringhe e modificarle.

Rust può accedere direttamente al contenuto di un array o di un altro contenitore, senza dover utilizzare numeri di indice. Questo è più breve e più sicuro. Non puoi rovinare i tuoi numeri di indice e mandare in crash il tuo programma accedendo a un elemento che non esiste.

```
for visitor in &visitor_list {
        if visitor == your_name { ... }
}
```

1.2 Struct

Una Struct è un tipo che raggruppa i dati. Gli struct contengono variabili dei campi membro di (quasi) qualsiasi tipo. Raggruppare i dati rende più facile

l'accesso a informazioni complicate una volta che si cerca nella struttura giusta, i singoli dati sono disponibili per nome.
Le strutture sono onnipresenti in Rust: String e StdIn sono entrambi tipi di struttura. I tipi struct di Rust combinano dati correlati e possono implementare funzionalità per quel tipo. Sono simili ai tipi di classe in altre lingue. Puoi definire le tue strutture. Le strutture sono un tipo, proprio come i32 , String e le enumerazioni. Un tipo non è una variabile: è una descrizione di quali variabili di quel tipo possono contenere.
Le strutture possono anche avere funzioni e metodi associati. String::new e StdIn::read_line sono entrambi metodi associati a una struttura.
Creare un visitor dovrebbe essere facile, quindi devi creare un costruttore. I costruttori sono funzioni associate a un tipo che forniscono un modo rapido per creare un'istanza di quel tipo. Le funzioni associate che operano su un'istanza della struttura sono talvolta chiamate metodi. In Rust, **&self** come parametro denota un metodo che ha accesso ai contenuti dell'istanza:

```
impl Visitor {
        fn new(name: &str, greeting: &str) -> Self {
        Self {
                name: name.to_lowercase(),
                greeting: greeting.to_string(),
        }
}
fn greet_visitor(&self) {
        println!("{}", self.greeting);
        }
}
```

Implementi le funzioni per una struct con **impl** e il nome della struct.
Questa è una funzione associata. L'elenco dei parametri non include self. Non puoi accedere a new con name.new(); invece, è disponibile nello spazio dei nomi della struttura. Puoi chiamarlo con Visitor::new().
Il costruttore restituisce il tipo Self. Questa è la stenografia di Rust. Clippy ti suggerirà di usare la stenografia se usi la forma lunga. Potresti ugualmente scrivere "Visitor" al suo posto, ma se dovessi cambiare il nome della struttura, dovresti ricordarti di tornare indietro e cambiare ogni implementazione. Usa Self per risparmiarti lavoro.
La mancanza di un punto e virgola denota la sintassi di "ritorno implicito". Crea una nuova istanza della struttura, sempre utilizzando Self invece di Visitor. Ogni campo deve essere elencato, nel formato nome_campo : valore,. La funzione accetta parametri di tipo &str, ma memorizza valori di tipo String. Le funzioni to_lowercase e to_string eseguono la conversione. Prendendo &str , la funzione accetta stringhe letterali senza conversione. Questo ci evita di digitare String::from("bert") quando si chiama il costruttore.
Il case del titolo Self si riferisce al tipo di struttura stesso. Il self minuscolo si riferisce all'istanza della struttura.
Questa funzione è una funzione o un metodo membro. Accetta self come parametro, che viene automaticamente passato alla funzione quando si fa riferimento a un'istanza della struttura (ad es. my_visitor.greet_visitor()) con il contenuto di quella specifica istanza della struttura.

```
let visitor_list = [
        Visitor::new("bert", "Hello Bert, enjoy your treehouse."),
        Visitor::new("steve","Hi Steve. Your milk is in the fridge."),
        Visitor::new("fred", "Wow, who invited Fred?"),
```

```
];
```

Rust fornisce una potente funzionalità nota come **iteratori** per la manipolazione dei dati. Gli iteratori sono un pò una funzionalità generica: possono fare molto. Quando lavori con elenchi di dati, gli iteratori sono il primo posto in cui cercare la funzionalità di cui hai bisogno. Gli iteratori sono progettati attorno al concatenamento di funzioni: ogni passaggio dell'iteratore funge da elemento costitutivo per raggruppare i dati del passaggio precedente in ciò di cui hai bisogno.
Gli iteratori includono una funzione find() per individuare i dati all'interno di una raccolta, che si tratti di un array, di un vettore o di qualcos'altro. Sostituisci il tuo ciclo for con il seguente:

```
let known_visitor = visitor_list
        .iter()
        .find(|visitor| visitor.name == name);
```

Assegna il risultato della funzione iteratore alla variabile known_visitor.
Crea un iteratore con iter() che contenga tutti i dati dal visitor_list.
find() esegue una *closure*. Se la closure restituisce true, find() restituisce il valore corrispondente. Il punto e virgola chiude l'affermazione.
Questa catena di funzioni crea un iteratore e memorizza i risultati della funzione di ricerca in known_visitor . Non puoi essere sicuro che il nome che hai cercato sia nell'elenco dei visitatori. Find() restituisce un tipo Rust chiamato Option . Le Options contengono un valore oppure no. Alcuni linguaggi usano null o nullptr per rappresentare l'assenza di un valore, ma regole deboli per la gestione dei valori null hanno provocato innumerevoli bug.
Le Options in Rust sono un'enumerazione. Le options hanno due possibili valori: Some(x) e None. Esistono molti modi diversi per interagire ed estrarre i dati da un'opzione. Per esempio usando match:

```
match known_visitor {
        Some(visitor) => visitor.greet_visitor(),
        None => println!("You are not on the visitor list. Please leave.")
}
```

Matching Some(visitor) verifica se l'opzione ha dati e rende il contenuto dell'opzione disponibile al codice in questa clausola come visitatore . La freccia grassa (**'=>'**) indica il codice da eseguire per questa corrispondenza, in questo caso, entrando nel metodo greet_visitor. Separare le opzioni di corrispondenza con virgole.

Le chiusure sono molto usate in Rust. Pensa a una chiusura come una funzione che definisci in atto.
La chiusura in linea |visitatore| nome.visitatore == nome equivale a definire una funzione:

```
fn check_visitor_name(visitor: &Visitor, name: &String) -> bool {
        return visitor.name == name;
}
```

1.3 Vettori

Gli array non possono cambiare dimensione. I vettori (Vec) sono progettati per essere dinamicamente ridimensionabili. Possono essere usati come array, puoi

aggiungere elementi con un metodo chiamato push(). I vettori possono continuare a crescere, sei limitato solo dalla dimensione della memoria del tuo computer.

Quando memorizzavi solo i nomi come stringhe, stamparli era facile. Inviali a println! e hai finito. Vuoi essere in grado di stampare i contenuti di una struttura Visitatore. I segnaposto di debug (:? per la stampa non elaborata e :#? , per la stampa "carina") stampano qualsiasi tipo che supporti il tratto Debug. L'aggiunta del supporto per il debug alla struttura dei visitatori è semplice e fa uso di un'altra comoda funzionalità di Rust: la macro Derive.

```
#[derive(Debug)]
        struct Visitor {
                name: String,
                greeting: String
}
```

Le macro *derive* sono un meccanismo molto potente per evitare di digitare codice standard ripetitivo. Puoi ricavare molte cose. La derivazione richiede che ogni campo membro nella struttura supporti la funzionalità che si sta derivando. Fortunatamente, le primitive di Rust come String lo supportano immediatamente. Una volta derivato Debug , puoi usare println! placeholder :? per stampare l'intera struttura.
I vettori di rust in funzioone sono simili agli array, il che rende relativamente facile sostituirli l'uno con l'altro. Rust fornisce una macro utile, vec!, per aiutare in questo. vec! ti consente di inizializzare un vettore con una sintassi simile all'inizializzazione dell'array:

```
let mut visitor_list = vec![
        Visitor::new("Bert", "Hello Bert, enjoy your treehouse."),
        Visitor::new("Steve", "Hi Steve. Your milk is in the fridge."),
        Visitor::new("Fred", "Wow, who invited Fred?"),
];
```

1.4 Datatypes

Ogni valore in Rust è di un certo tipo di dati, il che dice a Rust che tipo di dati vengono specificati in modo che sappia come lavorare con quei dati. Esamineremo due sottoinsiemi di tipi di dati: scalare e composto.
Tieni presente che Rust è un linguaggio tipizzato staticamente, il che significa che deve conoscere i tipi di tutte le variabili in fase di compilazione. Il compilatore di solito può dedurre quale tipo vogliamo utilizzare in base al valore e al modo in cui lo utilizziamo.

1.4.1 Tipi scalari

Un tipo scalare rappresenta un singolo valore. Rust ha quattro tipi scalari primari: numeri interi, numeri in virgola mobile, booleani e caratteri. Potresti riconoscerli da altri linguaggi di programmazione. Andiamo a vedere come funzionano in Rust.

Tipi interi

Un intero è un numero senza una componente frazionaria. Ad esempio preso, u32, questa dichiarazione di tipo indica che il valore a cui è associato dovrebbe essere un intero senza segno (i tipi interi con segno iniziano con i anziché u) che occupa 32 bit di spazio. La Tabella mostra i tipi interi incorporati in Rust. Possiamo usare una qualsiasi di queste varianti per dichiarare il tipo di un valore intero.

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128

Ogni variante può essere con segno o senza e ha una dimensione esplicita. Con segno e senza segno si riferisce alla possibilità che il numero sia negativo, in altre parole, se il numero deve avere un segno con sé (con segno) o se sarà sempre e solo positivo e potrà quindi essere rappresentato senza segno (senza segno).). È come scrivere numeri su carta: quando il segno conta, un numero viene indicato con il segno più o con il segno meno; tuttavia, quando è lecito ritenere che il numero sia positivo, viene visualizzato senza segno. I numeri con segno vengono memorizzati utilizzando la rappresentazione del complemento a due.

È possibile scrivere valori letterali interi in qualsiasi forma mostrata nella Tabella seguente. Tieni presente che i valori letterali numerici che possono essere più tipi numerici consentono un suffisso di tipo, ad esempio 57u8, per designare il tipo. I valori letterali numerici possono anche utilizzare _ come separatore visivo per rendere il numero più facile da leggere, ad esempio 1_000, che avrà lo stesso valore come se avessi specificato 1000.

Number literals	**Example**	Unsigned
Decimal	98_222	u8
Hex	0xff	u16
Octal	0o77	u32
Binary	0b1111_0000	u64
Byte (u8 only)	b'A'	u128

Tipi a virgola mobile

Rust ha anche due tipi primitivi per i numeri a virgola mobile, che sono numeri con punti decimali. I tipi a virgola mobile di Rust sono f32 e f64, che hanno rispettivamente una dimensione di 32 bit e 64 bit. Il tipo predefinito è f64 perché sulle CPU moderne ha più o meno la stessa velocità di f32 ma è capace di maggiore precisione. Tutti i tipi a virgola mobile sono con segno.

Esempio di operazioni

```
fn main() {
    // addition
```

```
    let sum = 5 + 10;

    // subtraction
    let difference = 95.5 - 4.3;

    // multiplication
    let product = 4 * 30;

    // division
    let quotient = 56.7 / 32.2;
    let truncated = -5 / 3; // Results in -1

    // remainder
    let remainder = 43 % 5;
}
```

Il tipo booleano

Come nella maggior parte degli altri linguaggi di programmazione, un tipo booleano in Rust ha due possibili valori: true e false. I booleani hanno la dimensione di un byte. Il tipo booleano in Rust viene specificato utilizzando bool. Per esempio:

```
fn main() {
    let t = true;

    let f: bool = false; // with explicit type annotation
}
```

Il tipo di carattere

Il tipo Rust char è il tipo alfabetico più primitivo della lingua. Ecco alcuni esempi di dichiarazione di valori char:

```
fn main() {
    let c = 'z';
    let z: char = ''; // with explicit type annotation
    let heart_eyed_cat = '';
}
```

Tieni presente che specifichiamo i valori letterali **char** con *virgolette singole*, al contrario dei valori letterali stringa, che utilizzano virgolette doppie. Il tipo char di Rust ha una dimensione di quattro byte e rappresenta un valore scalare Unicode, il che significa che può rappresentare molto più del semplice ASCII. Lettere accentate; Caratteri cinesi, giapponesi e coreani; emoji; e gli spazi a larghezza zero sono tutti valori char validi in Rust. I valori scalari Unicode vanno da U+0000 a U+D7FF e da U+E000 a U+10FFFF compreso. Tuttavia, un "carattere" non è realmente un concetto in Unicode, quindi la tua percezione umana su cosa sia un "carattere" potrebbe non corrispondere a cosa sia un carattere in Rust.

1.4.2 Tipi composti

I tipi composti possono raggruppare più valori in un unico tipo. Rust ha due tipi composti primitivi: tuple e array.

Il tipo tupla

Una tupla è un modo generale di raggruppare insieme un numero di valori con una varietà di tipi in un tipo composto. Le tuple hanno una lunghezza fissa: una volta dichiarate, non possono aumentare o ridursi di dimensione.
Creiamo una tupla scrivendo un elenco di valori separati da virgole all'interno di parentesi. Ogni posizione nella tupla ha un tipo e i tipi dei diversi valori nella tupla non devono essere gli stessi. Abbiamo aggiunto annotazioni di tipo facoltative in questo esempio:

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

La variabile *tup* si lega all'intera tupla perché una tupla è considerata un singolo elemento composto. Per ottenere i singoli valori da una tupla, possiamo utilizzare il pattern match per destrutturare un valore di tupla, in questo modo:

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {y}");
}
```

Questo programma crea prima una tupla e la associa alla variabile tup. Quindi utilizza un modello con let per prendere tup e trasformarlo in tre variabili separate, x, y e z. Questa operazione è chiamata **destrutturazione** perché spezza la singola tupla in tre parti.

Possiamo anche accedere direttamente a un elemento tupla utilizzando un punto (.) seguito dall'indice del valore a cui vogliamo accedere.

La tupla senza valori ha un nome speciale, unità. Questo valore e il tipo corrispondente sono entrambi scritti () e rappresentano un valore vuoto o un tipo restituito vuoto. Le espressioni restituiscono implicitamente il valore unitario se non restituiscono nessun altro valore.

Il tipo array

Un altro modo per avere una raccolta di più valori è utilizzare un array. A differenza di una tupla, ogni elemento di un array deve avere lo stesso tipo. A differenza degli array in altri linguaggi, gli array in Rust hanno una lunghezza fissa.
Scriviamo i valori in un array come un elenco separato da virgole all'interno di parentesi quadre:

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

Gli array sono utili quando vuoi che i tuoi dati siano allocati nello stack piuttosto che nell'heap o quando vuoi assicurarti di avere sempre un numero fisso di elementi. Tuttavia, un array non è flessibile come il tipo vettoriale. Un vettore è un tipo di raccolta simile fornito dalla libreria standard a cui è consentito

aumentare o ridurre le dimensioni. Se non sei sicuro se utilizzare un array o un vettore, probabilmente dovresti utilizzare un vettore.

Scrivi il tipo di un array utilizzando parentesi quadre con il tipo di ciascun elemento, un punto e virgola e quindi il numero di elementi nell'array, in questo modo:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

Qui, i32 è il tipo di ciascun elemento. Dopo il punto e virgola, il numero 5 indica che l'array contiene cinque elementi.

Puoi anche inizializzare un array in modo che contenga lo stesso valore per ciascun elemento specificando il valore iniziale, seguito da un punto e virgola, quindi la lunghezza dell'array tra parentesi quadre, come mostrato qui:

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

Accesso agli elementi dell'array

Un array è un singolo blocco di memoria di dimensione fissa e nota che può essere allocato nello stack. Puoi accedere agli elementi di un array utilizzando l'indicizzazione, in questo modo:

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

    let first = a[0];
    let second = a[1];
}
```

1.5 Funzioni

Funzione con iteratore

Scrivere una funzione Rust che utilizzi un iteratore può essere un po' più complicato che scrivere una semplice funzione, poiché richiede la comprensione del concetto di iteratori e come lavorare con essi.
Un iteratore è un oggetto che rappresenta una sequenza di valori che può essere ripetuto. In Rust, gli iteratori vengono implementati utilizzando il tratto Iterator, which definisce un insieme di metodi che devono essere implementati per creare un file personalizzato iteratore. Di seguito è riportato un esempio di una funzione Rust che accetta un iteratore di numeri interi come input e restituisce la somma di tutti gli elementi nell'iteratore:

```
fn sum_iterator(iter: impl Iterator<Item=i32>) -> i32 {
        let mut sum = 0;
        for x in iter {
                sum += x;
        }
        sum
}
```

Passaggio di funzioni come argomenti

Passare una funzione come argomento a un'altra funzione è utile quando vuoi riutilizzare una parte comune di codice in più posti o quando vuoi astrarre dal comportamento di una funzione.
Per passare una funzione come argomento, è necessario specificare i valori della funzione come tipo di argomento. La firma della funzione è costituita da il nome della funzione, i tipi di argomenti e il tipo restituito. Di seguito è riportato un esempio di una funzione Rust che accetta un'altra funzione come un argomento:

```
fn apply_twice<F>(f: F, x: i32) -> i32
        where F: Fn(i32) -> i32
{
        f(f(x))
}
```

Questa funzione, chiamata apply_twice, accetta una funzione f come primo argomento e un numero intero x come secondo argomento. Ha un parametro di tipo generico F, che rappresenta il tipo della funzione f. Il parametro type è vincolato dal tratto Fn, il che significa che f deve essere una funzione che assume un valore i32 come input e restituisce un valore i32.
Il corpo della funzione è costituito da una singola riga di codice che chiama la funzione f due volte sull'input x e restituisce il risultato. Per chiamare questa funzione da qualche altra parte nel tuo programma, dovresti usare il comando seguente sintassi:

```
let double = |x| x * 2;
let y = apply_twice(double, 5);
```

Ciò assegnerebbe il valore 20 alla variabile y, poiché (5 * 2) * 2 = 20.

1.5.1 Funzione nidificata (funzione all'interno di funzione)

Una funzione all'interno di una funzione, nota anche come funzione annidata, è una funzione definita all'interno di un'altra funzione. Le funzioni nidificate possono essere utili quando si desidera definire una funzione che viene utilizzata solo all'interno di uno specifico contesto o quando vuoi suddividere una funzione più grande in una più piccola.
Per definire una funzione annidata in Rust, definisci semplicemente la funzione all'interno del corpo di un'altra funzione utilizzando la parola chiave fn. La funzione annidata puo' accedere alle variabili e agli argomenti della funzione esterna, e può essere chiamata dall'interno della funzione esterna proprio come qualsiasi altra funzione.
Di seguito è riportato un esempio di una funzione Rust che contiene una funzione annidata:

```
fn outer(x: i32) -> i32 {
        fn inner(y: i32) -> i32 {
                x + y
        }
        inner(5)
}
```

1.5.2 Parametri

Possiamo definire le funzioni in modo che abbiano parametri, che sono variabili speciali che fanno parte della firma di una funzione. Quando una funzione ha parametri, puoi fornirle valori concreti per tali parametri. Tecnicamente, i valori concreti sono chiamati argomenti, ma nelle conversazioni informali le persone tendono a usare le parole parametro e argomento in modo intercambiabile sia per le variabili nella definizione di una funzione che per i valori concreti passati quando si chiama una funzione.

```
fn main() {
    another_function(5);
}

fn another_function(x: i32) {
    println!("The value of x is: {x}");
}
```

Nelle firme delle funzioni, è necessario dichiarare il tipo di ciascun parametro. Questa è una decisione deliberata nella progettazione di Rust: richiedere annotazioni di tipo nelle definizioni di funzione significa che il compilatore non ha quasi mai bisogno che tu le usi altrove nel codice per capire di che tipo intendi. Il compilatore è anche in grado di fornire messaggi di errore più utili se sa quali tipi si aspetta la funzione.

1.5.3 Dichiarazioni ed espressioni

I corpi delle funzioni sono costituiti da una serie di istruzioni che eventualmente terminano con un'espressione. Finora, le funzioni che abbiamo trattato non includevano un'espressione finale, ma hai visto un'espressione come parte di un'istruzione. Poiché Rust è un linguaggio basato sulle espressioni, questa è una distinzione importante da comprendere. Altre lingue non hanno le stesse distinzioni, quindi vediamo cosa sono le affermazioni e le espressioni e come le loro differenze influenzano l'insieme delle funzioni.

- Le **istruzioni** sono istruzioni che eseguono alcune azioni e non restituiscono un valore.
- Le **espressioni** restituiscono un valore risultante. Diamo un'occhiata ad alcuni esempi.

1.5.4 Funzioni con valori restituiti

Le funzioni possono restituire valori al codice che le chiama. Non nominiamo i valori restituiti, ma dobbiamo dichiararne il tipo dopo una freccia (**->**). In Rust, il valore restituito dalla funzione è sinonimo del valore dell'espressione finale nel blocco del corpo di una funzione. È possibile restituire anticipatamente una funzione utilizzando la parola chiave return e specificando un valore, ma la maggior parte delle funzioni restituisce implicitamente l'ultima espressione. Ecco un esempio di una funzione che restituisce un valore:

```
fn five() -> i32 {
    5
}

fn main() {
```

```
    let x = five();

    println!("The value of x is: {x}");
}
```

1.6 Flusso di controllo

La capacità di eseguire parte del codice a seconda che una condizione sia vera e di eseguire parte del codice ripetutamente mentre una condizione è vera sono elementi fondamentali nella maggior parte dei linguaggi di programmazione. I costrutti più comuni che consentono di controllare il flusso di esecuzione del codice Rust sono le espressioni if e i cicli.

1.6.1 Dichiarazione IF

La parola chiave if viene utilizzata per creare semplici test condizionali. Può essere utilizzato in congiunzione con la parola chiave else if e else.

```
use rand::Rng;
fn main() {
        let mut rng = rand::thread_rng();
        let n = rng.gen_range(-5..5);
        if n < 0 {
                println!("negative value");
        } else if n == 0 {
                println!("zero");
        } else {
                println!("positive value");
        }
}
```

Poiché if è un'espressione, possiamo usarla sul lato destro di un'istruzione let per assegnare il risultato a una variabile:

```
fn main() {
    let condition = true;
    let number = if condition { 5 } else { 6 };

    println!("The value of number is: {number}");
}
```

la funzione *five* non ha parametri e definisce il tipo del valore restituito, ma il corpo della funzione è un solitario 5 *senza punto e virgola* perché è un'espressione di cui vogliamo restituire il valore.

1.6.2 Dichiarazione LOOP

Viene creato un ciclo infinito con la parola chiave loop. Il ciclo è terminato con la parola chiave break. Uno degli usi di un ciclo è riprovare un'operazione che sai potrebbe non riuscire, come verificare se un thread ha completato il suo lavoro. Potrebbe anche essere necessario passare il risultato di tale operazione fuori dal ciclo al resto del codice. Per fare ciò, puoi aggiungere il valore che desideri venga restituito dopo l'espressione break che usi per interrompere il ciclo; quel

valore verrà restituito fuori dal ciclo in modo che tu possa usarlo, come mostrato qui:

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

    let result = loop {
        counter += 1;

        if counter == 10 {
            break counter * 2;
        }
    };

    println!("The result is {result}");
}
```

1.6.3 Dichiarazione WHILE

La parola chiave while viene utilizzata per creare un ciclo. Funziona finché non viene soddisfatta la condizione specificata.

```
fn main() {
        let mut x = 1;
        while x <= 10 {
                println!("{}", x);
                x += 1;
        }
        println!("x: {}", x);
}
```

1.6.4 Dichiarazione FOR

Con la parola chiave for, iteriamo su un intervallo o una raccolta di valori.
La sicurezza e la concisione dei cicli for li rendono il costrutto di ciclo più comunemente utilizzato in Rust. Anche nelle situazioni in cui desideri eseguire del codice un certo numero di volte, come nell'esempio del conto alla rovescia che utilizzava un ciclo while, la maggior parte dei Rustaceans utilizzerebbe un ciclo for. Il modo per farlo sarebbe utilizzare un Range, fornito dalla libreria standard, che genera tutti i numeri in sequenza iniziando da un numero e terminando prima di un altro numero. Ecco come apparirebbe il conto alla rovescia utilizzando un ciclo for e un altro metodo di cui non abbiamo ancora parlato, rev, per invertire l'intervallo:

```
fn main() {
    for number in (1..4).rev() {
        println!("{number}!");
    }
    println!("LIFTOFF!!!");
}
```

2 Ownership e Borrowing

La proprietà (**ownership**) è la caratteristica più unica di Rust e ha profonde implicazioni per il resto del linguaggio. Consente a Rust di fornire garanzie di sicurezza della memoria senza bisogno di un garbage collector, quindi è importante capire come funziona la proprietà. In questo capitolo parleremo della proprietà e di diverse funzionalità correlate: borrowing, slices, e come Rust dispone i dati in memoria.

2.1 Cos'e la Ownership

La proprietà è un insieme di regole che governano il modo in cui un programma Rust gestisce la memoria. Tutti i programmi devono gestire il modo in cui utilizzano la memoria del computer durante l'esecuzione. Alcune lingue dispongono di una funzione di garbage collection che ricerca regolarmente la memoria non più utilizzata durante l'esecuzione del programma; in altri linguaggi, il programmatore deve allocare e liberare esplicitamente la memoria. Rust utilizza un terzo approccio: la memoria è gestita attraverso un sistema di proprietà con un insieme di regole che il compilatore controlla. Se una qualsiasi delle regole viene violata, il programma non verrà compilato. Nessuna delle funzionalità di proprietà rallenterà il tuo programma mentre è in esecuzione.

Poiché la proprietà è un concetto nuovo per molti programmatori, ci vuole del tempo per abituarsi. La buona notizia è che più diventi esperto con Rust e le regole del sistema di proprietà, più facile sarà sviluppare in modo naturale un codice che sia sicuro ed efficiente.

Lo Stack e l'Heap

Molti linguaggi di programmazione non richiedono di pensare allo stack e all'heap molto spesso. Ma in un linguaggio di programmazione di sistema come Rust, il fatto che un valore sia nello stack o nell'heap influisce sul comportamento del linguaggio e sul motivo per cui devi prendere determinate decisioni.

Sia lo stack che l'heap sono parti di memoria disponibili per il codice da utilizzare in fase di esecuzione, ma sono strutturati in modi diversi. Lo stack memorizza i valori nell'ordine in cui li ottiene e rimuove i valori nell'ordine opposto. Questo viene definito *last in, first out*. Pensa a una pila di piatti: quando aggiungi altri piatti, li metti in cima alla pila e quando ti serve un piatto, ne togli uno dalla cima. Aggiungere o rimuovere i piatti dal centro o dal fondo non funzionerebbe altrettanto bene! L'aggiunta di dati viene definita inserimento nello stack *push*, mentre la rimozione dei dati viene definita estrazione dallo stack *pop*. Tutti i dati archiviati nello stack devono avere una dimensione nota e fissa. I dati con una dimensione sconosciuta in fase di compilazione o una dimensione che potrebbe cambiare devono invece essere archiviati nell'heap.

L'heap è meno organizzato: quando metti i dati nell'heap, richiedi una certa quantità di spazio. L'allocatore di memoria trova un punto vuoto nell'heap che sia sufficientemente grande, lo contrassegna come in uso e restituisce un puntatore, che è l'indirizzo di quella posizione. Questo processo è chiamato allocazione sull'heap e talvolta è abbreviato semplicemente come allocazione (l'inserimento di valori nello stack non è considerato allocazione). Poiché il puntatore all'heap ha una dimensione nota e fissa, è possibile archiviare il puntatore nello stack, ma quando si desiderano i dati effettivi è necessario seguire il puntatore.

Pensa di essere seduto in un ristorante. Quando entri, indichi il numero di persone nel tuo gruppo e l'host trova un tavolo vuoto adatto a tutti e ti conduce lì. Se qualcuno nel tuo gruppo arriva in ritardo, può chiederti dove sei stato seduto per trovarti.

Il push nello stack è più veloce dell'allocazione nell'heap perché l'allocatore non deve mai cercare un posto dove archiviare i nuovi dati; quella posizione è sempre in cima allo stack. In confronto, l'allocazione dello spazio sull'heap richiede più lavoro perché l'allocatore deve prima trovare uno spazio sufficientemente grande per contenere i dati e quindi eseguire la contabilità per prepararsi all'allocazione successiva.

L'accesso ai dati nell'heap è più lento dell'accesso ai dati nello stack perché è necessario seguire un puntatore per arrivarci. I processori attuali sono più veloci se utilizzano meno memoria. Continuando l'analogia, consideriamo un cameriere in un ristorante che prende ordini da molti tavoli. È più efficiente ricevere tutti gli ordini su un tavolo prima di passare al tavolo successivo. Prendere un ordine dal tavolo A, poi un ordine dal tavolo B, poi ancora uno da A e poi ancora uno da B sarebbe un processo molto più lento. Allo stesso modo, un processore può svolgere meglio il proprio lavoro se lavora su dati vicini ad altri dati (come sono nello stack) piuttosto che più lontani (come possono essere nell'heap).

Quando il codice chiama una funzione, i valori passati alla funzione (inclusi, potenzialmente, puntatori ai dati nell'heap) e le variabili locali della funzione vengono inseriti nello stack. Al termine della funzione, tali valori vengono estratti dallo stack.

Tenere traccia di quali parti del codice utilizzano quali dati nell'heap, ridurre al minimo la quantità di dati duplicati nell'heap e ripulire i dati inutilizzati nell'heap in modo da non esaurire lo spazio sono tutti problemi che la proprietà risolve. Una volta compresa la proprietà, non sarà necessario pensare molto spesso allo stack e all'heap, ma sapere che lo scopo principale della proprietà è gestire i dati dell'heap può aiutare a spiegare perché funziona in questo modo.

2.1.1 Regole di Ownership

Innanzitutto, diamo un'occhiata alle regole di proprietà. Tieni a mente queste regole mentre lavoriamo attraverso gli esempi che le illustrano:

- Ogni valore in Rust ha un proprietario.
- Può esserci un solo proprietario alla volta.
- Quando il proprietario esce dall'ambito, il valore verrà eliminato.

2.1.2 Ambito variabile

Ora che abbiamo superato la sintassi di base di Rust, non includeremo tutto il codice fn main() negli esempi, quindi se stai seguendo, assicurati di inserire manualmente i seguenti esempi all'interno di una funzione main. Di conseguenza, i nostri esempi saranno un po' più concisi, permettendoci di concentrarci sui dettagli reali piuttosto che sul codice standard.

Come primo esempio di proprietà, esamineremo la portata di alcune variabili. Un ambito è l'intervallo all'interno di un programma per il quale un elemento è valido. Prendi la seguente variabile:

```
let s = "hello";
```

La variabile *s* si riferisce a una stringa letterale, in cui il valore della stringa è codificato nel testo del nostro programma. La variabile è valida dal punto in cui è dichiarata fino alla fine dell'ambito corrente. Il Listato seguente mostra un programma con commenti che annotano dove la variabile s sarebbe valida.

```
{                      // s is not valid here, 'its not yet declared
    let s = "hello";   // s is valid from this point forward

    // do stuff with s
}                      // this scope is now over, and s is no longer valid
```

In altre parole, ci sono due punti importanti nel tempo qui:

- Quando s entra nell'ambito, è valido.
- Rimane valido finché non esce dal campo di applicazione.

A questo punto, la relazione tra gli ambiti e la validità delle variabili è simile a quella di altri linguaggi di programmazione. Ora svilupperemo questa comprensione introducendo il tipo String.

2.1.3 Il tipo String

Per illustrare le regole di ownership, abbiamo bisogno di un tipo di dati più complesso di quelli trattati precendentemente. I tipi trattati in precedenza hanno dimensioni note, possono essere archiviati nello stack e estratti dallo stack quando il loro ambito è terminato e possono essere copiati rapidamente e banalmente per creare una nuova istanza indipendente se un'altra parte del codice deve utilizzare lo stesso valore in un ambito diverso. Ma vogliamo esaminare i dati archiviati nell'heap ed esplorare come Rust sa quando ripulire tali dati, e il tipo String è un ottimo esempio.

Ci concentreremo sulle parti di String che riguardano la proprietà. Questi aspetti si applicano anche ad altri tipi di dati complessi, siano essi forniti dalla libreria standard o creati dall'utente.

Abbiamo già visto le stringhe letterali, in cui un valore stringa è codificato nel nostro programma. Le stringhe letterali sono convenienti, ma non sono adatte a tutte le situazioni in cui potremmo voler utilizzare del testo. Uno dei motivi è che sono immutabili. Un altro è che non tutti i valori di stringa possono essere conosciuti quando scriviamo il nostro codice: ad esempio, cosa succederebbe se volessimo prendere l'input dell'utente e memorizzarlo? Per queste situazioni, Rust ha un secondo tipo di stringa, String. Questa tipologia gestisce i dati allocati sull'heap e come tale è in grado di memorizzare una quantità di testo a noi sconosciuta in fase di compilazione. Puoi creare una stringa da una stringa letterale utilizzando la funzione from, in questo modo:

```
let s = String::from("hello");
```

L'operatore double colon **::** ci consente di assegnare un nome a questa particolare funzione from sotto il tipo String anziché utilizzare una sorta di nome come string_from.

Questo tipo di stringa può essere modificato:

```
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() appends a literal to a String
```

```
    println!("{}", s); // This will print `hello, world!`
```

2.1.4 Memoria e allocazione

Nel caso di una stringa letterale, ne conosciamo il contenuto in fase di compilazione, quindi il testo viene codificato direttamente nell'eseguibile finale. Questo è il motivo per cui le stringhe letterali sono veloci ed efficienti. Ma queste proprietà derivano solo dall'immutabilità della stringa letterale. Sfortunatamente, non possiamo inserire una porzione di memoria nel codice binario per ogni porzione di testo la cui dimensione è sconosciuta in fase di compilazione e la cui dimensione potrebbe cambiare durante l'esecuzione del programma.

Con il tipo String, per supportare una porzione di testo mutabile e espandibile, dobbiamo allocare una quantità di memoria sull'heap, sconosciuta in fase di compilazione, per contenere il contenuto. Questo significa:

- La memoria deve essere richiesta dall'allocatore di memoria in fase di esecuzione.
- Abbiamo bisogno di un modo per restituire questa memoria all'allocatore quando abbiamo finito con la nostra stringa.

La prima parte la facciamo noi: quando chiamiamo String::from, la sua implementazione richiede la memoria di cui ha bisogno. Questo è praticamente universale nei linguaggi di programmazione.

Tuttavia, la seconda parte è diversa. Nei linguaggi con un garbage collector (GC), il GC tiene traccia e ripulisce la memoria che non viene più utilizzata e non abbiamo bisogno di pensarci. Nella maggior parte dei linguaggi senza GC, è nostra responsabilità identificare quando la memoria non viene più utilizzata e chiamare il codice per liberarla esplicitamente, proprio come abbiamo fatto per richiederla. Farlo correttamente è stato storicamente un problema di programmazione difficile. Se dimentichiamo, sprecheremo memoria. Se lo facciamo troppo presto, avremo una variabile non valida. Se lo facciamo due volte, anche questo è un bug. Dobbiamo accoppiare esattamente un'allocazione con esattamente una libera.

Rust prende una strada diversa: la memoria viene restituita automaticamente una volta che la variabile che la possiede esce dall'ambito. Ecco una versione del nostro esempio di ambito che utilizza una stringa invece di una stringa letterale:

```
    {
        let s = String::from("hello"); // s is valid from this point
                                       // forward

        // do stuff with s
    }                                  // this scope is now over, and s
                                       // is no longer valid
```

C'è un punto naturale in cui possiamo restituire la memoria di cui la nostra String ha bisogno all'allocatore: quando s esce dall'ambito. Quando una variabile esce dall'ambito, Rust chiama per noi una funzione speciale. Questa funzione si chiama **drop** ed è dove l'autore di String può inserire il codice per restituire la memoria. Le chiamate Rust cadono automaticamente alla parentesi graffa di chiusura.

Questo modello ha un profondo impatto sul modo in cui viene scritto il codice Rust. Può sembrare semplice in questo momento, ma il comportamento del codice può essere inaspettato in situazioni più complicate quando vogliamo che più variabili utilizzino i dati che abbiamo allocato nell'heap. Esploriamo ora alcune di queste situazioni.

2.1.5 Variabili e dati che interagiscono con Move

Più variabili possono interagire con gli stessi dati in modi diversi in Rust. Diamo un'occhiata ad un esempio utilizzando un numero intero.

```
let x = 5;
let y = x;
```

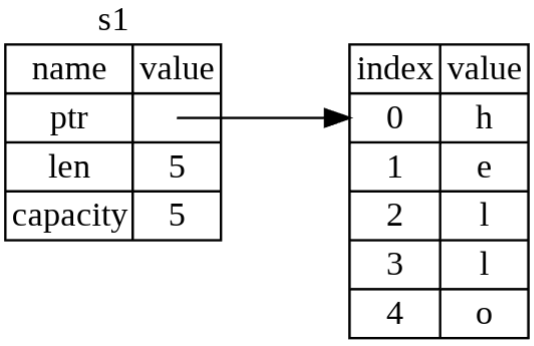
Probabilmente possiamo indovinare cosa sta facendo: "associare il valore 5 a x; quindi crea una copia del valore in x e collegala a y." Ora abbiamo due variabili, x e y, ed entrambe uguali a 5. Questo è effettivamente ciò che sta accadendo, perché gli interi sono valori semplici con una dimensione fissa e nota, e questi due valori 5 vengono inseriti nello stack.

Ora diamo un'occhiata alla versione String:

```
let s1 = String::from("hello");
let s2 = s1;
```

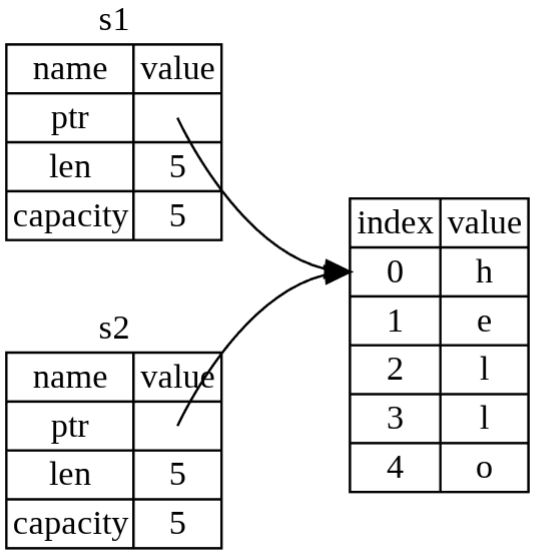
Sembra molto simile, quindi potremmo supporre che il modo in cui funziona sia lo stesso: ovvero, la seconda riga creerebbe una copia del valore in s1 e la collegherebbe a s2. Ma non è proprio ciò che accade.

Dai un'occhiata alla Figura seguente per vedere cosa sta succedendo a String sotto le coperte. Una String è composta da tre parti, mostrate a sinistra: un puntatore alla memoria che contiene il contenuto della stringa, una lunghezza e una capacità. Questo gruppo di dati viene archiviato nello stack. Sulla destra c'è la memoria sull'heap che contiene i contenuti.

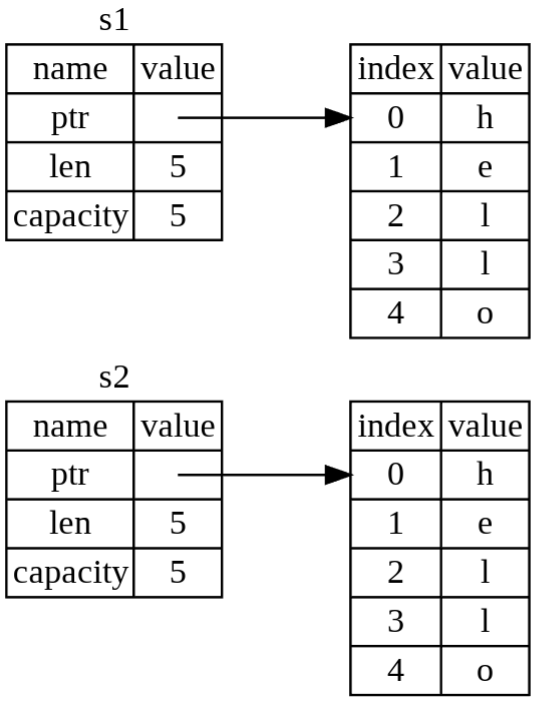


La lunghezza è la quantità di memoria, in byte, utilizzata attualmente dal contenuto della stringa. La capacità è la quantità totale di memoria, in byte, che String ha ricevuto dall'allocatore. La differenza tra lunghezza e capacità è importante, ma non in questo contesto, quindi per ora va bene ignorare la capacità.

Quando assegniamo s1 a s2, i dati String vengono copiati, ovvero copiamo il puntatore, la lunghezza e la capacità presenti nello stack. Non copiamo i dati sull'heap a cui fa riferimento il puntatore. In altre parole, la rappresentazione dei dati in memoria è simile alla figura:



La rappresentazione non assomiglia alla Figura precedente, che è come apparirebbe la memoria se Rust copiasse anche i dati dell'heap. Se Rust facesse questo, l'operazione s2 = s1 potrebbe essere molto costosa in termini di prestazioni di runtime se i dati sull'heap fossero di grandi dimensioni.



In precedenza, abbiamo detto che quando una variabile esce dall'ambito, Rust chiama automaticamente la funzione drop e ripulisce la memoria heap per quella variabile. Ma la Figura 4-2 mostra entrambi i puntatori di dati che puntano alla stessa posizione. Questo è un problema: quando s2 e s1 escono dall'ambito, entrambi proveranno a liberare la stessa memoria. Questo è noto come doppio errore gratuito ed è uno dei bug di sicurezza della memoria menzionati in precedenza. Liberare la memoria due volte può portare al danneggiamento della memoria, che può potenzialmente portare a vulnerabilità della sicurezza.

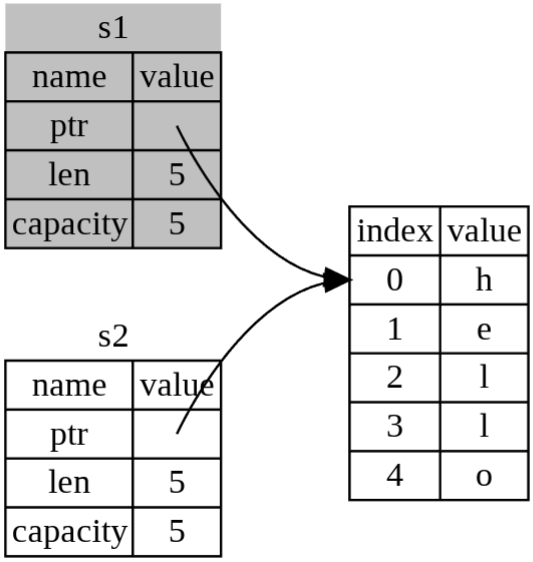
Per garantire la sicurezza della memoria, dopo la riga let s2 = s1;, Rust considera s1 non più valido. Pertanto, Rust non ha bisogno di liberare nulla quando s1 esce dall'ambito. Scopri cosa succede quando provi a utilizzare s1 dopo la creazione di s2; non funzionerà:

```
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{}, world!", s1);
```

Riceverai un errore come questo perché Rust ti impedisce di utilizzare il riferimento invalidato:

```
{footnotesize}
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement
  | the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes
  from the expansion of the macro `println` (in Nightly builds, run with -Z
  macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let s2 = s1.clone();
  |                ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` due to previous error
```

Se hai sentito i termini copia superficiale e copia profonda mentre lavori con altri linguaggi, il concetto di copiare il puntatore, la lunghezza e la capacità senza copiare i dati probabilmente suona come fare una copia superficiale. Ma poiché Rust invalida anche la prima variabile, invece di essere definita una copia superficiale, è nota come *move*. In questo esempio, diremmo che s1 è stato spostato in s2. Quindi, ciò che accade realmente è mostrato nella Figura seguente:



Questo risolve il nostro problema! Con solo s2 valido, quando esce dall'ambito, da solo libererà la memoria e il gioco è fatto.

Inoltre, c'è una scelta progettuale implicita in questo: Rust non creerà mai automaticamente copie "profonde" dei tuoi dati. Pertanto, si può presupporre che qualsiasi copia automatica sia poco costosa in termini di prestazioni di runtime.

2.1.6 Variabili e dati che interagiscono con Clone

Se vogliamo copiare in profondità i dati dell'heap di String, non solo i dati dello stack, possiamo utilizzare un metodo comune chiamato **clone**. Discuteremo la sintassi dei metodi nel prossimo capitolo, ma poiché i metodi sono una caratteristica comune in molti linguaggi di programmazione, probabilmente li avrete già visti.

Ecco un esempio del metodo clone in azione:

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

Funziona perfettamente e produce esplicitamente il comportamento mostrato nella Figura dove i dati dell'heap vengono copiati.
Quando vedi una chiamata a clone, sai che è in esecuzione del codice arbitrario e che il codice potrebbe essere costoso. È un indicatore visivo che sta succedendo qualcosa di diverso.

2.1.7 Stack-Only Data: Tratto Copy

C'è un'altra peculiarita' di cui non abbiamo ancora parlato. Questo codice che utilizza numeri interi, funziona ed è valido:

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

Ma questo codice sembra contraddire ciò che abbiamo appena imparato: non abbiamo una chiamata a clone, ma x è ancora valido e non è stato spostato in y.
Il motivo è che i tipi come gli interi che hanno una dimensione nota in fase di compilazione vengono archiviati interamente nello stack, quindi le copie dei valori effettivi sono veloci da creare. Ciò significa che non c'è motivo per cui vorremmo impedire che x sia valido dopo aver creato la variabile y. In altre parole, qui non c'è differenza tra copia profonda e superficiale, quindi chiamare clone non farebbe nulla di diverso dalla solita copia superficiale e possiamo tralasciarlo.
Rust ha un'annotazione speciale chiamata **tratto Copy** che possiamo posizionare sui tipi memorizzati nello stack, come lo sono gli interi. Se un tipo implementa il tratto Copy, le variabili che lo utilizzano non si spostano, ma vengono semplicemente copiate, rendendole ancora valide dopo l'assegnazione ad un'altra variabile.

Rust non ci permetterà di annotare un tipo con Copy se il tipo, o una qualsiasi delle sue parti, ha implementato il tratto Drop. Se il tipo necessita che accada qualcosa di speciale quando il valore esce dall'ambito e aggiungiamo l'annotazione Copy a quel tipo, otterremo un errore in fase di compilazione
Quindi, quali tipi implementano il tratto Copy?
Puoi controllare la documentazione per il tipo specificato per essere sicuro, ma come regola generale, qualsiasi gruppo di valori scalari semplici può implementare Copy e nulla che richieda l'allocazione o sia una qualche forma di risorsa può implementare Copy. Ecco alcuni dei tipi che implementano Copy:

- Tutti i tipi interi, come u32.
- Il tipo booleano, bool, con valori true e false.
- Tutti i tipi a virgola mobile, come f64.
- Il tipo di carattere, char.
- Tuple, se contengono solo tipi che implementano anche Copy. Ad esempio, (i32, i32) implementa Copy, ma (i32, String) no.

2.1.8 Proprietà e funzioni

I meccanismi per passare un valore a una funzione sono simili a quelli per assegnare un valore a una variabile. Passare una variabile a una funzione verrà spostata o copiata, proprio come fa l'assegnazione. Il Listato 4-3 contiene un esempio con alcune annotazioni che mostrano dove le variabili entrano ed escono dall'ambito.

```
fn main() {
    let s = String::from("hello");  // s entra nello scope

    takes_ownership(s);  // il valore di s viene mosso nella funzione...
                         // ... e quindi qui non è più valido

    let x = 5;           // x comes into scope

    makes_copy(x);       // x dovrebbe essere mosso nella funzione,
                         // ma i32 implementa Copy, so it's okay to still
                         // use x afterward
```

```
} // Here, x goes out of scope, then s. But because s's value
  // was moved, nothing special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Qui, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

Se provassimo a utilizzare s dopo la chiamata a takes_ownership, Rust genererebbe un errore in fase di compilazione. Questi controlli statici ci proteggono dagli errori. Prova ad aggiungere codice a main che utilizza s e x per vedere dove puoi usarli e dove le regole di proprietà ti impediscono di farlo.

2.1.9 Valori restituiti e ambito

I valori restituiti possono anche trasferire la proprietà.

```
fn main() {
    let s1 = gives_ownership();    // gives_ownership moves its return
                                   // value into s1

    let s2 = String::from("hello"); // s2 comes into scope

    let s3 = takes_and_gives_back(s2);  // s2 is moved into
                                        // takes_and_gives_back, which also
                                        // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {   // gives_ownership will move its
                                   // return value into the function
                                   // that calls it

    let some_string = String::from("yours"); // some_string entra into scope

    some_string                    // some_string is returned and
                                   // moves out to the calling
                                   // function
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
                                                      // scope

    a_string  // a_string is returned and moves out to the calling function
}
```

La proprietà di una variabile segue ogni volta lo stesso schema: assegnare un valore a un'altra variabile la sposta. Quando una variabile che include dati

nell'heap esce dall'ambito, il valore verrà cancellato per eliminazione a meno che la proprietà dei dati non sia stata spostata su un'altra variabile.

Anche se funziona, assumerne la proprietà e poi restituirla con ogni funzione è un po' noioso. Cosa succede se vogliamo consentire a una funzione di utilizzare un valore ma non di assumerne la proprietà? È piuttosto fastidioso che tutto ciò che passiamo debba essere restituito anche se vogliamo usarlo di nuovo, oltre a tutti i dati risultanti dal corpo della funzione che potremmo voler restituire.

Rust ci consente di restituire più valori utilizzando una tupla

```
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

Ma queste sono troppe cerimonie e molto lavoro per un concetto che dovrebbe essere comune. Fortunatamente per noi, Rust ha una funzionalità per utilizzare un valore senza trasferirne la proprietà, chiamata *referenze*.

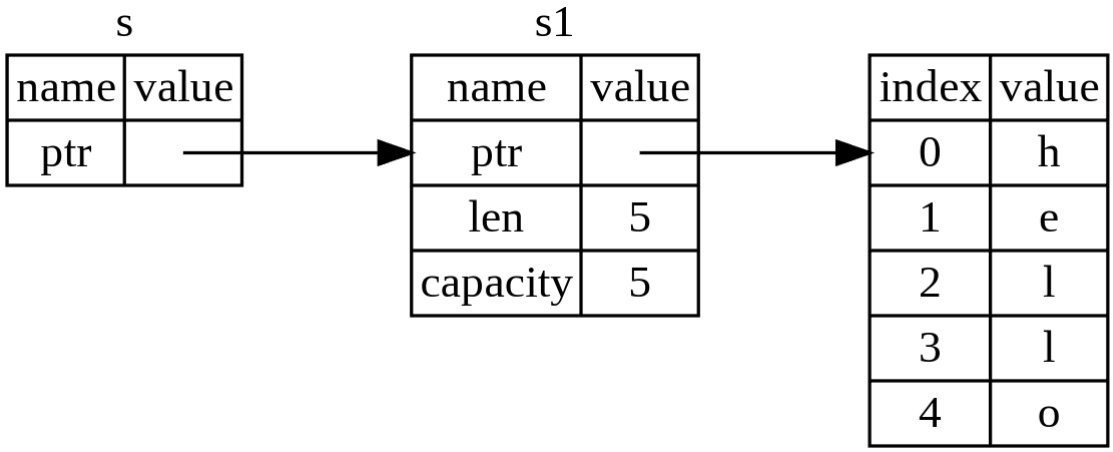
2.2 References e Borrowing

Il problema con il codice tupla nel listato precedente è che dobbiamo restituire la String alla funzione chiamante in modo da poter ancora utilizzare la String dopo la chiamata a calculate_length, perché la Stringa è stata spostata in calculate_length. Possiamo invece fornire un riferimento al valore String. Un riferimento è come un puntatore in quanto è un indirizzo che possiamo seguire per accedere ai dati archiviati in quell'indirizzo; quei dati sono posseduti da qualche altra variabile. A differenza di un puntatore, è garantito che un riferimento punti a un valore valido di un tipo particolare per la vita di quel riferimento.

Ecco come definiresti e utilizzeresti una funzione calculate_length che abbia un riferimento a un oggetto come parametro invece di assumere la proprietà del valore:

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

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

Innanzitutto, nota che tutto il codice tupla nella dichiarazione della variabile e il valore restituito dalla funzione sono scomparsi. In secondo luogo, nota che passiamo &s1 in calculate_length e, nella sua definizione, prendiamo &String anziché String. Queste e commerciali rappresentano riferimenti e consentono di fare riferimento a un valore senza assumersene la proprietà.



Nota: l'opposto del riferimento tramite & è il *dereferenziamento*, che si ottiene con l'operatore di dereferenziamento, *.

La sintassi &s1 ci consente di creare un riferimento che fa riferimento al valore di s1 ma non lo possiede. Poiché non ne è il proprietario, il valore a cui punta non verrà eliminato quando il riferimento smette di essere utilizzato.
Allo stesso modo, la firma della funzione calculate_length(s: &String) utilizza & per indicare che il tipo del parametro s è un riferimento. Aggiungiamo alcune annotazioni esplicative:

L'ambito in cui la variabile s è valida è lo stesso dell'ambito di qualsiasi parametro di funzione, ma il valore a cui punta il riferimento non viene eliminato quando s smette di essere utilizzato, perché s non ha la proprietà. Quando le funzioni hanno riferimenti come parametri anziché valori effettivi, non avremo bisogno di restituire i valori per restituire la proprietà, perché non abbiamo mai avuto la proprietà.

Chiamiamo l'azione di creare un prestito di riferimento (reference borrowing). Come nella vita reale, se una persona possiede qualcosa, puoi prenderglielo in prestito. Quando hai finito, devi restituirlo. Non lo possiedi. Quindi, cosa succede se proviamo a modificare qualcosa che stiamo prendendo in prestito? Prova il codice seguente. Avviso spoiler: non funziona!

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

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}

Errore
```

```
error[E0596]: cannot borrow `*some_string` as mutable, as it is behind
a `&` reference
 --> src/main.rs:8:5
```

Proprio come le variabili sono immutabili per impostazione predefinita, lo sono anche i riferimenti. Non ci è consentito modificare qualcosa a cui abbiamo un riferimento.

2.2.1 Mutable References

Possiamo correggere il codice del Listato precedente per permetterci di modificare un valore preso in prestito con solo alcune piccole modifiche che utilizzano, invece, un riferimento mutabile:

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

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Per prima cosa cambiamo s in *mut*. Quindi creiamo un riferimento mutabile con *&mut* s dove chiamiamo la funzione change e aggiorniamo la firma della funzione per accettare un riferimento mutabile con some_string: &mut String. Ciò rende molto chiaro che la funzione di cambiamento muterà il valore preso in prestito.

I riferimenti mutabili hanno una grande limitazione: se hai un riferimento mutabile a un valore, non puoi avere altri riferimenti a quel valore.
La restrizione che impedisce più riferimenti modificabili agli stessi dati contemporaneamente consente la mutazione ma in modo molto controllato. È qualcosa con cui i nuovi Rustacei lottano perché la maggior parte delle lingue ti consente di mutare quando vuoi. Il vantaggio di avere questa restrizione è che Rust può prevenire corse di dati in fase di compilazione. Una gara di dati è simile a una condizione di competizione e si verifica quando si verificano questi tre comportamenti:

- Due o più puntatori accedono contemporaneamente agli stessi dati.
- Almeno uno dei puntatori viene utilizzato per scrivere nei dati.
- Non viene utilizzato alcun meccanismo per sincronizzare l'accesso ai dati.

Le gare di dati causano comportamenti indefiniti e possono essere difficili da diagnosticare e risolvere quando si tenta di rintracciarle in fase di esecuzione; Rust previene questo problema rifiutandosi di compilare codice con corse di dati!

Come sempre, possiamo utilizzare le parentesi graffe per creare un nuovo ambito, consentendo più riferimenti mutabili, ma non simultanei:

```
let mut s = String::from("hello");

{
    let r1 = &mut s;
} // r1 goes out of scope here, so we can make a new reference
  // with no problems.

let r2 = &mut s;
```

Rust applica una regola simile per combinare riferimenti mutabili e immutabili. Questo codice genera un errore:

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

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

Inoltre, non possiamo avere un riferimento mutabile mentre ne abbiamo uno immutabile con lo stesso valore.

Gli utenti di un riferimento immutabile non si aspettano che il valore cambi improvvisamente sotto di loro! Tuttavia, sono consentiti più riferimenti immutabili perché nessuno che sta solo leggendo i dati ha la capacità di influenzare la lettura dei dati da parte di qualcun altro.

Tieni presente che l'ambito di un riferimento inizia da dove viene introdotto e continua fino all'ultima volta in cui viene utilizzato il riferimento. Ad esempio, questo codice verrà compilato perché l'ultimo utilizzo dei riferimenti immutabili, println!, si verifica prima che venga introdotto il riferimento mutabile:

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{} and {}", r1, r2);
// variables r1 and r2 will not be used after this point

let r3 = &mut s; // no problem
println!("{}", r3);
```

Gli ambiti dei riferimenti immutabili r1 e r2 terminano dopo println! dove sono stati utilizzati l'ultima volta, ovvero prima che venga creato il riferimento mutabile r3. Questi ambiti non si sovrappongono, quindi questo codice è consentito: il compilatore può indicare che il riferimento non viene più utilizzato in un punto prima della fine dell'ambito.

Anche se a volte gli errori di prestito possono essere frustranti, ricorda che è il compilatore Rust a segnalare tempestivamente un potenziale bug (in fase di compilazione anziché in fase di runtime) e a mostrarti esattamente dove si trova il problema. Quindi non dovrai cercare di capire perché i tuoi dati non sono quelli che pensavi fossero.

2.2.2 Riferimenti pendenti

Nei linguaggi con puntatori, è facile creare erroneamente un puntatore sospeso, un puntatore che fa riferimento a una posizione nella memoria che potrebbe essere stata data a qualcun altro, liberando parte della memoria e preservando un puntatore a quella memoria. In Rust, al contrario, il compilatore garantisce che i riferimenti non saranno mai riferimenti pendenti: se si ha un riferimento ad alcuni dati, il compilatore si assicurerà che i dati non escano dall'ambito prima che lo faccia il riferimento ai dati.

Proviamo a creare un riferimento pendente per vedere come Rust li previene con un errore in fase di compilazione:

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}'

Heres the error:

> cargo run

error[E0106]: missing lifetime specifier
 --> src/main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected named lifetime parameter
```

Questo messaggio di errore si riferisce a una funzionalità che non abbiamo ancora trattato: la durata. Ma, se trascuriamo le parti relative alle vite, il messaggio contiene la chiave del motivo per cui questo codice è un problema:

Poiché s viene creato all'interno di dangle, una volta terminato il codice di dangle, s verrà deallocato. Ma abbiamo provato a restituirne un riferimento. Ciò significa che questo riferimento punterebbe a una stringa non valida. Non va bene! Rust non ce lo permetterà.

La soluzione qui è restituire direttamente la stringa:

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

- In qualsiasi momento, puoi avere un riferimento mutabile o un numero qualsiasi di riferimenti immutabili.
- Le referenze devono essere sempre valide.

2.3 Il tipo Slice

Le slice (sezioni) ti consentono di fare riferimento a una sequenza contigua di elementi in una raccolta anziché all'intera raccolta. Una sezione è una sorta di riferimento, quindi non ha proprietà.

Ecco un piccolo problema di programmazione: scrivi una funzione che prenda una stringa di parole separate da spazi e restituisca la prima parola che trova in quella stringa. Se la funzione non trova uno spazio nella stringa, l'intera stringa deve essere una parola, quindi deve essere restituita l'intera stringa.

Analizziamo come scriveremo la firma di questa funzione senza utilizzare le fette, per capire il problema che le slice risolveranno:

```
fn first_word(s: &String) -> ?
```

La funzione first_word ha una &String come parametro. Non vogliamo la proprietà, quindi va bene. Ma cosa dovremmo restituire? Non abbiamo davvero un modo per parlare di parte di una stringa. Potremmo però restituire l'indice della fine della parola, indicato da uno spazio.

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

Poiché dobbiamo esaminare la stringa elemento per elemento e verificare se un valore è uno spazio, convertiremo la nostra stringa in un array di byte utilizzando il metodo as_bytes.

```
    let bytes = s.as_bytes();
```

Successivamente, creiamo un iteratore sull'array di byte utilizzando il metodo iter:

```
    for (i, &item) in bytes.iter().enumerate() {
```

Discuteremo gli iteratori in modo più dettagliato dopo. Per ora, sappi che *iter* è un metodo che restituisce ogni elemento in una raccolta e che *enumerate* percorre il risultato di iter e restituisce invece ciascun elemento come parte di una tupla. Il primo elemento della tupla restituita da enumerate è l'indice e il secondo elemento è un riferimento all'elemento. Questo è un po' più conveniente che calcolare l'indice da soli.

Poiché il metodo enumerate restituisce una tupla, possiamo utilizzare i modelli per destrutturare quella tupla. Discuteremo maggiormente i pattern in un'altro capitolo. Nel ciclo for, specifichiamo un pattern che ha *i* come indice nella

tupla e &item per il singolo byte nella tupla. Poiché otteniamo un riferimento all'elemento da .iter().enumerate(), utilizziamo & nel pattern.

All'interno del ciclo for cerchiamo il byte che rappresenta lo spazio utilizzando la sintassi letterale byte. Se troviamo uno spazio, restituiamo la posizione. Altrimenti restituiamo la lunghezza della stringa utilizzando s.len().

```
        if item == b' ' {
            return i;
        }
    }

    s.len()
```

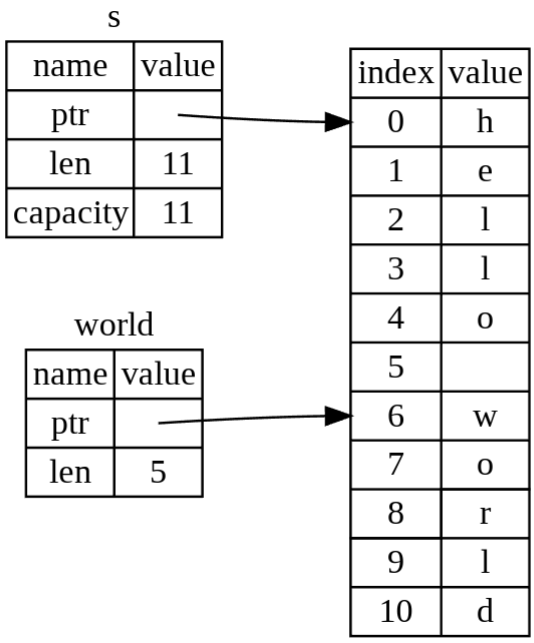
Ora abbiamo un modo per scoprire l'indice della fine della prima parola nella stringa, ma c'è un problema. Stiamo restituendo un usize da solo, ma è solo un numero significativo nel contesto di &String. In altre parole, poiché è un valore separato da String, non vi è alcuna garanzia che sarà ancora valido in futuro

2.3.1 String Slices

Una porzione di stringa è un riferimento a parte di una stringa e assomiglia a questa:

```
    let s = String::from("hello world");

    let hello = &s[0..5];
    let world = &s[6..11];
```

Piuttosto che un riferimento all'intera String, hello è un riferimento a una porzione della String, specificata nel bit extra [0..5]. Creiamo slices utilizzando un intervallo tra parentesi specificando [indice_iniziale..indice_finale], dove indice_iniziale è la prima posizione nella sezione e indice_finale è una posizione in più rispetto all'ultima posizione nella sezione. Internamente, la struttura dati della sezione memorizza la posizione iniziale e la lunghezza della sezione, che corrisponde a indice_finale meno indice_iniziale. Quindi, nel caso di let world = &s[6..11];, world sarebbe una slice che contiene un puntatore al byte nell'indice 6 di s con un valore di lunghezza pari a 5.



Con la sintassi **..** range di Rust, se vuoi iniziare dall'indice 0, puoi eliminare il valore prima dei due punti. In altre parole, sono uguali:

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

Allo stesso modo, se la tua porzione include l'ultimo byte della String, puoi eliminare il numero finale. Ciò significa che sono uguali:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

Puoi anche eliminare entrambi i valori per prendere una fetta dell'intera stringa. Quindi sono uguali:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

Nota: gli indici dell'intervallo di sezioni di stringhe devono trovarsi in corrispondenza dei limiti di caratteri UTF-8 validi. Se provi a creare una porzione di stringa nel mezzo di un carattere multibyte, il programma uscirà con un errore. Ai fini dell'introduzione delle porzioni di stringa, in questa sezione assumiamo ASCII solo; una discussione più approfondita sulla gestione di UTF-8 si trova nella sezione "Memorizzazione di testo codificato UTF-8 con stringhe".

Tenendo a mente tutte queste informazioni, riscriviamo first_word per restituire una slice. Il tipo che indica "slice stringa" è scritto come &str:

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

Otteniamo l'indice per la fine della parola nello stesso modo in cui abbiamo fatto nel Listato 4-7, cercando la prima occorrenza di uno spazio. Quando troviamo uno spazio, restituiamo una porzione di stringa utilizzando l'inizio della stringa e l'indice dello spazio come indici iniziale e finale.

Ora, quando chiamiamo first_word, otteniamo un singolo valore legato ai dati sottostanti. Il valore è costituito da un riferimento al punto iniziale della sezione e dal numero di elementi nella sezione.

2.3.2 Stringhe letterali come slice

Ricordiamo che abbiamo parlato delle stringhe letterali memorizzate all'interno del binario. Ora che conosciamo le fette, possiamo comprendere correttamente le stringhe letterali:

```
let s = "Ciao mondo!";
```

Il tipo di s qui è &str: è una slice che punta a quel punto specifico del binario. Questo è anche il motivo per cui i valori letterali delle stringhe sono immutabili; &str è un riferimento immutabile.

2.3.3 String Slices come Parametri

Sapere che puoi prendere porzioni di valori letterali e valori String ci porta a un ulteriore miglioramento su first_word, e questa è la sua firma:

```
fn first_word(s: &String) -> &str {
```

Un Rustacean più esperto scriverebbe invece la firma mostrata nel listato perché ci consente di utilizzare la stessa funzione sia sui valori &String che sui valori &str.

```
fn first_word(s: &str) -> &str {
```

Se abbiamo una porzione di stringa, possiamo passarla direttamente. Se abbiamo una String, possiamo passare una porzione della String o un riferimento alla String. Questa flessibilità trae vantaggio dalle *coercizioni derivanti (deref coercions)*.

Definire una funzione per prendere una porzione di stringa invece di un riferimento a una stringa rende la nostra API più generale e utile senza perdere alcuna funzionalità:

```
fn main() {
    let my_string = String::from("hello world");

    // `first_word` works on slices of `String`s, whether partial or whole
    let word = first_word(&my_string[0..6]);
    let word = first_word(&my_string[..]);
    // `first_word` also works on references to `String`s, which are equivalent
    // to whole slices of `String`s
    let word = first_word(&my_string);

    let my_string_literal = "hello world";

    // `first_word` works on slices of string literals, whether partial or whole
    let word = first_word(&my_string_literal[0..6]);
    let word = first_word(&my_string_literal[..]);

    // Because string literals *are* string slices already,
    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
}
```

2.3.4 Altre Slices

Le porzioni di stringa, come puoi immaginare, sono specifiche delle stringhe. Ma esiste anche un tipo di sezione più generale. Considera questa matrice:

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

\\Just as we might want to refer to part of a string,
\\we might want to refer to part of an array.
\\'Wed do so like this:

let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];

assert_eq!(slice, &[2, 3]);
```

3 Struct

Una **struct**, o struttura, è un tipo di dati personalizzato che consente di creare pacchetti e denominare più valori correlati che costituiscono un gruppo significativo. Se hai familiarità con un linguaggio orientato agli oggetti, una struttura è come gli attributi dei dati di un oggetto. In questo capitolo confronteremo e confronteremo le tuple con le strutture per sviluppare ciò che già conosci e dimostrare quando le strutture sono un modo migliore per raggruppare i dati.

Dimostreremo come definire e istanziare le strutture. Discuteremo come definire le funzioni associate, in particolare il tipo di funzioni associate chiamate metodi, per specificare il comportamento associato a un tipo di struttura. Strutture ed enumerazioni (discusse nel Capitolo 6) sono gli elementi costitutivi per la creazione di nuovi tipi nel dominio del programma per sfruttare appieno il controllo dei tipi in fase di compilazione di Rust.

3.1 Definire e Instanziare una Struct

Le strutture sono simili alle tuple, discusse nella sezione "Il tipo di tupla", in quanto entrambe contengono più valori correlati. Come le tuple, le parti di una struttura possono essere di tipo diverso. A differenza delle tuple, in una struttura nominerai ciascun dato in modo che sia chiaro cosa significano i valori. L'aggiunta di questi nomi significa che le strutture sono più flessibili delle tuple: non è necessario fare affidamento sull'ordine dei dati per specificare o accedere ai valori di un'istanza.

Per definire una struttura, inseriamo la parola chiave **struct** e diamo un nome all'intera struttura. Il nome di una struttura dovrebbe descrivere il significato dei dati raggruppati insieme. Quindi, all'interno delle parentesi graffe, definiamo i nomi e i tipi dei dati, che chiamiamo campi. Ad esempio, il listato seguente mostra una struttura che memorizza le informazioni su un account utente.

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

Per ottenere un valore specifico da una struttura, utilizziamo la notazione punto. Ad esempio, per accedere all'indirizzo email di questo utente, utilizziamo user1.email. Se l'istanza è mutabile, possiamo modificare un valore utilizzando la notazione punto e assegnandolo a un campo particolare. Il listato mostra come modificare il valore nel campo email di un'istanza User modificabile.

```
fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

Tieni presente che l'intera istanza deve essere modificabile; Rust non ci consente di contrassegnare solo determinati campi come mutabili. Come con qualsiasi espressione, possiamo costruire una nuova istanza della struttura come ultima espressione nel corpo della funzione per restituire implicitamente quella nuova istanza.

Il listato seguente mostra una funzione build_user che restituisce un'istanza

User con l'e-mail e il nome utente specificati. Il campo attivo ottiene il valore true e sign_in_count ottiene il valore 1.

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
```

Ha senso nominare i parametri della funzione con lo stesso nome dei campi della struttura, ma dover ripetere i nomi e le variabili dei campi email e nome utente è un po' noioso. Se la struttura avesse più campi, ripetere ogni nome diventerebbe ancora più fastidioso. Fortunatamente, c'è una comoda abbreviazione!

3.2 Utilizzare la scorciatoia Field Init

Poiché i nomi dei parametri e i nomi dei campi della struttura sono esattamente gli stessi nel listato, possiamo usare la sintassi abbreviata field init per riscrivere build_user in modo che si comporti esattamente allo stesso modo ma non abbia la ripetizione di nome utente ed email, come mostrato:

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
```

Qui stiamo creando una nuova istanza della struttura User, che ha un campo denominato email. Vogliamo impostare il valore del campo email sul valore nel parametro email della funzione build_user. Poiché il campo email e il parametro email hanno lo stesso nome, dobbiamo solo scrivere email anziché email: email.

3.3 Creazione di istanze da altre istanze con la sintassi di aggiornamento di Struct

Spesso è utile creare una nuova istanza di una struttura che includa la maggior parte dei valori di un'altra istanza, ma ne modifichi alcuni. Puoi farlo usando la *sintassi struct update*.

```
fn main() {
    // --snip--

    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
```

```
    };
}
```

Utilizzando la sintassi struct update, possiamo ottenere lo stesso effetto con meno codice, come mostrato nel Listato. La sintassi **..** specifica che i restanti campi non impostati esplicitamente dovrebbero avere lo stesso valore dei campi nell'istanza data.

```
fn main() {
    // --snip--

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

Il codice crea anche un'istanza in utente2 che ha un valore diverso per email ma ha gli stessi valori per i campi username, active e sign_in_count di utente1. La keyword ..user1 deve venire per ultimo per specificare che tutti i campi rimanenti dovrebbero ottenere i propri valori dai campi corrispondenti in user1, ma possiamo scegliere di specificare i valori per tutti i campi che vogliamo in qualsiasi ordine, indipendentemente dall'ordine dei campi in la definizione della struttura.

Si noti che la sintassi dell'aggiornamento della struttura utilizza = come un assegnamento; questo perché sposta i dati, proprio come abbiamo visto nella sezione "Variabili e dati che interagiscono con lo spostamento". In questo esempio, non possiamo più utilizzare user1 nel suo insieme dopo aver creato user2 perché la stringa nel campo nome utente di user1 è stata spostata in user2. Se avessimo fornito a utente2 nuovi valori String sia per l'e-mail che per il nome utente e quindi avessimo utilizzato solo i valori active e sign_in_count di utente1, utente1 sarebbe ancora valido dopo aver creato utente2. Sia active che sign_in_count sono tipi che implementano la caratteristica Copy, quindi si applicherebbe il comportamento discusso nella sezione "Dati solo stack: copia".

3.4 Utilizzo di strutture di tupla senza campi con nome per creare tipi diversi

Rust supporta anche strutture che sembrano simili alle tuple, chiamate tuple struct. Le strutture di tupla hanno il significato aggiunto fornito dal nome della struttura ma non hanno nomi associati ai relativi campi; piuttosto, hanno solo i tipi di campi. Le strutture di tupla sono utili quando si desidera dare un nome all'intera tupla e rendere la tupla di un tipo diverso dalle altre tuple e quando nominare ciascun campo come in una struttura regolare risulterebbe prolisso o ridondante.

Per definire una struttura di tupla, iniziare con la parola chiave struct e il nome della struttura seguiti dai tipi nella tupla. Ad esempio, qui definiamo e utilizziamo due strutture di tupla denominate Color e Point:

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
```

```
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

Tieni presente che i valori black e origin sono di tipo diverso perché sono istanze di strutture di tupla diverse. Ogni struttura definita ha il proprio tipo, anche se i campi all'interno della struttura potrebbero avere gli stessi tipi. Ad esempio, una funzione che accetta un parametro di tipo Color non può accettare un Point come argomento, anche se entrambi i tipi sono costituiti da tre valori i32. Altrimenti, le istanze di tuple struct sono simili alle tuple in quanto puoi destrutturarle nelle loro singole parti e puoi utilizzare un file . seguito dall'indice per accedere a un singolo valore.

3.5 Strutture simili a unità senza campi

Puoi anche definire strutture che non hanno campi! Queste sono chiamate strutture di tipo unitario perché si comportano in modo simile a (), il tipo di unità che abbiamo menzionato nella sezione "Il tipo di tupla". Le strutture di tipo unitario possono essere utili quando è necessario implementare un tratto su un tipo ma non si dispone di dati che si desidera archiviare nel tipo stesso. Discuteremo i tratti nel Capitolo 10. Ecco un esempio di dichiarazione e istanziazione di una struttura unitaria denominata AlwaysEqual:

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

Per definire AlwaysEqual, utilizziamo la parola chiave struct, il nome che vogliamo e quindi un punto e virgola. Non c'è bisogno di parentesi graffe o parentesi! Quindi possiamo ottenere un'istanza di AlwaysEqual nella variabile soggetto in un modo simile: utilizzando il nome che abbiamo definito, senza parentesi graffe o parentesi. Immagina che in seguito implementeremo il comportamento per questo tipo in modo tale che ogni istanza di AlwaysEqual sia sempre uguale a ogni istanza di qualsiasi altro tipo, magari per avere un risultato noto a scopo di test. Non avremmo bisogno di dati per implementare quel comportamento! Vedremo in seguito come definire i tratti e implementarli su qualsiasi tipo, comprese le strutture simili a unità.

Proprietà dei dati della struttura

Nella definizione della struttura User nel Listato 5-1, abbiamo utilizzato il tipo String posseduto anziché il tipo di slice stringa &str. Questa è una scelta deliberata perché vogliamo che ogni istanza di questa struttura possieda tutti i suoi dati e che tali dati siano validi finché è valida l'intera struttura.

È anche possibile che le strutture memorizzino riferimenti a dati posseduti da qualcos'altro, ma per farlo è necessario l'uso di lifetimes, una funzionalità di Rust di cui parleremo nel capitolo 10. Lifetimes garantisce che i dati a cui fa riferimento una struct siano validi per finché lo è la struttura. Diciamo che provi a memorizzare un riferimento in una struttura senza specificare la durata, come la seguente; questo non funzionerà:

```
struct User {
```

```
    active: bool,
    username: &str,
    email: &str,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        active: true,
        username: "someusername123",
        email: "someone@example.com",
        sign_in_count: 1,
    };
}

error[E0106]: missing lifetime specifier
 --> src/main.rs:4:12
  |
```

4 Un programma di esempio che utilizza le strutture

Per capire quando potremmo voler utilizzare le strutture, scriviamo un programma che calcoli l'area di un rettangolo. Inizieremo utilizzando variabili singole, quindi effettueremo il refactoring del programma finché non utilizzeremo invece le strutture.

Creiamo un nuovo progetto binario con Cargo chiamato rettangoli che prenderà la larghezza e l'altezza di un rettangolo specificate in pixel e calcolerà l'area del rettangolo. Il Listato mostra un breve programma con un modo per fare esattamente questo nel file src/main.rs del nostro progetto.

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Questo codice riesce a calcolare l'area del rettangolo chiamando la funzione area con ciascuna dimensione, ma possiamo fare di più per rendere questo codice chiaro e leggibile.

Il problema con questo codice è evidente nella firma dell'area:

```
fn area(width: u32, height: u32) -> u32 {
```

La funzione area dovrebbe calcolare l'area di un rettangolo, ma la funzione che abbiamo scritto ha due parametri e non è chiaro da nessuna parte nel nostro programma che i parametri siano correlati. Sarebbe più leggibile e più gestibile raggruppare insieme larghezza e altezza. Abbiamo già discusso un modo in cui potremmo farlo nella sezione "Il tipo di tupla" del Capitolo 3: utilizzando le tuple.

4.1 Refactoring with Tuples

Il Listato seguente mostra un'altra versione del nostro programma che utilizza le tuple.

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

In un certo senso, questo programma è migliore. Le tuple ci permettono di aggiungere un po' di struttura e ora stiamo passando solo un argomento. Ma in un altro modo, questa versione è meno chiara: le tuple non danno un nome ai propri elementi, quindi dobbiamo indicizzare le parti della tupla, rendendo il nostro calcolo meno ovvio.
Mescolare larghezza e altezza non avrebbe importanza per il calcolo dell'area, ma se volessimo disegnare il rettangolo sullo schermo, sarebbe importante! Dovremmo tenere presente che la larghezza è l'indice della tupla 0 e l'altezza è l'indice della tupla 1. Questo sarebbe ancora più difficile per qualcun altro capirlo e tenerlo a mente se dovesse usare il nostro codice. Poiché non abbiamo trasmesso il significato dei nostri dati nel nostro codice, ora è più facile introdurre errori.

4.2 Refactoring con struct: aggiungere più significato

Usiamo le strutture per aggiungere significato etichettando i dati. Possiamo trasformare la tupla che stiamo utilizzando in una struttura con un nome per l'intero e nomi per le parti, come mostrato nel listato.

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
```

```
    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

Qui abbiamo definito una struttura e l'abbiamo chiamata Rectangle. All'interno delle parentesi graffe, abbiamo definito i campi come larghezza e altezza, entrambi di tipo u32. Quindi, in main, abbiamo creato una particolare istanza di Rectangle che ha una larghezza di 30 e un'altezza di 50.

La nostra funzione d'area è ora definita con un parametro, che abbiamo chiamato rettangolo, il cui tipo è un prestito immutabile di un'istanza della struttura Rectangle. Come accennato nel capitolo 4, vogliamo prendere in prestito la struttura anziché assumerne la proprietà. In questo modo, main mantiene la sua proprietà e può continuare a utilizzare rect1, motivo per cui utilizziamo & nella firma della funzione e dove chiamiamo la funzione.

La funzione area accede ai campi larghezza e altezza dell'istanza Rectangle (nota che l'accesso ai campi di un'istanza di struct presa in prestito non sposta i valori del campo, motivo per cui spesso vedi prestiti di struct). La nostra firma della funzione per l'area ora dice esattamente cosa intendiamo: calcolare l'area di Rettangolo, utilizzando i campi larghezza e altezza. Ciò comunica che la larghezza e l'altezza sono correlate tra loro e fornisce nomi descrittivi ai valori anziché utilizzare i valori dell'indice di tupla pari a 0 e 1. Questo è un vantaggio per la chiarezza.

4.3 Aggiunta di funzionalità utili con tratti derivati

Sarebbe utile poter stampare un'istanza di Rectangle mentre stiamo eseguendo il debug del nostro programma e vedere i valori per tutti i suoi campi. Il Listato seguente prova a utilizzare la macro *println!* come abbiamo utilizzato nei capitoli precedenti. Tuttavia, questo non funzionerà.

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {}", rect1);
}
```

Quando compiliamo questo codice, riceviamo un errore con questo messaggio principale:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

La macro println! può eseguire molti tipi di formattazione e, per impostazione predefinita, le parentesi graffe indicano println! utilizzare la formattazione nota come Display: output destinato al consumo diretto da parte dell'utente finale. I tipi primitivi che abbiamo visto finora implementano Display per impostazione predefinita perché c'è solo un modo in cui vorresti mostrare un 1 o qualsiasi altro tipo primitivo a un utente. Ma con le strutture, è meno chiaro come println! dovrebbe formattare l'output perché ci sono più possibilità di visualizzazione: vuoi le virgole o no? Vuoi stampare le parentesi graffe? Dovrebbero essere visualizzati tutti i campi? A causa di questa ambiguità, Rust non tenta di indovinare ciò che vogliamo e le strutture non hanno un'implementazione fornita di Display da utilizzare con println! e il segnaposto .

Proviamolo! La macro println! alla macro ora apparirà come println!("rect1 is :?", rect1);. Mettendo lo specificatore **:?** all'interno delle parentesi graffe indica println! vogliamo utilizzare un formato di output chiamato Debug. La caratteristica Debug ci consente di stampare la nostra struttura in un modo che sia utile per gli sviluppatori in modo che possiamo vederne il valore mentre eseguiamo il debug del nostro codice.

Compila il codice con questa modifica. Maledizione! Riceviamo ancora un errore:

```
error[E0277]: `Rectangle` doesn't implement `Debug`

   = help: the trait `Debug` is not implemented for `Rectangle`
   = note: add `#[derive(Debug)]` to `Rectangle` or manually
   `impl Debug for Rectangle`
```

Rust include funzionalità per stampare informazioni di debug, ma dobbiamo esplicitamente optare per rendere tale funzionalità disponibile per la nostra struttura. Per fare ciò, aggiungiamo l'attributo esterno #[derive(Debug)] appena prima della definizione della struttura, come mostrato nel Listato.

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {:?}", rect1);
}
```

Ottimo! Non è l'output più carino, ma mostra i valori di tutti i campi per questa istanza, il che sarebbe sicuramente d'aiuto durante il debug. Quando abbiamo strutture più grandi, è utile avere un output un po' più facile da leggere; in

questi casi, possiamo usare **{:#?}** invece di **{:?}** nel println!. In questo esempio, utilizzando lo stile {:#?} verrà restituito quanto segue:

```
> cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle {
    width: 30,
    height: 50,
}
```

Un altro modo per stampare un valore utilizzando il formato Debug è utilizzare il formato macro **dbg!**, che assume la proprietà di un'espressione (al contrario di println!, che accetta un riferimento), stampa il file e il numero di riga in cui si trova quel dbg! la chiamata macro si verifica nel codice insieme al valore risultante di tale espressione e restituisce la proprietà del valore.

Nota

richiamare la macro dbg! stampa sul flusso della console di errore standard (stderr), al contrario di println!, che stampa sul flusso della console di output standard (stdout). Parleremo più approfonditamente di stderr e stdout nella sezione "Scrittura di messaggi di errore sull'errore standard invece che sull'output standard".

Ecco un esempio in cui siamo interessati al valore che viene assegnato al campo larghezza, nonché al valore dell'intera struttura in rect1:

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

Possiamo mettere dbg! attorno all'espressione scala 30 * e, poiché dbg! restituisce la proprietà del valore dell'espressione, il campo larghezza otterrà lo stesso valore come se non avessimo chiamato il dbg! lì. Non vogliamo DBG! per assumere la proprietà di rect1, quindi utilizziamo un riferimento a rect1 nella chiamata successiva.

```
> cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished dev [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/rectangles`
[src/main.rs:10] 30 * scale = 60
```

```
[src/main.rs:14] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

Oltre al tratto Debug, Rust ci ha fornito una serie di tratti da utilizzare con l'attributo derive che possono aggiungere comportamenti utili ai nostri tipi personalizzati. Questi tratti e i relativi comportamenti sono elencati nell'Appendice C. Parleremo di come implementare questi tratti con comportamenti personalizzati e di come creare i propri tratti nel Capitolo 10. Ci sono anche molti attributi diversi da derivati; per ulteriori informazioni, vedere la sezione "Attributi" del Rust Reference.

5 Sintassi dei Metodi

I metodi sono simili alle funzioni: li dichiariamo con la parola chiave fn e un nome, possono avere parametri e un valore restituito e contengono del codice che viene eseguito quando il metodo viene chiamato da qualche altra parte. A differenza delle funzioni, i metodi sono definiti nel contesto di una struttura (o di un oggetto enum o trait, di cui parleremo rispettivamente nei seguenti capitoli) e il loro primo parametro è sempre self, che rappresenta l'istanza della struttura dove viene invocato il metodo.

5.1 Definizione dei metodi

Modifichiamo la funzione area che ha un'istanza Rectangle come parametro e creiamo invece un metodo area definito sulla struttura Rectangle, come mostrato.

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Per definire la funzione nel contesto di Rectangle, iniziamo un blocco **impl (implementazione)** per Rectangle. Tutto all'interno di questo blocco impl sarà associato al tipo Rettangolo. Quindi spostiamo la funzione area all'interno delle parentesi graffe impl e modifichiamo il primo (e in questo caso l'unico) parametro in modo che sia self nella firma e ovunque all'interno del corpo. In main, dove abbiamo chiamato la funzione area e passato rect1 come argomento, possiamo invece usare la sintassi del metodo per chiamare il metodo area sulla nostra istanza Rectangle. La sintassi del metodo va dopo un'istanza: aggiungiamo un punto seguito dal nome del metodo, parentesi ed eventuali argomenti.

Nella firma per l'area usiamo &self invece del rettangolo: &Rectangle. &self è in realtà l'abbreviazione di self: &Self. All'interno di un blocco impl, il tipo Self è un alias per il tipo a cui è destinato il blocco impl. I metodi devono avere un parametro chiamato self di tipo Self come primo parametro, quindi Rust consente di abbreviarlo solo con il nome self nel primo punto del parametro. Nota che dobbiamo ancora usare & davanti alla scorciatoia self per indicare che questo metodo prende in prestito l'istanza Self, proprio come abbiamo fatto nel rettangolo: &Rectangle. I metodi possono assumere la proprietà del sé, prenderlo in prestito in modo immutabile, come abbiamo fatto qui, o prenderlo in prestito in modo mutabile, proprio come possono fare con qualsiasi altro parametro.

Abbiamo scelto &self qui per lo stesso motivo per cui abbiamo utilizzato &Rectangle nella versione con funzione: non vogliamo assumerne la proprietà e vogliamo solo leggere i dati nella struttura, non scrivervi. Se volessimo cambiare l'istanza su cui abbiamo chiamato il metodo come parte di ciò che fa il metodo, utilizzeremmo *&mut self* come primo parametro. Avere un metodo che assume la proprietà dell'istanza utilizzando solo self come primo parametro è raro; questa tecnica viene solitamente utilizzata quando il metodo trasforma self in qualcos'altro e si desidera impedire al chiamante di utilizzare l'istanza originale dopo la trasformazione.

Il motivo principale per cui si utilizzano metodi anziché funzioni, oltre a fornire la sintassi del metodo e a non dover ripetere il tipo di self nella firma di ogni metodo, è l'organizzazione. Abbiamo messo tutte le cose che possiamo fare con un'istanza di un tipo in un blocco impl invece di costringere i futuri utenti del nostro codice a cercare le funzionalità di Rectangle in vari punti della libreria che forniamo.

Nota che possiamo scegliere di dare a un metodo lo stesso nome di uno dei campi della struttura. Ad esempio, possiamo definire un metodo su Rectangle chiamato anche larghezza:

```
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
```

```
    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {}", rect1.width);
    }
}
```

Qui, scegliamo di fare in modo che il metodo width restituisca true se il valore nel campo width dell'istanza è maggiore di 0 e false se il valore è 0: possiamo utilizzare un campo all'interno di un metodo con lo stesso nome per qualsiasi scopo. In main, quando seguiamo rect1.width con parentesi, Rust sa che intendiamo la larghezza del metodo. Quando non usiamo le parentesi, Rust sa che intendiamo la larghezza del campo.

Spesso, ma non sempre, quando diamo a un metodo lo stesso nome di un campo vogliamo che restituisca solo il valore nel campo e non faccia nient'altro. Metodi come questo sono chiamati getter e Rust non li implementa automaticamente per i campi struct come fanno altri linguaggi. I getter sono utili perché puoi rendere il campo privato ma il metodo pubblico e quindi abilitare l'accesso in sola lettura a quel campo come parte dell'API pubblica del tipo.

5.2 Metodi con più parametri

Facciamo pratica con l'uso dei metodi implementando un secondo metodo sulla struttura Rectangle. Questa volta vogliamo che un'istanza di Rectangle prenda un'altra istanza di Rectangle e restituisca true se il secondo Rectangle può adattarsi completamente a self (il primo Rectangle); in caso contrario, dovrebbe restituire false. Cioè, una volta definito il metodo can_hold, vogliamo poter scrivere il programma mostrato nel Listato

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

Sappiamo che vogliamo definire un metodo, quindi sarà all'interno del blocco impl Rectangle. Il nome del metodo sarà can_hold e prenderà in prestito immutabile un altro rettangolo come parametro. Possiamo dire quale sarà il tipo del parametro osservando il codice che chiama il metodo: rect1.can_hold(&rect2) passa in &rect2, che è un prestito immutabile a rect2, un'istanza di Rectangle. Questo ha senso perché dobbiamo solo leggere rect2 (invece che scrivere, il che significherebbe che avremmo bisogno di un prestito mutabile) e vogliamo che

main mantenga la proprietà di rect2 in modo da poterlo riutilizzare dopo aver chiamato il metodo can_hold. Il valore restituito di can_hold sarà booleano e l'implementazione controllerà se la larghezza e l'altezza di self sono maggiori rispettivamente della larghezza e dell'altezza dell'altro rettangolo. Aggiungiamo il nuovo metodo can_hold al blocco impl

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

5.3 Funzioni associate

Tutte le funzioni definite all'interno di un blocco impl sono chiamate funzioni associate perché sono associate al tipo che prende il nome dal blocco impl. Possiamo definire funzioni associate che non hanno self come primo parametro (e quindi non sono metodi) perché non necessitano di un'istanza del tipo con cui lavorare. Abbiamo già utilizzato una funzione come questa: la funzione String::from definita nel tipo String.

Le funzioni associate che non sono metodi vengono spesso utilizzate per i costruttori che restituiranno una nuova istanza della struttura. Questi sono spesso chiamati *new*, ma new non è un nome speciale e non è integrato nel linguaggio. Ad esempio, potremmo scegliere di fornire una funzione associata denominata quadrato che abbia un parametro di dimensione e utilizzarlo sia come larghezza che come altezza, rendendo così più semplice creare un rettangolo quadrato anziché dover specificare lo stesso valore due volte:

```
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
```

Le parole chiave Self nel tipo restituito e nel corpo della funzione sono alias per il tipo che appare dopo la parola chiave impl, che in questo caso è Rectangle.

Per chiamare questa funzione associata, utilizziamo la sintassi :: con il nome della struttura; let sq = Rectangle::square(3); è un esempio. Questa funzione ha uno spazio dei nomi dalla struttura: la sintassi :: viene utilizzata sia per le funzioni associate che per gli spazi dei nomi creati dai moduli.

5.4 Blocchi impl multipli

Ogni struttura può avere più blocchi impl. Ad esempio:

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Non c'è motivo di separare questi metodi in più blocchi impl qui, ma questa è una sintassi valida. Vedremo un caso in cui più blocchi impl sono utili nel Capitolo 10, dove discuteremo di tipi e tratti generici.

6 Enumerazioni e corrispondenza di modelli

In questo capitolo esamineremo le enumerazioni, chiamate anche enums. Le enumerazioni consentono di definire un tipo enumerando le sue possibili varianti. Per prima cosa definiremo e utilizzeremo un'enumerazione per mostrare come un'enumerazione può codificare il significato insieme ai dati. Successivamente, esploreremo un'enumerazione particolarmente utile, chiamata Opzione, che esprime che un valore può essere qualcosa o niente. Poi vedremo come la corrispondenza dei modelli nell'espressione match semplifica l'esecuzione di codice diverso per valori diversi di un'enumerazione. Infine, tratteremo di come il costrutto if let sia un altro linguaggio comodo e conciso disponibile per gestire le enumerazioni nel codice.

6.1 Definizione di un'enumerazione

Laddove le strutture ti danno un modo per raggruppare campi e dati correlati, come un rettangolo con la sua larghezza e altezza, le enumerazioni ti danno un modo per dire che un valore è uno di un possibile insieme di valori. Ad esempio, potremmo voler dire che Rettangolo è una di un insieme di forme possibili che include anche Cerchio e Triangolo. Per fare ciò, Rust ci permette di codificare queste possibilità come un'enumerazione.

Diamo un'occhiata a una situazione che potremmo voler esprimere nel codice e vediamo perché le enumerazioni sono utili e più appropriate delle strutture in questo caso. Supponiamo che dobbiamo lavorare con gli indirizzi IP. Attualmente vengono utilizzati due standard principali per gli indirizzi IP: la versione quattro e la versione sei. Poiché queste sono le uniche possibilità per un indirizzo IP che il nostro programma incontrerà, possiamo enumerare tutte le possibili varianti, da qui il nome enumerazione.

Qualsiasi indirizzo IP può essere un indirizzo della versione quattro o della versione sei, ma non entrambi contemporaneamente. Questa proprietà degli indirizzi IP rende la struttura dei dati enum appropriata perché un valore enum può essere solo una delle sue varianti. Sia gli indirizzi della versione quattro che quelli della versione sei sono ancora fondamentalmente indirizzi IP, quindi dovrebbero essere trattati come lo stesso tipo quando il codice gestisce situazioni

che si applicano a qualsiasi tipo di indirizzo IP.

Possiamo esprimere questo concetto in codice definendo un'enumerazione IpAddrKind ed elencando i possibili tipi di indirizzo IP, V4 e V6. Queste sono le varianti dell'enum:

```
enum IpAddrKind {
    V4,
    V6,
}
```

IpAddrKind è ora un tipo di dati personalizzato che possiamo utilizzare altrove nel nostro codice.

6.2 Valori di enumerazione

Possiamo creare istanze di ciascuna delle due varianti di IpAddrKind in questo modo:

```
  let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;
```

Tieni presente che le varianti dell'enumerazione hanno uno spazio dei nomi sotto il suo identificatore e utilizziamo i due punti doppi per separarli. Ciò è utile perché ora entrambi i valori IpAddrKind::V4 e IpAddrKind::V6 sono dello stesso tipo: IpAddrKind. Possiamo quindi, ad esempio, definire una funzione che accetta qualsiasi IpAddrKind:

```
fn route(ip_kind: IpAddrKind) {}
```

E possiamo chiamare questa funzione con entrambe le varianti:

```
    route(IpAddrKind::V4);
    route(IpAddrKind::V6);
```

L'uso delle enumerazioni presenta ancora più vantaggi. Pensando più al nostro tipo di indirizzo IP, al momento non abbiamo un modo per memorizzare i dati effettivi dell'indirizzo IP; sappiamo solo di che tipo è. Dato che hai appena imparato a conoscere le strutture, potresti essere tentato di affrontare questo problema con le strutture come mostrato:

```
    enum IpAddrKind {
        V4,
        V6,
    }

    struct IpAddr {
        kind: IpAddrKind,
        address: String,
    }

    let home = IpAddr {
        kind: IpAddrKind::V4,
        address: String::from("127.0.0.1"),
    };
```

```
    let loopback = IpAddr {
        kind: IpAddrKind::V6,
        address: String::from("::1"),
    };
```

Qui abbiamo definito una struttura IpAddr che ha due campi: un campo kind che è di tipo IpAddrKind (l'enum che abbiamo definito in precedenza) e un campo indirizzo di tipo String. Abbiamo due istanze di questa struttura. Il primo è casa e ha il valore IpAddrKind::V4 nel suo genere con i dati dell'indirizzo associati di 127.0.0.1. La seconda istanza è il loopback. Ha l'altra variante di IpAddrKind come valore di tipo, V6, e ha l'indirizzo ::1 associato ad esso. Abbiamo utilizzato una struttura per raggruppare insieme i valori del tipo e dell'indirizzo, quindi ora la variante è associata al valore.

Tuttavia, rappresentare lo stesso concetto utilizzando solo un'enumerazione è più conciso: anziché un'enumerazione all'interno di una struttura, possiamo inserire i dati direttamente in ciascuna variante dell'enumerazione. Questa nuova definizione dell'enumerazione IpAddr afferma che entrambe le varianti V4 e V6 avranno valori String associati:

```
    enum IpAddr {
        V4(String),
        V6(String),
    }

    let home = IpAddr::V4(String::from("127.0.0.1"));

    let loopback = IpAddr::V6(String::from("::1"));
```

Alleghiamo i dati direttamente a ciascuna variante dell'enumerazione, quindi non è necessaria una struttura aggiuntiva. Qui è anche più semplice vedere un altro dettaglio di come funzionano le enumerazioni: il nome di ciascuna variante di enumerazione che definiamo diventa anche una funzione che costruisce un'istanza dell'enumerazione. Cioè, IpAddr::V4() è una chiamata di funzione che accetta un argomento String e restituisce un'istanza del tipo IpAddr. Otteniamo automaticamente questa funzione di costruzione definita come risultato della definizione dell'enum.

C'è un altro vantaggio nell'usare un'enumerazione piuttosto che una struttura: ogni variante può avere diversi tipi e quantità di dati associati. Gli indirizzi IP della versione quattro avranno sempre quattro componenti numerici che avranno valori compresi tra 0 e 255. Se volessimo memorizzare gli indirizzi V4 come quattro valori u8 ma esprimere comunque gli indirizzi V6 come un valore String, non saremmo in grado di farlo con una struttura . Le enumerazioni gestiscono questo caso con facilità:

```
    enum IpAddr {
        V4(u8, u8, u8, u8),
        V6(String),
    }

    let home = IpAddr::V4(127, 0, 0, 1);

    let loopback = IpAddr::V6(String::from("::1"));
```

Abbiamo mostrato diversi modi per definire strutture dati per memorizzare gli indirizzi IP della versione 4 e della versione 6. Tuttavia, a quanto pare, voler memorizzare indirizzi IP e codificarne il tipo è così comune che la libreria standard ha una definizione che possiamo usare!
Tieni presente che anche se la libreria standard contiene una definizione per IpAddr, possiamo comunque creare e utilizzare la nostra definizione senza conflitti perché non abbiamo incluso la definizione della libreria standard nel nostro ambito.

Diamo un'occhiata a un altro esempio di enum nel listato: questo ha un'ampia varietà di tipi incorporati nelle sue varianti.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

- Quit non ha alcun dato associato.
- Move ha campi con nome, come fa una struttura.
- Write include una singola stringa.
- ChangeColor include tre valori i32.

Definire un'enumerazione con varianti come queste è simile alla definizione di diversi tipi di definizioni di struct, tranne per il fatto che l'enumerazione non utilizza la parola chiave struct e tutte le varianti sono raggruppate insieme nel tipo Message. Le strutture seguenti potrebbero contenere gli stessi dati contenuti nelle varianti enum precedenti:

```
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

Ma se utilizzassimo le diverse strutture, ognuna delle quali ha il proprio tipo, non potremmo definire facilmente una funzione per ricevere uno qualsiasi di questi tipi di messaggi come potremmo con l'enumerazione Message, che è una unico tipo.

C'è un'altra somiglianza tra enumerazioni e strutture: proprio come siamo in grado di definire metodi sulle strutture utilizzando impl, siamo anche in grado di definire metodi sulle enumerazioni. Ecco un metodo chiamato call che potremmo definire nella nostra enumerazione Message:

```
    impl Message {
        fn call(&self) {
            // method body would be defined here
        }
    }
```

```
let m = Message::Write(String::from("hello"));
m.call();
```

Il corpo del metodo utilizzerebbe self per ottenere il valore su cui abbiamo chiamato il metodo. In questo esempio, abbiamo creato una variabile m che ha il valore Message::Write(String::from("hello")), e questo è ciò che self sarà nel corpo del metodo call quando esegue m.call( ).

Diamo un'occhiata a un'altra enumerazione nella libreria standard che è molto comune e utile: **Option**.

6.3 L'enumerazione delle opzioni e i suoi vantaggi rispetto ai valori nulli

Questa sezione esplora un caso di studio di *Option*, che è un'altra enumerazione definita dalla libreria standard. Il tipo Option codifica lo scenario molto comune in cui un valore potrebbe essere qualcosa o potrebbe essere nulla.

Ad esempio, se richiedi il primo elemento di un elenco non vuoto, otterrai un valore. Se richiedi il primo elemento in un elenco vuoto, non otterrai nulla. Esprimere questo concetto in termini di sistema di tipi significa che il compilatore può verificare se hai gestito tutti i casi che dovresti gestire; questa funzionalità può prevenire bug estremamente comuni in altri linguaggi di programmazione.

La progettazione del linguaggio di programmazione viene spesso pensata in termini di funzionalità da includere, ma anche le funzionalità da escludere sono importanti. Rust non ha la caratteristica nulla che hanno molti altri linguaggi. *Null* è un valore che significa che non c'è alcun valore. Nei linguaggi con null, le variabili possono sempre trovarsi in uno dei due stati: null o non null.

Nella sua presentazione del 2009 "Null References: The Billion Dollar Mistake", Tony Hoare, l'inventore di null, dice questo:

> Lo chiamo il mio errore da un miliardo di dollari. A quel tempo stavo progettando il primo sistema di tipi completo per i riferimenti in un linguaggio orientato agli oggetti. Il mio obiettivo era garantire che tutti gli usi dei riferimenti fossero assolutamente sicuri, con il controllo eseguito automaticamente dal compilatore. Ma non ho potuto resistere alla tentazione di inserire un riferimento nullo, semplicemente perché era così facile da implementare. Ciò ha portato a innumerevoli errori, vulnerabilità e arresti anomali del sistema, che probabilmente hanno causato un miliardo di dollari di dolore e danni negli ultimi quarant'anni.

Il problema con i valori null è che se provi a utilizzare un valore null come valore non null, otterrai un errore di qualche tipo. Poiché questa proprietà nulla o non nulla è pervasiva, è estremamente facile commettere questo tipo di errore. Tuttavia, il concetto che null sta cercando di esprimere è ancora utile: un valore null è un valore attualmente non valido o assente per qualche motivo.

Il problema non è tanto nel concetto quanto nella particolare implementazione. Pertanto, Rust non ha valori nulli, ma ha un'enumerazione che può codificare il concetto di valore presente o assente. Questa enumerazione è **Option<T>** ed è definita dalla libreria standard come segue:

```
enum Option<T> {
    None,
    Some(T),
}
```

L'enumerazione Option<T> è così utile che è persino inclusa nel preludio; non è necessario inserirlo esplicitamente nell'ambito. Le sue varianti sono incluse anche nel preludio: puoi usare Some e None direttamente senza il prefisso Option::. L'enumerazione Option<T> è ancora solo un'enumerazione regolare e Some(T) e None sono ancora varianti del tipo Option<T>.

La sintassi <T> è una caratteristica di Rust di cui non abbiamo ancora parlato. È un parametro di tipo generico e tratteremo i generici in modo più dettagliatou. Per ora, tutto ciò che devi sapere è che <T> significa che la variante Some dell'enumerazione Option può contenere un dato di qualsiasi tipo e che ogni tipo concreto utilizzato al posto di T rende il tipo Option<T> complessivo un tipo diverso. Ecco alcuni esempi di utilizzo dei valori di opzione per contenere tipi di numeri e tipi di stringa:

```
    let some_number = Some(5);
    let some_char = Some('e');

    let absent_number: Option<i32> = None;
```

Il tipo di some_number è Option<i32>. Il tipo di some_char è Option<char>, che è un tipo diverso. Rust può dedurre questi tipi perché abbiamo specificato un valore all'interno della variante Some. Per absent_number, Rust ci richiede di annotare il tipo Option generale: il compilatore non può dedurre il tipo che manterrà la variante Some corrispondente guardando solo un valore None. Qui diciamo a Rust che intendiamo che absent_number sia di tipo Option<i32>.

Quando abbiamo un valore Some, sappiamo che un valore è presente e che il valore è contenuto nel Some. Quando abbiamo un valore None, in un certo senso significa la stessa cosa di null: non abbiamo un valore valido. Allora perché avere Option<T> è meglio che avere null?

In breve, poiché Option<T> e T (dove T può essere qualsiasi tipo) sono tipi diversi, il compilatore non ci consentirà di utilizzare un valore Option<T> come se fosse sicuramente un valore valido. Ad esempio, questo codice non verrà compilato, perché sta tentando di aggiungere un i8 a un Option<i8>:

```
    let x: i8 = 5;
    let y: Option<i8> = Some(5);

    let sum = x + y;

error[E0277]: cannot add `Option<i8>` to `i8`
```

Intenso! In effetti, questo messaggio di errore significa che Rust non capisce come aggiungere un i8 e un Option<i8>, perché sono di tipi diversi. Quando abbiamo un valore di tipo come i8 in Rust, il compilatore si assicurerà che abbiamo sempre un valore valido. Possiamo procedere con sicurezza senza dover verificare la presenza di null prima di utilizzare quel valore. Solo quando abbiamo un Option<i8> (o qualsiasi tipo di valore con cui stiamo lavorando) dobbiamo

preoccuparci di non avere un valore e il compilatore si assicurerà che gestiamo quel caso prima di utilizzare il valore.

In altre parole, devi convertire un Option<T> in un T prima di poter eseguire operazioni T con esso. In genere, questo aiuta a individuare uno dei problemi più comuni con null: supporre che qualcosa non sia null quando in realtà lo è.

Eliminare il rischio di assumere erroneamente un valore nullo ti aiuta ad avere più fiducia nel tuo codice. Per avere un valore che può essere nullo, è necessario acconsentire esplicitamente rendendo il tipo di quel valore Option<T>. Quindi, quando usi quel valore, ti viene richiesto di gestire esplicitamente il caso in cui il valore è null. Ovunque un valore abbia un tipo che non è un Option<T>, puoi tranquillamente presupporre che il valore non sia null. Questa è stata una decisione progettuale deliberata di Rust per limitare la pervasività di null e aumentare la sicurezza del codice Rust.

Quindi, come si ottiene il valore T da una variante Some quando si ha un valore di tipo Option<T> in modo da poter utilizzare quel valore? L'enumerazione Option<T> dispone di un gran numero di metodi utili in una varietà di situazioni; puoi verificarli nella sua documentazione. Acquisire familiarità con i metodi su Option<T> sarà estremamente utile nel tuo viaggio con Rust.

In generale, per utilizzare un valore Option<T>, è necessario disporre di codice che gestisca ogni variante. Vuoi del codice che verrà eseguito solo quando hai un valore Some(T) e questo codice può utilizzare la T interna. Vuoi che qualche altro codice venga eseguito solo se hai un valore None. L'espressione **match** è un costrutto del flusso di controllo che fa proprio questo quando viene utilizzato con le enumerazioni: eseguirà codice diverso a seconda della variante dell'enumerazione di cui dispone e quel codice può utilizzare i dati all'interno del valore corrispondente.

6.4 Il costrutto del flusso di controllo della corrispondenza

Rust ha un costrutto del flusso di controllo estremamente potente chiamato **match** che consente di confrontare un valore con una serie di modelli e quindi eseguire il codice in base a quale modello corrisponde. I modelli possono essere costituiti da valori letterali, nomi di variabili, caratteri jolly e molte altre cose. La potenza della corrispondenza deriva dall'espressività dei modelli e dal fatto che il compilatore conferma che tutti i casi possibili sono stati gestiti.

Pensa a un'espressione di corrispondenza come a una macchina per la selezione delle monete: le monete scivolano lungo un binario con fori di varie dimensioni lungo di esso e ogni moneta cade attraverso il primo foro che incontra in cui si inserisce. Allo stesso modo, i valori attraversano ogni modello in una corrispondenza e al primo modello il valore "si adatta", il valore cade nel blocco di codice associato da utilizzare durante l'esecuzione.

A proposito di monete, usiamole come esempio usando il match! Possiamo scrivere una funzione che prende una moneta americana sconosciuta e, in modo simile alla macchina per contare, determina di quale moneta si tratta e restituisce il suo valore in centesimi

```
enum Coin {
    Penny,
```

```
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Analizziamo la corrispondenza nella funzione value_in_cents. Per prima cosa elenchiamo la parola chiave della corrispondenza seguita da un'espressione, che in questo caso è il valore moneta. Sembra molto simile a un'espressione condizionale usata con if, ma c'è una grande differenza: con if, la condizione deve essere valutata come un valore booleano, ma qui può essere di qualsiasi tipo. Il tipo di moneta in questo esempio è l'enumerazione Coin definita nella prima riga.

Poi ci sono le braccia del match. Un braccio è composto da due parti: un modello e del codice. Il primo braccio qui ha un pattern che è il valore Coin::Penny e poi l'operatore **=>** che separa il pattern e il codice da eseguire. Il codice in questo caso è solo il valore 1. Ogni braccio è separato dal successivo con una virgola.

Quando l'espressione match viene eseguita, confronta il valore risultante con il modello di ciascun braccio, in ordine. Se un modello corrisponde al valore, viene eseguito il codice associato a quel modello. Se lo schema non corrisponde al valore, l'esecuzione continua con il braccio successivo, proprio come in una macchina per la selezione delle monete. Possiamo avere tutte le braccia di cui abbiamo bisogno.

Il codice associato a ciascun braccio è un'espressione e il valore risultante dell'espressione nel braccio corrispondente è il valore che viene restituito per l'intera espressione di corrispondenza.

Solitamente non utilizziamo le parentesi graffe se il codice del braccio di corrispondenza è breve, come nel nostro esempio dove ogni braccio restituisce semplicemente un valore. Se desideri eseguire più righe di codice in un braccio di corrispondenza, devi utilizzare parentesi graffe e la virgola che segue il braccio è quindi facoltativa. Ad esempio, il codice seguente stampa "Lucky penny!" ogni volta che il metodo viene chiamato con Coin::Penny, ma restituisce comunque l'ultimo valore del blocco, 1:

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
```

```
        Coin::Quarter => 25,
    }
}
```

6.4.1 Modelli che si legano ai valori

Un'altra caratteristica utile delle match arm è che possono legarsi alle parti dei valori che corrispondono al modello. Ecco come possiamo estrarre valori dalle varianti enum.

Ad esempio, modifichiamo una delle nostre varianti enum per contenere i dati al suo interno. Dal 1999 al 2008, gli Stati Uniti hanno coniato quarti con disegni diversi per ciascuno dei 50 stati su un lato. Nessun'altra moneta ha ricevuto disegni statali, quindi solo i quarti hanno questo valore extra. Possiamo aggiungere queste informazioni alla nostra enumerazione modificando la variante Quarter per includere un valore UsState memorizzato al suo interno:

```
#[derive(Debug)] // so we can inspect the state in a minute
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

Immaginiamo che un amico stia cercando di raccogliere tutti i 50 quarter statali. Mentre ordiniamo gli spiccioli per tipo di moneta, chiameremo anche il nome dello stato associato a ciascun trimestre in modo che, se è uno che il nostro amico non ha, possa aggiungerlo alla sua collezione.

Nell'espressione di corrispondenza per questo codice, aggiungiamo una variabile chiamata state al modello che corrisponde ai valori della variante Coin::Quarter. Quando *Coin::Quarter* corrisponde, la variabile di stato si legherà al valore dello stato di quel trimestre. Quindi possiamo usare state nel codice per quel braccio, in questo modo:

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

Se dovessimo chiamare value_in_cents(Coin::Quarter(UsState::Alaska)), coin sarebbe Coin::Quarter(UsState::Alaska). Quando confrontiamo quel valore con ciascuna delle armi di corrispondenza, nessuna di esse corrisponde finché non raggiungiamo Coin::Quarter(state). A quel punto, l'associazione per lo stato sarà il valore UsState::Alaska. Possiamo quindi usare quel valore nel println! espressione, ottenendo così il valore dello stato interno dalla variante Coin enum per Quarter.

6.4.2 Matching con Option<T>

Nella sezione precedente, volevamo ottenere il valore T interno dal caso Some quando si utilizza Option<T>; possiamo anche gestire Option<T> usando match, come abbiamo fatto con l'enumerazione Coin! Invece di confrontare le monete, confronteremo le varianti di Option<T>, ma il modo in cui funziona l'espressione di corrispondenza rimane lo stesso.

Diciamo che vogliamo scrivere una funzione che accetta un'Option<i32> e, se c'è un valore al suo interno, aggiunge 1 a quel valore. Se non è presente un valore all'interno, la funzione dovrebbe restituire il valore None e non tentare di eseguire alcuna operazione.

Questa funzione è molto facile da scrivere, grazie a match, e sarà:

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

La combinazione di match ed enum è utile in molte situazioni. Vedrai spesso questo schema nel codice Rust: confronta un'enumerazione, associa una variabile ai dati all'interno e quindi esegui il codice basato su di essa. All'inizio è un po' complicato, ma una volta che ti sarai abituato desidererai averlo in tutti i linguaggi. È costantemente uno dei preferiti dagli utenti.

6.4.3 Modelli catch-all e segnaposto _

Utilizzando le enume, possiamo anche eseguire azioni speciali per alcuni valori particolari, ma per tutti gli altri valori eseguire un'azione predefinita. Immagina di implementare un gioco in cui, se ottieni 3 su un lancio di dado, il tuo giocatore non si muove, al contrario riceve un premio. Se ottieni un 7, il tuo giocatore perde il premio. Per tutti gli altri valori, il tuo giocatore muove quel numero di spazi sul tabellone. Ecco una corrispondenza che implementa quella logica, con il risultato del lancio del dado codificato anziché un valore casuale, e tutta l'altra logica rappresentata da funzioni senza corpi perché la loro effettiva implementazione non rientra nell'ambito di questo esempio:

```
let dice_roll = 9;
   match dice_roll {
       3 => add_fancy_hat(),
       7 => remove_fancy_hat(),
```

```
        other => move_player(other),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
    fn move_player(num_spaces: u8) {}
```

Per i primi due bracci, i modelli sono i valori letterali 3 e 7. Per l'ultimo braccio che copre ogni altro valore possibile, il modello è la variabile che abbiamo scelto di nominare altro. Il codice eseguito per l'altro braccio utilizza la variabile passandola alla funzione move_player.

Questo codice viene compilato, anche se non abbiamo elencato tutti i possibili valori che un u8 può avere, perché l'ultimo modello corrisponderà a tutti i valori non specificatamente elencati. Questo modello onnicomprensivo soddisfa il requisito secondo cui la corrispondenza deve essere esaustiva. Tieni presente che dobbiamo mettere il braccio catch-all per ultimo perché i modelli vengono valutati in ordine. Se mettessimo prima il braccio pigliatutto, gli altri bracci non funzionerebbero mai, quindi Rust ci avviserà se aggiungiamo le braccia dopo un pigliatutto!

Rust ha anche un pattern che possiamo usare quando vogliamo un catch-all ma non vogliamo usare il valore nel pattern catch-all: _ è un pattern speciale che corrisponde a qualsiasi valore e non si lega a quel valore. Questo dice a Rust che non utilizzeremo il valore, quindi Rust non ci avviserà di una variabile inutilizzata.

Cambiamo le regole del gioco: ora, se ottieni un risultato diverso da 3 o 7, devi lanciare nuovamente. Non abbiamo più bisogno di utilizzare il valore catch-all, quindi possiamo modificare il nostro codice per utilizzare _ invece della variabile denominata other:

```
    let dice_roll = 9;
    match dice_roll {
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => reroll(),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
    fn reroll() {}
```

Questo esempio soddisfa anche il requisito di esaustività perché stiamo esplicitamente ignorando tutti gli altri valori nell'ultimo braccio; non abbiamo dimenticato nulla.

Infine, cambieremo le regole del gioco ancora una volta in modo che non accada nient'altro nel tuo turno se ottieni qualcosa di diverso da un 3 o un 7. Possiamo esprimere ciò utilizzando il valore dell'unità (il tipo di tupla vuota di cui abbiamo parlato nella sezione "Il tipo di tupla") come il codice che accompagna il caso _:

```
    let dice_roll = 9;
    match dice_roll {
```

```
        3 => add_fancy_hat(),
        7 => remove_fancy_hat(),
        _ => (),
    }

    fn add_fancy_hat() {}
    fn remove_fancy_hat() {}
```

Qui, stiamo dicendo esplicitamente a Rust che non utilizzeremo nessun altro valore che non corrisponda a un modello in un braccio precedente, e in questo caso non vogliamo eseguire alcun codice.

6.4.4 Flusso di controllo conciso con if let

La sintassi if let ti consente di combinare if e let in un modo meno dettagliato per gestire i valori che corrispondono a un modello ignorando il resto. Considera il programma nel listato che corrisponde a un valore Option<u8> nella variabile config_max ma desidera eseguire codice solo se il valore è la variante Some.

```
    let config_max = Some(3u8);
    match config_max {
        Some(max) => println!("The maximum is configured to be {}", max),
        _ => (),
    }
```

Se il valore è Some, stampiamo il valore nella variante Some legando il valore alla variabile max nel modello. Non vogliamo fare nulla con il valore None. Per soddisfare l'espressione di corrispondenza, dobbiamo aggiungere _ => () dopo aver elaborato solo una variante, il che è fastidioso da aggiungere nel codice standard.

Invece, potremmo scriverlo in un modo più breve usando if let. Il codice seguente si comporta allo stesso modo della corrispondenza in:

```
    let config_max = Some(3u8);
    if let Some(max) = config_max {
        println!("The maximum is configured to be {}", max);
    }
```

La sintassi accetta uno schema e un'espressione separati da un segno di uguale. Funziona allo stesso modo di un match, dove l'espressione è data al match e il modello è il suo primo braccio. In questo caso, il modello è Some(max) e il massimo si lega al valore all'interno di Some. Possiamo quindi utilizzare max nel corpo del blocco if let nello stesso modo in cui abbiamo utilizzato max nel corrispondente braccio di corrispondenza. Il codice nel blocco if let non viene eseguito se il valore non corrisponde al modello.

Usare if let significa meno digitazione, meno rientro e meno codice standard. Tuttavia, si perde il controllo esaustivo imposto dalla corrispondenza. La scelta tra match e if let dipende da cosa stai facendo nella tua situazione particolare e se acquisire concisione è un compromesso appropriato per perdere il controllo esaustivo.
In altre parole, puoi pensare a if let come precisazione per una corrispondenza che esegue il codice quando il valore corrisponde a un modello e quindi ignora tutti gli altri valori.

Possiamo includere un else con un if let. Il blocco di codice che va con else è lo stesso del blocco di codice che va con il case _ nell'espressione match che è equivalente a if let e else. Ricordiamo la definizione dell'enumerazione Coin del listato precedente, dove anche la variante Quarter conteneva un valore UsState. Se volessimo contare tutte le monete diverse dai quarti che vediamo annunciando allo stesso tempo lo stato dei quarti, potremmo farlo con un'espressione match, come questa:

```
let mut count = 0;
match coin {
  Coin::Quarter(state) => println!("State quarter from {:?}!", state),
  _ => count += 1,
}
```

Oppure potremmo usare un'espressione if let and else, come questa:

```
    let mut count = 0;
    if let Coin::Quarter(state) = coin {
        println!("State quarter from {:?}!", state);
    } else {
        count += 1;
    }
```

Se ti trovi in una situazione in cui il tuo programma ha una logica troppo dettagliata per essere espressa utilizzando una match, ricorda che esiste anche let nel toolbox di Rust.

7 Gestire progetti in crescita con pacchetti, crates e moduli

Man mano che scrivi programmi di grandi dimensioni, l'organizzazione del codice diventerà sempre più importante. Raggruppando funzionalità correlate e separando il codice con funzionalità distinte, chiarirai dove trovare il codice che implementa una particolare funzionalità e dove andare per modificare il funzionamento di una funzionalità.

I programmi che abbiamo scritto finora erano contenuti in un modulo in un file. Man mano che un progetto cresce, dovresti organizzare il codice suddividendolo in più moduli e quindi in più file. Un pacchetto può contenere più crate binari e facoltativamente un crate di libreria. Man mano che un pacchetto cresce, puoi estrarre parti in contenitori separati che diventano dipendenze esterne. Questo capitolo tratta tutte queste tecniche. Per progetti molto grandi che comprendono una serie di pacchetti interconnessi che si evolvono insieme, Cargo fornisce spazi di lavoro.

Discuteremo anche i dettagli di implementazione dell'incapsulamento, che ti consente di riutilizzare il codice a un livello superiore: una volta implementata un'operazione, altro codice può richiamare il tuo codice tramite la sua interfaccia pubblica senza dover sapere come funziona l'implementazione. Il modo in cui scrivi il codice definisce quali parti sono pubbliche per essere utilizzate da altro codice e quali parti sono dettagli di implementazione privati che ti riservi il diritto di modificare. Questo è un altro modo per limitare la quantità di dettagli che devi tenere a mente.

Un concetto correlato è l'ambito: il contesto annidato in cui viene scritto il codice ha un insieme di nomi definiti come "nell'ambito". Durante la lettura, la scrittura e la compilazione del codice, programmatori e compilatori devono sapere se un particolare nome in un punto particolare si riferisce a una variabile, funzione, struttura, enum, modulo, costante o altro elemento e cosa significa quell'elemento. È possibile creare ambiti e modificare quali nomi rientrano o meno nell'ambito. Non puoi avere due elementi con lo stesso nome nello stesso ambito; sono disponibili strumenti per risolvere i conflitti di nomi.

Rust ha una serie di funzionalità che ti consentono di gestire l'organizzazione del tuo codice, inclusi quali dettagli sono esposti, quali dettagli sono privati e quali nomi sono in ciascun ambito nei tuoi programmi. Queste funzionalità, a volte indicate collettivamente come sistema di moduli, includono:

- **Pacchetti**: una funzionalità Cargo che ti consente di costruire, testare e condividere crates
- **Crates**: un albero di moduli che produce una libreria o un eseguibile
- **Moduli** e **use**: consentono di controllare l'organizzazione, l'ambito e la privacy dei percorsi
- **Paths**: un modo di denominare un elemento, ad esempio una struttura, una funzione o un modulo

7.1 Packages e Crates

Le prime parti del sistema di moduli che tratteremo sono i package e le crate.

Un crate è la più piccola quantità di codice che il compilatore Rust considera alla volta. Anche se si esegue rusticc anziché cargo e si passa un singolo file di codice sorgente, il compilatore considera quel file come un crate. I crate possono contenere moduli e i moduli possono essere definiti in altri file che vengono compilati con il crate, come vedremo nelle prossime sezioni.

Una crate può presentarsi in due forme: una *crate binaria* o una *crate della libreria*. I crate binari sono programmi che puoi compilare in un eseguibile che puoi eseguire, come un programma da riga di comando o un server. Ognuno deve avere una funzione chiamata main che definisce cosa succede quando viene eseguito l'eseguibile. Tutte le crate che abbiamo creato finora erano crate binarie.

Le *library crates* non hanno una funzione principale e non vengono compilati in un eseguibile. Invece, definiscono funzionalità destinate ad essere condivise con più progetti. Ad esempio, la crate rand che abbiamo utilizzato nelle sezioni iniziali fornisce funzionalità che generano numeri casuali. La maggior parte delle volte quando i Rustacei dicono "ctare", intendono una library crate, e usano "crate" in modo intercambiabile con il concetto generale di programmazione di una "library".

La root del crate è un file sorgente da cui parte il compilatore Rust e costituisce il modulo root del crate (spiegheremo i moduli in modo approfondito nella sezione "Definizione dei moduli per controllare l'ambito e la privacy").

Un pacchetto è un insieme di uno o più crate che fornisce una serie di funzionalità. Un pacchetto contiene un file Cargo.toml che descrive come costruire quelle crates. Cargo è in realtà un pacchetto che contiene il pacchetto binario per lo strumento da riga di comando che hai utilizzato per creare il tuo codice. Il pacchetto Cargo contiene anche un crate di libreria da cui dipende il crate binario. Altri progetti possono dipendere dal contenitore della libreria Cargo per utilizzare la stessa logica utilizzata dallo strumento da riga di comando Cargo.

Un pacchetto può contenere tutti i crate binari che desideri, ma al massimo un solo crate della libreria. Un pacchetto deve contenere almeno un crate, che si tratti di una libreria o di un crate binario.

Cargo segue una convenzione secondo cui src/main.rs è la radice del crate di un crate binario con lo stesso nome del pacchetto. Allo stesso modo, Cargo sa che se la directory del pacchetto contiene src/lib.rs, il pacchetto contiene un crate di libreria con lo stesso nome del pacchetto e src/lib.rs è la radice del crate. Cargo passa i file root del crate a rusticc per creare la libreria o il binario.

Un pacchetto può avere più crate binari posizionando i file nella directory src/bin: ogni file sarà un crate binario separato.

7.2 Definizione di moduli per il controllo dell'ambito e della privacy

In questa sezione parleremo dei moduli e di altre parti del sistema dei moduli, ovvero dei percorsi che consentono di nominare gli elementi; la parola chiave **use** che inserisce un percorso nell'ambito; e la parola chiave **pub** per rendere pubblici gli elementi. Discuteremo anche la parola chiave **as**, i pacchetti esterni e l'operatore **glob**.

Innanzitutto, inizieremo con un elenco di regole di facile riferimento quando organizzerai il tuo codice in futuro. Quindi spiegheremo ciascuna delle regole in dettaglio.

7.2.1 Module Cheat Sheet

Qui forniamo un rapido riferimento su come funzionano i moduli, i percorsi, la parola chiave **use** e la parola chiave **pub** nel compilatore e su come la maggior parte degli sviluppatori organizza il proprio codice. Nel corso di questo capitolo esamineremo esempi di ciascuna di queste regole, ma questo è un ottimo punto a cui fare riferimento per ricordare come funzionano i moduli.

- **Inizia dalla radice del crate**: quando compila un crate, il compilatore cerca prima nel file root del crate (di solito src/lib.rs per un crate di libreria o src/main.rs per un crate binario) per il codice da compilare.
- **Dichiarazione dei moduli**: nel file root del crate è possibile dichiarare nuovi moduli; diciamo, dichiari un modulo "garden" con mod garden;. Il compilatore cercherà il codice del modulo in questi posti:

  - I n linea, tra parentesi graffe che sostituiscono il punto e virgola dopo mod garden
  - N el file src/garden.rs
  - N el file src/garden/mod.rs

- **Dichiarazione di sottomoduli**: in qualsiasi file diverso dalla root del crate, puoi dichiarare i sottomoduli. Ad esempio, potresti dichiarare mod verdure; in src/garden.rs. Il compilatore cercherà il codice del sottomodulo all'interno della directory denominata per il modulo genitore in questi luoghi:

  - I n linea, direttamente dopo il mod verdure, tra parentesi graffe al posto del punto e virgola

  - N el file src/garden/verdure.rs

  - N el file src/garden/verdure/mod.rs

- **Percorsi del codice nei moduli**: una volta che un modulo fa parte del tuo crate, puoi fare riferimento al codice in quel modulo da qualsiasi altra parte dello stesso crate, purché le regole sulla privacy lo consentano, utilizzando il percorso del codice. Ad esempio, un tipo Asparagus nel modulo verdure dell'orto si troverebbe in crate::garden::verdure::Asparagus.

- **Privato vs pubblico**: il codice all'interno di un modulo è privato rispetto ai moduli principali per impostazione predefinita. Per rendere pubblico un modulo, dichiaralo con *pub mod* anziché *mod*. Per rendere pubblici anche gli elementi all'interno di un modulo pubblico, utilizzare *pub* prima delle loro dichiarazioni.

- **La parola chiave use**: all'interno di un ambito, la parola chiave use crea scorciatoie agli elementi per ridurre la ripetizione di percorsi lunghi. In qualsiasi ambito che può fare riferimento a crate::garden::vegetables::Asparagus, puoi creare una scorciatoia con use crate::garden::vegetables::Asparagus; e da quel momento in poi basterà scrivere Asparagus per utilizzare quel tipo nell'ambito.

Qui creiamo una crate binaria denominata backyard che illustra queste regole. La directory del crate, chiamata anche backyard, contiene questi file e directory:

```
backyard
├── Cargo.lock
├── Cargo.toml
└── src
    ├── garden
    │   └── vegetables.rs
    ├── garden.rs
    └── main.rs
```

Il file root del crate in questo caso è src/main.rs e contiene:

```
use crate::garden::vegetables::Asparagus;

pub mod garden;

fn main() {
    let plant = Asparagus {};
    println!("I'm growing {:?}!", plant);
}
```

La linea pub mod garden; dice al compilatore di includere il codice che trova in *src/garden.rs*, che è:

```
pub mod vegetables;
```

Qui pub mod garden; significa che é incluso anche il codice in *src/garden/vegetables.rs*

```
 #[derive(Debug)]
pub struct Asparagus {}
```

7.3 Raggruppamento del codice correlato in moduli

I moduli ci consentono di organizzare il codice all'interno di un contenitore per garantire leggibilità e facilitá di riutilizzo. I moduli ci consentono anche di controllare la privacy degli elementi, poiché il codice all'interno di un modulo è privato per impostazione predefinita. Gli elementi privati sono dettagli di implementazione interna non disponibili per l'uso esterno. Possiamo scegliere di rendere pubblici i moduli e gli elementi al loro interno, il che li espone per consentire l'utilizzo e la dipendenza del codice esterno da essi.

Ad esempio, scriviamo una libreria che fornisce la funzionalità di un ristorante. Definiremo le firme delle funzioni ma lasceremo vuoti i loro corpi per concentrarci sull'organizzazione del codice, piuttosto che sulla realizzazione di un ristorante.

Nel settore della ristorazione, alcune parti del ristorante vengono chiamate front of house e altre come back of house. La parte anteriore della casa è dove si trovano i clienti; questo comprende il luogo in cui i padroni di casa fanno sedere i clienti, i camarieri prendono gli ordini e i pagamenti e i baristi preparano le bevande. Il retro della casa è il luogo in cui chef e cuochi lavorano in cucina, le lavastoviglie puliscono e i manager svolgono il lavoro amministrativo.

Per strutturare il nostro crate in questo modo, possiamo organizzare le sue funzioni in moduli nidificati. Crea una nuova libreria denominata restaurant eseguendo cargo new restaurant –lib; quindi inserisci il codice in src/lib.rs per definire alcuni moduli e firme di funzioni. Ecco la sezione della reception:

```
Filename: src/lib.rs

mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
```

```
    }
}
```

Definiamo un modulo con la parola chiave mod seguita dal nome del modulo (in questo caso, front_of_house). Il corpo del modulo va quindi racchiuso tra parentesi graffe. All'interno dei moduli possiamo collocare altri moduli, come in questo caso con i moduli Hosting e Serving. I moduli possono anche contenere definizioni per altri elementi, come strutture, enumerazioni, costanti, tratti e funzioni.

Utilizzando i moduli, possiamo raggruppare insieme le definizioni correlate e nominare il motivo per cui sono correlate. I programmatori che utilizzano questo codice possono navigare nel codice in base ai gruppi anziché dover leggere tutte le definizioni, rendendo più semplice trovare le definizioni per loro rilevanti. I programmatori che aggiungono nuove funzionalità a questo codice saprebbero dove posizionare il codice per mantenere il programma organizzato.

In precedenza, abbiamo menzionato che src/main.rs e src/lib.rs sono chiamati radici dei crate. Il motivo del loro nome è che il contenuto di uno di questi due file forma un modulo chiamato crate alla radice della struttura del modulo del crate, noto come albero dei moduli.

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

Questo albero mostra come alcuni moduli si annidano uno dentro l'altro; ad esempio, ospitare nidi all'interno di front_of_house. L'albero mostra anche che alcuni moduli sono fratelli tra loro, nel senso che sono definiti nello stesso modulo; hosting e servizio sono fratelli definiti in front_of_house. Se il modulo A è contenuto all'interno del modulo B, diciamo che il modulo A è figlio del modulo B e che il modulo B è il genitore del modulo A. Si noti che l'intero albero dei moduli ha la sua radice nel modulo implicito denominato crate.

L'albero dei moduli potrebbe ricordarti l'albero delle directory del filesystem sul tuo computer; è un paragone molto azzeccato! Proprio come le directory in un filesystem, usi i moduli per organizzare il tuo codice. E proprio come i file in una directory, abbiamo bisogno di un modo per trovare i nostri moduli.