Basi di C

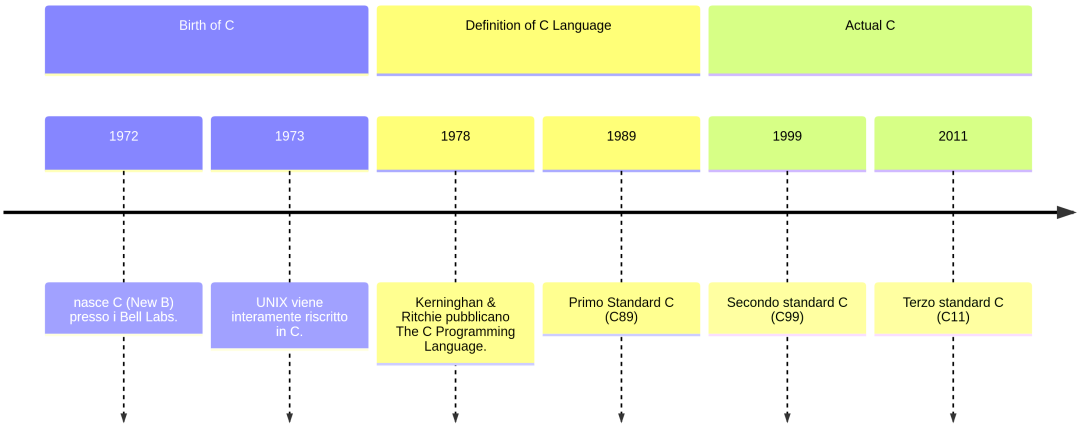
Matteo Spanio

12 marzo 2024

In questa lezione si introduce il linguaggio di programmazione C illustrando la sua storia e quali elementi del suo design lo rendano un linguaggio usato ancora dopo 50 anni dalla sua creazione. A seguire una descrizione della sintassi di C, introducendo anche elementi di I/O con esempi ed esercizi.

Il linguaggio C ha la fama di *linguaggio di basso livello* e sintassi complicata. Nel 1972 però, questo linguaggio compariva per la prima volta per semplificare la scrittura di codice, infatti a quel tempo i programmatori scrivevano soprattutto in Assembly. Vale la pena quindi ripercorrere i punti di passaggio che hanno compartecipato alla creazione e diffusione del linguaggio.

Storia e Standard



Principali momenti storici per lo sviluppo del linguaggio C

Dopo aver scritto l'intero sistema operativo UNIX in assembly i programmatori dei Bell Laboratories si accorsero che era difficile da mantenere[1], così si misero a lavorare su nuovi

[1] Per *codice difficile da mantenere* si intendono quei codici che hanno una struttura poco adatta all'aggiunta di nuove funzionalità nel tempo.

linguaggio, prima B e successivamente New B, ribattezzato poi C, per reimplementare del tutto UNIX in C nel 1973.

C risultò subito un linguaggio vanataggioso da usare soprattutto per la sua portabilità (di cui discuteremo più avanti in dettaglio), così, per tutti gli anni '70 e 80', il suo utilizzo si diffuse enormemente. In queste situazioni, ossia quando uno strumento di lavoro viene diffuso molto rapidamente e su larga scala, è una buona idea proporre uno Standard, cioè una serie di "regole" che chiariscano come vada usato in maniera appropriata lo strumento, in questo caso un modo comune di scrivere in C; nel 1978 Brian Kernigan e Dennis Ritchie (l'inventore di C) scrissero *The C Programming Language* (Kernighan e Ritchie 1978), il primo tutorial di C ma questo non era uno standard, ma negli anni 80 nacquero comunque molte varianti di C ognuna con un proprio modo di scrivere.

Uno Standard per tutti

> "The nice thing about standards is that you have so many to choose from.",
> **Andrew S. Tanenbaum**

È in questo scenario che si vede l'intervento dell'American National Standard Institute (ANSI) che a partire dal 1983 si mise al lavoro per proporre uno standard C concludendolo nel 1989 e successivamente approvato dall'International Organization for Standardization (ISO) nel 1990 come ISO/IEC 9899:1990, meglio conosciuto come C89. Negli anni vennero poi apportate altre migliorie al linguaggio che portarono, nel 1999, alla creazione di un nuovo standard: l'ISO/IEC 9899:1999, solitamente detto C99.

Dopo il C99 sono stati creati altri standard: C11, C17 e C23. Ognuno di questi introduce variazioni che servono a mantenere il linguaggio aggiornato alle esigenze dei programmatori attuali. Dopo la creazione di uno standard però i compilatori devono essere riscritti per comprendere le nuove regole proposte, questo fa sì che spesso non sia possibile compilare i programmi con l'ultimo standard perchè questo esiste solo da un punto di vista teorico. Attualmente i compilatori più diffusi coprono gli standard C99 e C11 ma non è ancora possibile scrivere programmi in C23.

Caratteristiche

In questa sezione si discutono brevemente le caratteristiche del linguaggio C evidenziandone pregi e difetti. L'aver concluso il precedente paragrafo parlando di compilatori lascia intuire che C sia un linguaggio **compilato**. Questo significa che il codice sorgente scritto in C deve essere tradotto in linguaggio macchina prima di poter essere eseguito. Altri punti caratteristici di C sono l'essere **minimale** (C rinuncia a molte astrazioni, questo, per esempio, permette di ridurre il numero di parole chiave del linguaggio), il fatto di essere **fortemente tipato** (le variabili devono essere dichiarate con un tipo e non possono cambiare tipo durante l'esecuzione del programma) e il fatto di essere **permissivo** (il compilatore non si lamenta di errori che in altri linguaggi sarebbero considerati gravi). Tecnicamente C è un linguaggio di alto livello, ma nella pratica ha molte feature che lo rendono uno dei linguaggi di più basso livello. Specialmente sulla gestione di memoria.

Compilare o interpretare?

Non tutti i linguaggi di programmazione vanno compilati. Alcuni vengono interpretati. La differenza, semplificando, è che un linguaggio compilato viene tradotto in linguaggio macchina una volta per tutte, mentre un linguaggio interpretato viene tradotto in linguaggio macchina ogni volta che viene eseguito. Esistono moltissimi linguaggi di entrambi i tipi, e non esiste una regola che renda un sistema migliore dell'altro. Generalmente i linguaggi compilati sono più veloci, ma i linguaggi interpretati sono più flessibili. Spesso per progetti di piccole dimensioni o script i programmatori preferiscono usare linguaggi interpretati, mentre per progetti di grandi dimensioni o software che deve essere veloce si preferisce usare linguaggi compilati (non è sempre così però).
L'alternarsi tra i due tipi di linguaggi è una questione che esiste da tempo. È importante notare a riguardo che il predecessore di C, il linguaggio B, era un linguaggio interpretato. La scelta di C di essere compilato è stata una delle ragioni del suo successo. Allo stesso modo molti linguaggi interpretati si sono diffusi proprio per essere tali.

Pregi e difetti

Alcuni linguaggi sono permissivi. Al programmatore basta avere solo un senso di base di come le cose funzionano. Gli errori nel codice vengono segnalati dal sistema di compilazione o di esecuzione e il programmatore può arrangiarsi e alla fine sistemare le cose in modo che funzionino correttamente. Il linguaggio C non è così.

Il modello di programmazione in C è che il programmatore sa esattamente cosa vuole fare e come utilizzare le possibilità del linguaggio per raggiungere quel obiettivo. Il linguaggio permette al programmatore esperto di esprimere ciò che desidera nel minor tempo possibile, rimanendo fuori dal suo cammino. C è "semplice" nel senso che il numero di componenti nel linguaggio è piccolo: se due funzionalità del linguaggio realizzano più o meno la stessa cosa, C ne includerà solo una. La sintassi di C è concisa e il linguaggio non limita ciò che è "consentito": il programmatore può praticamente fare ciò che desidera.

Il sistema di tipi di C e i controlli degli errori esistono solo durante la compilazione. Il codice compilato viene eseguito in un modello di esecuzione ridotto senza controlli di sicurezza per conversioni di tipo errate, indici di array errati o puntatori errati. Non c'è un *garbage collector* per gestire la memoria. Invece, il programmatore gestisce manualmente la memoria heap. Tutto ciò rende C veloce ma fragile.

The billion-dollar mistake

> "I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist

> the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.", **Tony Hoare**
>
> Seppure non sia una caratteristica introdotta con C, la null reference è un problema che affligge molti linguaggi di programmazione. Tony Hoare, scrivendo il linguaggio ALGOL W, introdusse questo concetto, C, ispirandosi fortemente ad ALGOL, ereditò questa caratteristica. L'utilizzo sbagliato di un puntatore nullo è una delle cause principali di errori in C.

Di seguito sono commentate le caratteristiche principali considerate come i maggiori pregi e difetti del C:

1. **Efficienza**: il linguaggio C è nato per essere efficiente, doveva andare a sostituire il codice Assembly, è stato scritto quindi per essere veloce;
2. **Compattezza**: il C è un linguaggio molto compatto, non ci sono molte parole chiave, non ci sono molte funzioni predefinite, non ci sono molte strutture dati predefinite, fornisce soltanto "lo stretto indispensabile";
3. **Portabilità**: sebbene il C non sia stato pensato per essere portabile, la creazione di uno standard e la sua associazione con UNIX ha reso il C un linguaggio solido e usato su molte piattaforme. Uno degli slogan che accompagna il C da molti anni è "write once, compile everywhere" (scrivi una volta, compila ovunque), che poi venne ripreso con l'arrivo di Java e del suo "write once, run everywhere";
4. **Permissività**: il C si basa su un'assunzione molto forte: *il programmatore sa cosa sta facendo*. Questo è sia un pro che un contro: da un lato permette al programmatore di fare tutto quello che vuole, dall'altro lato permette al programmatore di fare tutto quello che vuole. Questo significa che il programmatore può fare cose molto potenti, ma può anche fare cose molto pericolose.
5. **Error prone**: proprio per la permissività appena menzionata, c'è spazio per fare molti errori. Questo è un problema che si può risolvere con l'uso di strumenti di analisi statica e dinamica, con l'uso di buone pratiche di programmazione e con l'uso di commenti e documentazione.
6. **Difficile da leggere**: proprio per la compattezza, il C può diventare difficile da leggere. Esistono addirittura delle competizioni di programmazione in cui il C viene usato per scrivere codice illeggibile.
7. **Run time**: una volta compilato, il programma C perde molte informazioni, per esempio non è più possibile conoscere i tipi delle variabili, pertanto non è possibile fare controlli a run time.
8. **Non Object-oriented**: lo stretto indispensabile fornito da C non include gli oggetti. Questo significa che non esistono le astrazioni tipiche di Java, classi, variabili di istanza, metodi… Esistono evoluzioni del C con queste caratteristiche (C++, C# e lo stesso Java ha ereditato molto dal C).

Installazione

Per lavorare con il C, avrete bisogno di un compilatore C. Il compilatore più diffuso è `gcc`, è un compilatore open source, sviluppato da GNU. Storicamente lo sviluppo in C è stato fatto su sistemi Unix, quindi `gcc` è facilmente compatibile con tutti i sistemi Unix-like, mentre su Windows è necessario installare un ambiente di sviluppo come MinGW. Di seguito sono riportate le istruzioni per installare il compilatore su Linux e Mac. Per Windows, le istruzioni saranno caricate su Moodle.

> **Linux e Mac**[a]
>
> [a] probabilmente è già installato. Aprire il terminale per verificarlo con `gcc -v`
>
> **Debian/Ubuntu**[2]: per installare il compilatore: `sudo apt install gcc -y`
> **Mac**: se non lo avete, facendo `gcc -v` il sistema vi proporrà di scaricare i tool da linea di comando di Apple. Accettando, vi verrà installato.

> **Windows**
>
> potete installare MinGW e modificare le variabili di PATH. Le istruzioni saranno caricate su Moodle.

Per scrivere effettivamente il codice è poi richiesto di installare un editor di testo adatto al codice. Dal momento che i programmatori adorano sviluppare strumenti per la programmazione è pieno di editor di testo che più o meno si equivalgono e che, in ogni caso, ricoprono abbondantemente le necessità del corso. Alcuni esempi sono: notepad++, sublime, gedit, kate, emacs, nano, vim...

> **Importante**
>
> È sconsigliato studiare su un IDE, anche se è consigliato usarli quando si lavora effettivamente.

> **Impara facendo**
>
> Riscrivere tutti gli esempi di codice senza usare copia e incolla aiuta a familiarizzare con la sintassi del linguaggio e gli strumenti di lavoro. Copiando si commettono errori da cui è possibile imparare, il compilatore segnalerà gli errori e si imparerà a correggerli.

La compilazione

Ora che abbiamo installato il compilatore, possiamo iniziare a scrivere i nostri primi programmi.

[2] per altre distribuzioni non *debian based*, usare il package manager di sistema

Prendiamo in esame il nostro primo programma, `hello.c`

```c
#include <stdio.h>

/* Il mio primo programma in C! */
int main(void)
{
    printf("Hello, World!");
}
```

Rispettando la tradizione, il nostro primo programma stamperà "Hello, World!" a terminale. Per il momento non concentriamoci troppo sulla sintassi, vediamo piuttosto l'esecuzione del programma. Ma come possiamo farlo eseguire? Come già anticipato in precedenza, il nostro codice sorgente non è direttamente eseguibile, ma deve essere prima compilato. Vediamo come fare.

Nel venire compilato, il nostro programma passa attraverso altri tre programmi:

1. **Preprocessor**: elimina i commenti ed esegue le direttive del preprocessore (le istruzioni che iniziano con `#`).
2. **Compiler**: controlla se il codice è corretto (sintatticamente) e lo converte in linguaggio macchina, generando il codice oggetto.
3. **Linker**: combina vari file oggetto e le librerie, producendo il file eseguibile. (`a.out` oppure `a.exe`)

Esistono comandi per accedere separatamente ai vari passaggi, ma in genere viene tutto gestito da `gcc` (GNU C Compiler → GNU Compiler Collection)

```
gcc -o <nome_eseguibile> <sorgente.c> <sorgente2.c> ... <sorgenteN.c>
```

Il comando chiama il preprocessore su tutti i file, per tutti compila il file oggetto e chiama il linker che li unisce nell'unico eseguibile chiamato `<nome_eseguibile>`

> **Nota**
>
> `gcc -c <sorgente.c> <sorgente2.c> ... <sorgenteN.c>`
>
> Non esegue il linking (utile se non si ha ancora il main ad esempio)

Sintassi di C

```c
#include <stdio.h>                                          ①
/* Il mio primo programma in C! */                         ②
int main(void)                                             ③
{
  int anno = 2024;                                         ④
  printf("Hello Dati e Algoritmi %d", anno);               ⑤
}                                                          ③
```

① Direttiva: le direttive del preprocessore iniziano con `#` e vengono eseguite prima del resto del codice, in questo caso includiamo la libreria standard di input/output. Ci sono due principali usi delle direttive: includere librerie e definire macro. L'istruzione `#include` comunica al preprocessore che il programma ha bisogno delle funzioni definite in `stdio.h`, la libreria standard di input/output (infatti viene usato `printf`). L'istruzione `#define` la incontreremo più avanti, ad ogni modo è un sistema che viene spesso usato per definire costanti.

② Commento: i commenti in C89 iniziano con `/*` e terminano con `*/`. I commenti possono essere su più righe. Dal C99 si possono usare anche i commenti monoriga (`//`).

③ Funzione: la funzione `main` è il punto di partenza di ogni programma C. Il tipo di ritorno è `int`, e accetta un singolo argomento di tipo `void`. Il corpo della funzione è racchiuso tra parentesi graffe `{}`. `main` è una funzione speciale che indica il punto di partenza di ogni programma C. Il tipo di ritorno `int` indica che la funzione restituirà un valore intero (a volte il valore di ritorno del main può essere di tipo `void`, ma è una pratica non consigliata). L'argomento `void` indica che la funzione non accetta alcun argomento, si sarebbe potuto omettere, ma è buona pratica includerlo per chiarezza e attivare il controllo del compilatore.

④ Dichiarazione e assegnazione di variabile: `int anno = 2024;` dichiara una variabile di tipo `int` chiamata `anno` e le assegna il valore `2024`. Storicamente la dichiarazione di variabili in C doveva avvenire all'inizio di un blocco di codice, dal C99 non è più una regola da seguire rigidamente.

⑤ Chiamata di funzione: in questo caso stiamo usando la funzione `printf` definita in `stdio.h` a cui vengono passati 2 argomenti.

Dichiarazioni e inizializzazioni

Tutte le variabili devono essere dichiarate prima di essere usate:

```c
int altezza;
```

```c
float temperatura, peso;
```

Una delle fonti d'errore maggiori in C è che non esiste l'inizializzazione di default delle variabili, prima di utilizzarle nel programma bisogna assegnare un valore, come nel seguente esempio:

```c
int altezza;
altezze = 175;
float temperatura = 36.4f;
```

Nella prima riga viene dichiarata la variabile altezza e nella seconda le viene assegnato un valore, questa procedura può essere compattata in un'unica riga come nel caso di `temperatura`.

> **!** Importante
>
> Quando si dichiara una variabile il compilatore crea dello spazio nella memoria per contenere la variabile del tipo dichiarato (ogni tipo ha una dimensione diversa…), per essere più efficiente C non azzera la memoria che si trova in quello spazio, pertanto, se si andasse a leggere il valore di una variabile prima del primo assegnamento si troverebbero dei valori casuali.

Keyword riservate

Come detto in precedenza C è un linguaggio con poche parole chiave, quelle poche che ci sono però non possono essere usate per altri scopi, pertanto le parole sotto riportate sono da considerarsi riservate per il linguaggio e non possono essere usate per dichiarazioni di variabili o macro.

```
auto     break   case    char      const      continue   default   do
double   else    enum    extern    float      for        goto      if
inline*  int     long    register  restrict*  return     short     signed
sizeof   static  struct  switch    typedef    union      unsigned  void
volatile while   _Bool*  _Complex* _Imaginary*
```

*solo da C99

> **⚠** Il C è case sensitive
>
> Nomi di variabili valide sono:
> `int a, A; float Auto;`
> N.B. solo perché si può non vuol dire che sia una buona idea…

Input/Output

Difficilmente un programma può fare a meno di interagire con l'utente o con l'ambiente esterno. In C questo avviene tramite le funzioni `printf` e `scanf`. Queste funzioni fanno parte della libreria standard `stdio.h`, che è disponibile di default in ogni installazione di C.

Per comprenderle meglio, è necessario capire come funzionano i formati di stampa e di lettura. Un buon punto di partenza è il manuale di `printf` e `scanf` (che si può trovare digitando `man printf` e `man scanf` in un terminale).

> **Il comando `man`**
>
> Il comando `man` è un comando che permette di visualizzare il manuale di un comando o di una funzione. Il numero tra parentesi tonde dopo il nome del comando indica la sezione del manuale in cui cercare. Per esempio, `man 3 printf` cerca il manuale della funzione `printf` nella sezione 3, che contiene le funzioni di libreria.
> Le sezioni del manuale sono:
>
> 1. Programmi eseguibili e comandi della shell
> 2. Chiamate al sistema (funzioni fornite dal kernel)
> 3. Chiamate alle librerie (funzioni all'interno delle librerie di sistema)
> 4. File speciali (di solito trovabili in /dev)
> 5. Formati dei file e convenzioni p.es. /etc/passwd
> 6. Giochi
> 7. Pacchetti di macro e convenzioni p.es. man(7), groff(7).
> 8. Comandi per l'amministrazione del sistema (solitamente solo per root)
> 9. Routine del kernel [Non standard]
>
> Un buon punto di partenza per saperne di più è `man man`.

Output con printf

```
$ man 3 printf

PRINTF(3)              Linux Programmer's Manual              PRINTF(3)

NAME
       printf,  fprintf, dprintf, sprintf, snprintf, vprintf, vfprintf,
       vdprintf, vsprintf, vsnprintf - formatted output conversion

SYNOPSIS
       #include <stdio.h>

       int printf(const char *format, ...);
       int fprintf(FILE *stream, const char *format, ...);
       int dprintf(int fd, const char *format, ...);
       int sprintf(char *str, const char *format, ...);
       int snprintf(char *str, size_t size, const char *format, ...);

...
```

```
DESCRIPTION
       The functions in the printf() family produce output according to
       a format as described below.  The functions printf() and
       vprintf() write output to stdout, the standard output stream;
       fprintf() and vfprintf() write output to the given output
       stream; sprintf(), snprintf(), vsprintf(), and vsnprintf() write
       to the character string str.
```

La pagina del manuale di `printf` è veramente lunga e rischia di essere un po' dispersiva. Uno dei punti cruciali è la descrizione degli specificatori di formato, che sono i simboli che permettono di formattare l'output. Questi simboli iniziano tutti con `%`, e per ciascuno deve esserci un parametro dopo la stringa di formattazione.

Ad esempio `%d` è il simbolo per stampare un intero, `printf("%d", 10)` pertanto stampa `10`. Di seguito è riportata una lista di specificatori di formato:

Tabella 1: Formattazione stringhe

Format specifier	Value	Output
%c	'm'	m
%d	255	255
%ld	99	99
%x	255	ff
%f	25.8	25.799999
%g	25.8	25.8
%s	"hello"	hello
%8.3f	25.8	25.800
%6d	255	255
%06d	255	000255
%-6d	255	255

Il seguente codice C mette in pratica i simboli di formattazione seguendo gli esempi della tabella, si noti che, per comodità, le espressioni usate più di una volta vengono salvate in variabili:

```c
#include <stdio.h>

int main(void)
{
    char c_val = 'm';
    int i_val = 255;
    long int l_val = 99;
    float f_val = 25.8;
```

```
    printf("%c\n", c_val);
    printf("%d\n", i_val);
    printf("%ld\n", l_val);
    printf("%x\n", i_val);
    printf("%f\n", f_val);
    printf("%g\n", f_val);
    printf("%s\n", "hello");
    printf("%8.3f\n", f_val);
    printf("%6d\n", i_val);
    printf("%06d\n", i_val);
    printf("%-6d\n", i_val);
}
```

La seguente lista invece è un elenco più esaustivo di esempi di formattazione:

- `%d` per interi, es. 10
- `%f` per float, es. 3.14
- `%e` per float in notazione scientifica, es. 5.2×10^4
- `%lf` per double, es. 3.14
- `%s` per stringhe, es. `"ciao"`
- `%c` per caratteri, es. `'a'`
- `%p` per puntatori, es. `0x7fffbf7f3b4c`
- `%x` per interi in esadecimale, es. `ff`
- `%o` per interi in ottale, es. `77`
- `%Nf` per avere N "spazi" per la stampa, es. `%5d` per `112` con due spazi vuoti prima
- `%.Nf` per float con N cifre decimali, es. `%.2f` per 3.14

String format

Oltre ai simboli di formattazione, ci sono anche i caratteri speciali, che si scrivono con `\` (backslash) il carattere di *escaping*. Questi caratteri permettono di stampare a monitor dei codici che vengono interpretati in maniera speciale. Il più comune è `\n` per andare a capo, ma ce ne sono altri, di seguito una lista dei più ricorrenti:

- `\n` per andare a capo
- `\t` per tabulare
- `\b` per backspace
- `\\` per stampare il backslash
- `\"` per stampare le virgolette
- `\0` per terminare una stringa

Input con scanf

Per ricevere l'input da terminale si usa la funzione `scanf`, questa funzione esegue pattern matching sull'input per popolare delle variabili. Per esempio:

```
int base, altezza;
scanf(
    "%d%d",                                                    ①
    &base, &altezza                                            ②
    );
```

① Pattern: `%d%d` indica che si aspettano due interi
② Variabili da popolare: `base` e `altezza` sono le variabili che verranno popolate con i valori letti da terminale, si noti che il tipo deve essere coerente con il pattern. Se il pattern fosse stato `%f` le variabili avrebbero dovuto essere di tipo `float`.

> **Avviso**
>
> `scanf` scorre l'input ignorando ogni carattere bianco (spazi, tabulazioni, a capo). Se trova caratteri *compatibili* (+/-, 0-9, .) li legge e li converte nel tipo di variabile corrispondente.

Si noti che è stato usato lo strano simbolo `&` davanti alle variabili base e altezza. Questo simbolo è chiamato *operatore di indirizzamento* e restituisce l'indirizzo di memoria della variabile. Questo è necessario perché `scanf` deve scrivere direttamente nella variabile, e non può farlo se non conosce l'indirizzo di memoria. Se si dimentica il simbolo `&` si otterrà un errore a runtime.

Di fatto stiamo introducendo il concetto di puntatore, che è un argomento più avanzato e verrà trattato in seguito. Per ora è sufficiente sapere che un puntatore è un indirizzo di memoria. Quando si passa un puntatore a una funzione, si passa l'indirizzo di memoria della variabile, non la variabile stessa. Questo permette alla funzione di scrivere direttamente nella variabile, senza doverne fare una copia.

Esercizi

1. Scrivere un programma che legga due numeri e stampi la somma.
2. Scrivere un programma che legga due frazioni in formato `"n/m"` e stampi la loro somma (non semplificata).
3. Scrivere un programma che legga un numero e stampi il suo quadrato.
4. Compilare i programmi precedenti ed esplorarli usando il debugger `gdb`.

Altri esercizi si possono trovare a questo link.

Kernighan, B. W., e D. M. Ritchie. 1978. *The C Programming Language*. Prentice-Hall software series. Prentice Hall. https://books.google.it/books?id=161QAAAAMAAJ.