Capitolo 1

Per iniziare

In questo capitolo

- **Un piccolo programma**
- **Ciao mondo!**
- **Stampa di combinazioni di stringhe letterali**
- **Stampa di più righe di testo**
- **Stampa di numeri interi**
- **Commenti**

Un piccolo programma

Il più piccolo programma Rust valido è questo:

```
fn main(){}
```

Ovviamente non fa niente. Definisce semplicemente una funzione vuota denominata `main`. Per *funzione*, si intende un insieme di istruzioni che "fanno qualcosa", e alle quali è stato dato un nome. `fn` è l'abbreviazione di *function*, mentre `main` è il nome di questa funzione. La coppia di parentesi tonde contiene i possibili argomenti della funzione; in questo caso non ce ne sono. Infine, la coppia di parentesi graffe contiene le possibili istruzioni che compongono il corpo della funzione; in questo caso non ci sono istruzioni.

Quando viene lanciato un programma scritto in Rust, viene eseguita la sua funzione `main`. Se non c'è una funzione chiamata `main`, allora non si tratta di un programma completo, però potrebbe essere una libreria. In generale, le librerie possono contenere diversi punti di ingresso; invece, i programmi hanno un solo punto di ingresso, e per quelli scritti in Rust si tratta, appunto, della funzione `main`.

Per eseguire questo programma, dovete prima installare gli strumenti di sviluppo per Rust. Finora, esiste un'unica implementazione realmente affidabile degli strumenti di sviluppo per il linguaggio Rust, ed è quella sviluppata dalla Fondazione Rust, in collaborazione con la community degli sviluppatori. Sono però in corso sviluppi di strumenti alternativi, per cui in futuro potreste avere più scelte.

Gli strumenti di sviluppo della Fondazione Rust possono essere scaricati gratuitamente dal sito `www.rust-lang.org` facendo clic su *GET STARTED*. Le piattaforme di sviluppo supportate sono Linux, Windows e macOS. Per ogni piattaforma esistono tre versioni: *stable* (la versione stabile), *beta* e *nightly* ("notturna"). Vi consiglio di usare la versione *stable*: è un po' più vecchia, ma è anche la più collaudata e quella che dà maggiore garanzia di non introdurre drastici cambiamenti al linguaggio. Tutte queste versioni del programma dovrebbero essere utilizzate dalla riga di comando di un terminale.
Dopo l'installazione, per verificare quale versione è stata installata, digitate da riga di comando: `rustc --version`. La parola `rustc` significa "Rust compiler". Il codice in questo libro è stato verificato utilizzando la versione 1.75, ma probabilmente anche le versioni successive andranno bene.
Quando disporrete di un'installazione funzionante, potrete eseguire le seguenti azioni.

- Creare o scegliere una cartella in cui verranno archiviati i vostri esercizi di Rust.
- Utilizzando un qualsiasi editor di testo, creare in quella cartella un file chiamato `main.rs`, avente come contenuto il programma esempio riportato all'inizio di questo capitolo.
- Quel programma deve venire compilato da un comando in un terminale. Pertanto, nella cartella contenente il file, dovete digitare: `rustc main.rs`. Il prompt dovrebbe tornare quasi immediatamente, dopo aver creato un file chiamato `main` (in ambiente Windows si chiamerà `main.exe`). In effetti, il comando `rustc` ha compilato correttamente il file specificato; cioè l'ha letto, ha generato il codice macchina corrispondente e ha memorizzato questo codice macchina in un programma eseguibile nella stessa cartella.
- Per eseguire tale programma appena generato, stando sempre nel terminale, dovete digitare il comando `./main` (o semplicemente `main`, se siete in ambiente Windows). Il prompt tornerà immediatamente, poiché questo programma non fa niente.

Ciao mondo!

Vediamo come stampare del testo sul terminale. Modificate il programma precedente nel seguente:

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

Se tale codice viene compilato ed eseguito come prima, il programma stamperà sullo schermo le parole: `Ciao mondo!`.
Notate che la riga appena aggiunta contiene otto elementi di sintassi, noti come *token*. Esaminiamoli.

- `print`: è il nome di una macro definita nella libreria standard di Rust.
- Il simbolo `!` specifica che il nome precedente indica una macro. Senza tale simbolo, `print` indicherebbe invece una funzione. Ma nella libreria standard di Rust non esiste una funzione con questo nome, e quindi otterreste un errore di compilazione.
- La parentesi `(` inizia la lista degli argomenti della macro.

- Le doppie virgolette `"` iniziano la stringa letterale.
- `Ciao mondo!` è il contenuto della stringa letterale.
- Le doppie virgolette `"` terminano la stringa letterale.
- La parentesi `)` termina la lista degli argomenti della macro.
- Il punto e virgola finale, `;` termina l'istruzione.

Una macro è qualcosa di simile a una funzione: un blocco di codice Rust, cui si assegna un nome. Usando questo nome in un'invocazione di macro, ottenete l'esecuzione di tale codice. La differenza principale tra una funzione e una macro è la seguente.

- Concettualmente, una funzione esiste anche in fase di esecuzione, come sequenza separata di istruzioni, che costituisce il corpo della funzione. Ogni chiamata di tale funzione sposta il controllo dal codice chiamante a quella sequenza di istruzioni, per poi tornare al punto di chiamata quando si raggiunge la fine della funzione.
- Invece, una macro non esiste in fase di esecuzione; il compilatore non fa altro che copiare le istruzioni contenute nella macro in ogni punto in cui tale macro viene invocata.

Esaminiamo il significato dell'espressione "stringa letterale". La parola *stringa* significa "sequenza finita di caratteri, eventualmente comprensivi di spazi e punteggiatura". La parola *letterale* significa "specificata direttamente nel codice sorgente". Pertanto, una "stringa letterale" non è altro che una "sequenza finita di caratteri (eventualmente comprensivi di spazi e punteggiatura) specificati direttamente nel codice sorgente".
Una stringa non letterale, invece, è una variabile il cui valore in fase di esecuzione è una stringa. Il valore di tale stringa potrebbe essere letto da un file o digitato dall'utente o copiato da una stringa letterale, come vedremo quando parleremo delle variabili di tipo stringa.
La macro `print` inserisce nel programma alcune istruzioni che stampano sul terminale il testo usato come argomento della macro.
Rust distingue sempre tra lettere maiuscole e minuscole: si dice che è *case-sensitive*. Questo significa che, se si sostituisce una lettera maiuscola con una minuscola o viceversa, in qualunque punto di un programma, eccetto che nelle stringhe letterali e nei commenti, in genere ottenete un programma non più valido o comunque un programma con un comportamento differente. Invece, apportare tali modifiche all'interno di stringhe letterali consente sempre una compilazione corretta, ma è possibile che il comportamento del programma cambi. Per esempio, considerate questo programma:

```
fn Main() {}
```

Se provate a compilarlo, otterrete l'errore di compilazione `'main' function not found in crate 'main'` (ossia: "funzione `'main'` non trovata nel crate `'main'`"), poiché nel programma `main.rs` non è definita alcuna funzione avente nome `main` (con la `m` minuscola). La parola *crate* significa *programma o libreria*.
Per evitare di scrivere sempre `fn main {` all'inizio di tutti gli esempi, d'ora in poi, salvo quando specificato, supporremo che il codice di esempio si trovi sempre all'interno delle parentesi graffe della funzione `main`. Quindi, sia le parentesi graffe sia il testo che le precede verranno omessi.

Stampa di combinazioni di stringhe letterali

Invece di utilizzare una sola stringa letterale, potete stamparne più d'una, anche in una singola istruzione, in questo modo:

```
print!("{}, {}!", "Ciao", "mondo");
```

Questa istruzione, se racchiusa tra le parentesi graffe della funzione `main`, stamperà ancora: `Ciao mondo!`. In questo caso, la macro `print` riceve tre argomenti, separati da virgole. Tutti e tre tali argomenti sono stringhe letterali. La prima stringa, però, contiene due coppie di parentesi graffe (`{}`). Ogni coppia di graffe è un segnaposto, che indica la posizione in cui inserire un valore specificato di seguito nella stessa macro.
Quindi, la macro analizza gli argomenti successivi al primo e cerca di associare ciascuno di essi a una coppia di graffe all'interno del primo argomento. Se tale associazione ha successo, le graffe vengono sostituite dall'argomento corrispondente. Questo codice assomiglia alla seguente istruzione del linguaggio C:

```
printf("%s, %s!", "Ciao", "mondo");
```

Però c'è una differenza importante. Se provate a compilare:

```
print!("{}, !", "Ciao", "mondo");
```

ottenete l'errore di compilazione `argument never used` (ossia: "argomento mai utilizzato"), poiché il primo argomento della macro contiene un solo segnaposto, ma dopo il primo argomento ci sono due argomenti da stampare. Quindi, per l'ultimo argomento della macro, non c'è nessun segnaposto corrispondente.
D'altra parte, ottenete un errore anche se provate a compilare la seguente istruzione:

```
print!("{}, {}!", "Ciao");
```

In questo caso il messaggio d'errore è `2 positional arguments in format string, but there is 1 argument` (ossia: "2 argomenti posizionali nella stringa di formato, ma c'è 1 argomento"). Cioè, il primo argomento contiene due segnaposto, ma dopo il primo argomento c'è un solo argomento da stampare. Quindi, per il secondo segnaposto, non c'è un argomento corrispondente.
Invece, le istruzioni corrispondenti nel linguaggio C di solito si limitano a emettere un warning, e poi generano un programma compilato che si comporterà male.

Stampa di più righe di testo

Finora abbiamo scritto programmi che stampano una sola riga. Ma, anche usando una sola istruzione, si possono stampare più righe. Si fa in questo modo:

```
print!("Prima riga\nSeconda riga\nTerza riga\n");
```

Output:

```
Prima riga
Seconda riga
Terza riga
```

La sequenza di caratteri `\n` (dove la `n` sta per *newline*, ossia *nuova riga*) viene trasformata dal compilatore nella sequenza di caratteri che rappresenta il terminatore di riga per il sistema operativo utilizzato.
In effetti, è molto comune terminare un'istruzione di stampa con un passaggio alla nuova riga. Per farlo, potete aggiungere i due caratteri `\n` alla fine della riga, oppure potete utilizzare un'altra macro della libreria standard, `println`, il cui nome significa "print line" (ossia: "stampa riga"). Si usa in questo modo:

```
println!("testo della riga");
```

Questa istruzione equivale a:

```
print!("testo della riga\n");
```

Stampa di numeri interi

Se volete stampare il testo `Il mio numero è 140`, potete digitare:

```
print!("Il mio numero è 140");
```

Oppure, utilizzando un segnaposto e un argomento aggiuntivo:

```
print!("Il mio numero è {}", "140");
```

Oppure, rimuovendo le virgolette attorno al secondo argomento:

```
print!("Il mio numero è {}", 140);
```

Nell'ultima istruzione, il secondo argomento non è più una stringa letterale, bensì è un *numero intero letterale* o, in breve, un intero letterale.
Rispetto alle stringhe, gli interi sono un altro tipo di dati.
La macro `print` può utilizzare anche numeri interi per sostituire i corrispondenti segnaposto all'interno del suo primo argomento. Infatti, il compilatore, quando elabora la sequenza di tre caratteri `140` contenuta nel codice sorgente, la interpreta come un numero, espresso in formato decimale. Di conseguenza, genera il numero equivalente in formato binario, e salva quel numero binario nel programma eseguibile.
In fase di esecuzione, il programma carica dal file eseguibile quel numero binario; lo trasforma nella stringa di tre caratteri `140`, utilizzando la notazione decimale, quindi sostituisce il segnaposto con quella stringa, generando così la stringa da stampare; infine,

invia tale stringa al terminale. Questa procedura spiega, per esempio, perché se viene scritto il seguente programma:

```
print!("Il mio numero è {}", 000140);
```

il compilatore genera esattamente lo stesso programma eseguibile generato in precedenza, che ovviamente avrà lo stesso comportamento. Questo perché, quando la sequenza di sei caratteri del codice sorgente `000140` viene convertita nel formato binario, gli zeri iniziali vengono ignorati.
I tipi di argomenti possono anche essere misti. Provate questa istruzione:

```
print!("{} è {}", "Il mio numero", 140);
```

Verrà stampata la stessa riga di prima. Qui il primo segnaposto corrisponde a una stringa letterale, mentre il secondo corrisponde a un numero intero letterale.

Commenti

Come ogni altro linguaggio di programmazione, anche Rust consente di incorporare commenti nel codice. Provate il seguente codice:

```
// Questo programma
// stampa un numero.
print!("{}", 34);   // trenta quattro
/* print!("{}", 80);
*/
```

Output:

```
34
```

Le prime due righe iniziano con una coppia di barre (`//`). Tale coppia di caratteri indica l'inizio di un commento di riga, ovvero un commento che termina alla fine della riga stessa. Per scrivere un commento su più righe, la coppia di barre deve essere ripetuta a ogni riga del commento, come nella seconda riga del programma precedente. Solitamente si lascia uno spazio subito dopo la doppia barra, per migliorare la leggibilità.
Come vediamo nella terza riga, un commento di riga può iniziare anche dopo un'istruzione, solitamente con uno spazio di separazione.
C'è un altro tipo di commento, esemplificato nella quarta e quinta riga. Tale commento inizia con la coppia di caratteri `/*` e termina con la coppia `*/`. Dato che questo tipo di commento può estendersi su un blocco di più righe, viene chiamato *commento a blocco*.
I programmatori Rust solitamente evitano i commenti su più righe nel codice di produzione, utilizzando solo commenti su riga singola. I commenti su più righe vengono usati solo per escludere temporaneamente dalla compilazione una parte del codice. Questo ha il vantaggio che, quando si estraggono le righe di codice utilizzando uno strumento di ricerca o quando si confrontano diverse versioni del codice di un sistema, è immediatamente chiaro quali righe sono commenti e quali no.

I commenti di Rust sembrano identici ai moderni commenti del linguaggio C. In realtà, c'è una differenza importante tra i commenti dei due linguaggi: i commenti di Rust possono essere annidati, e devono essere annidati correttamente. Quelli in C non hanno tale requisito. In questo modo, potete prima escludere alcune porzioni di un file di codice sorgente dalla compilazione, quindi escluderne una porzione più grande contenente le porzioni più piccole. Ecco un commento valido in Rust, ma non in C:

```
/* Questo è /* un commento*/
valido, anche /* se /* contiene
commenti*/al suo */interno. */
```

Invece, questo codice genera l'errore di compilazione: `unterminated block comment` (ossia: "commento a blocco non terminato"):

```
/* Questo /* invece non è consentito in Rust,
mentre in C è tollerato (ma può generare un warning).*/
```