

Manuale C per principianti: impara le basi del linguaggio di programmazione C

Autore: Flavio Copes (inglese)

Traduttore: Angelo Mirabelli



Articolo originale: The C Beginner's Handbook: Learn C Programming Language basics in just a few hours di Flavio Copes

Tradotto e adattato da: [Angelo Mirabelli](#)

Questo manuale per principianti in C segue la regola 80/20. Imparerai l'80% del linguaggio di programmazione C nel 20% del tempo. Tale approccio ti darà una panoramica completa della lingua. Questo manuale non cerca di coprire tutti i concetti relativi al C. Si concentra sul nucleo del linguaggio, cercando di semplificare gli argomenti più complessi. E nota: [puoi ottenere una versione in inglese in PDF ed ePub di questo Manuale C per principianti qui.](#)!

Sommario

1. [Introduzione a C](#)
2. [Variabili e tipi](#)
3. [Costanti](#)
4. [Operatori](#)
5. [Condizionali](#)
6. [Cicli](#)
7. [Array](#)
8. [Stringhe](#)
9. [Puntatori](#)
10. [Funzioni](#)
11. [Input e output](#)
12. [Ambito delle variabili](#)
13. [Variabili statiche](#)
14. [Variabili globali](#)
15. [Definizioni di tipo](#)
16. [Tipi enumerati](#)
17. [Strutture](#)
18. [Parametri della riga di comando](#)
19. [File di intestazione](#)
20. [Il preprocessore](#)
21. [Conclusione](#)

Introduzione a C

C'è probabilmente il linguaggio di programmazione più conosciuto. È usato come linguaggio di riferimento per i corsi di informatica in tutto il mondo ed è probabilmente il linguaggio che le persone imparano di più a scuola insieme a Python e Java.

Ricordo che era il mio secondo linguaggio di programmazione in assoluto, dopo Pascal. C non è solo ciò che gli studenti usano per imparare a programmare. Non è un linguaggio accademico. E direi che non è il linguaggio più semplice, perché C è un linguaggio di programmazione di basso livello.

Oggi, C è ampiamente utilizzato nei dispositivi embedded e lo si trova nella maggior parte dei server Internet, che sono costruiti utilizzando Linux. Il kernel Linux è costruito usando C, e questo significa anche che C alimenta il core di tutti i dispositivi Android. Possiamo dire che il codice C lo troviamo quasi ovunque nel mondo intero. Proprio adesso. Abbastanza notevole. Quando è stato creato, C era considerato un linguaggio di alto livello, perché era portabile su tutte le macchine. Oggi diamo per scontato di poter eseguire un programma scritto su un Mac su Windows o Linux, magari usando Node.js o Python.

Una volta non era affatto così. Quello che C ha portato al tavolo era un linguaggio semplice da implementare e che aveva un compilatore che poteva essere facilmente trasferito su macchine diverse. Ho detto compilatore: C è un linguaggio di programmazione compilato, come Go, Java, Swift o Rust. Altri popolari linguaggi di programmazione come Python, Ruby o JavaScript, vengono interpretati. La differenza è consistente: un linguaggio compilato genera un file binario che può essere direttamente eseguito e distribuito.

C non ha il *garbage collection*. Ciò significa che dobbiamo gestire la memoria da soli. È un compito complesso e che richiede molta attenzione per prevenire bug, ma è anche ciò che rende C ideale per scrivere programmi per dispositivi embedded come Arduino.

C non nasconde la complessità e le capacità della macchina sottostante. Hai molto potere su di essa, una volta che sai cosa puoi fare. Voglio introdurre ora il primo programma C, che chiameremo "Hello, World!"

hello.c

```
#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}
```

Descriviamo il codice sorgente del programma: importiamo prima la libreria **stdio** (il nome sta per libreria standard input-output).

Questa libreria ci dà accesso alle funzioni di input/output.

C è un linguaggio molto piccolo al suo interno e tutto ciò che non fa parte del nucleo è fornito dalle librerie. Alcune di queste librerie sono costruite da normali programmatori e rese disponibili per l'uso da parte di altri. Alcune altre librerie sono integrate nel compilatore. Come **stdio** e altre.

stdio è la libreria che fornisce la funzione **printf()**.

Questa funzione è racchiusa in una funzione **main()**. La funzione **main()** è il punto di ingresso di qualsiasi programma C.

Ma, comunque, cos'è una funzione?

Una funzione è una routine che accetta uno o più argomenti e restituisce un singolo valore.

Nel caso di `main()`, la funzione non ottiene argomenti e restituisce un numero intero.

Indichiamo questo usando la parola chiave `void` per l'argomento e la parola chiave `int` per il valore restituito.

La funzione ha un corpo racchiuso tra parentesi graffe. All'interno del corpo abbiamo tutto il codice di cui la funzione ha bisogno per svolgere le sue operazioni.

La funzione `printf()` è scritta in modo diverso, come puoi vedere. Non ha un valore di ritorno definito e passiamo una stringa, racchiusa tra virgolette. Non abbiamo specificato il tipo di argomento.

Questo perché questa è una chiamata di funzione. Da qualche parte, all'interno della libreria `stdio`, `printf` è definito come

```
int printf(const char *format, ...);
```

Non c'è bisogno ora di capire cosa significhi, ma semplificando, questa è la definizione. E quando chiamiamo `printf("Hello, World!");`, è qui che viene eseguita la funzione.

La funzione `main()` che abbiamo definito sopra:

```
#include <stdio.h>
```

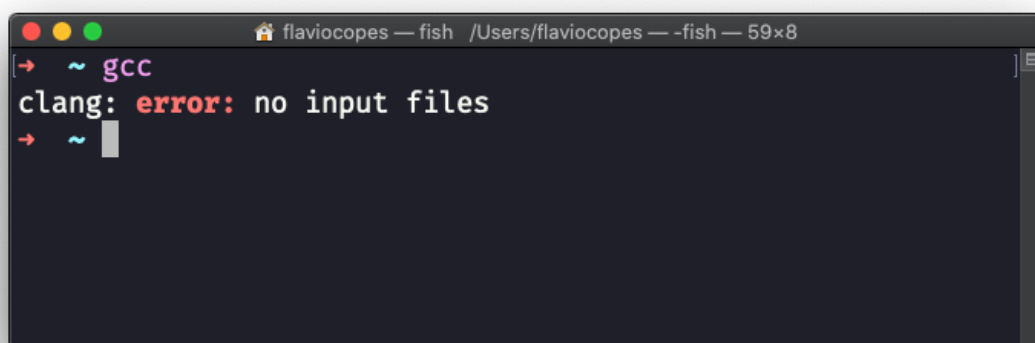
```
int main(void) {  
    printf("Hello, World!");  
}
```

verrà eseguito dal sistema operativo al momento dell'esecuzione del programma.

Come eseguiamo un programma C?

Come accennato, C è un linguaggio compilato. Per eseguire il programma dobbiamo prima compilarlo. Qualsiasi computer Linux o macOS è già dotato di un compilatore C integrato. Per Windows, puoi utilizzare il sottosistema Windows per Linux (WSL).

In ogni caso, quando apri la finestra del terminale puoi digitare `gcc`, e questo comando dovrebbe restituire un errore dicendo che non hai specificato alcun file:

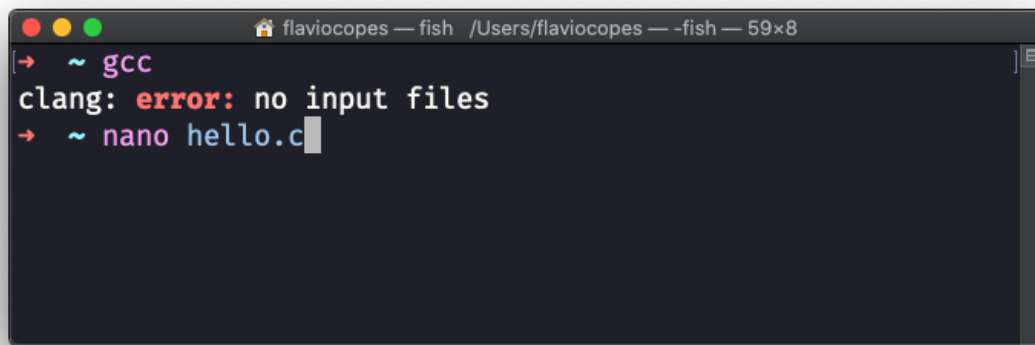


```
flaviocopes — fish /Users/flaviocopes — -fish — 59x8  
→ ~ gcc  
clang: error: no input files  
→ ~
```

Va bene. Significa che il compilatore C è lì e possiamo iniziare a usarlo.

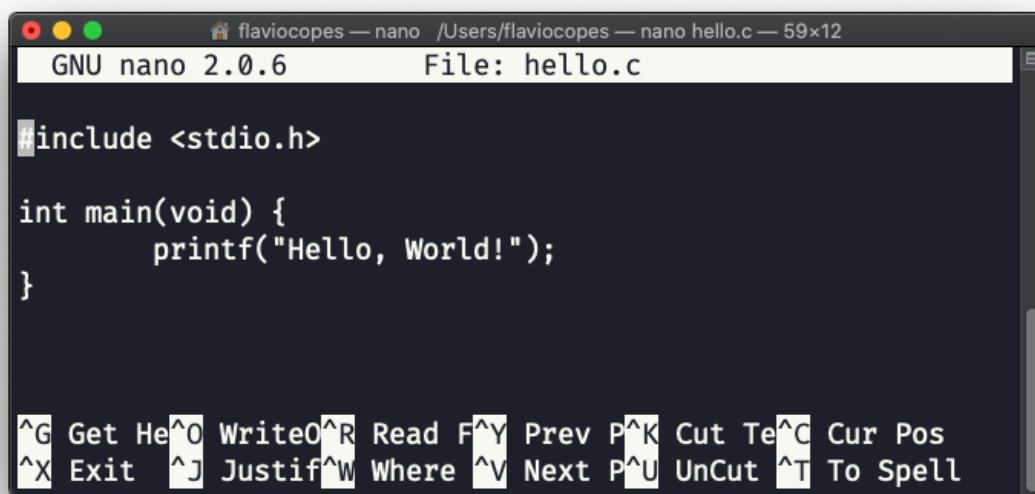
Ora scrivi le righe del programma di sopra in un file `hello.c`. Puoi usare qualsiasi editor, ma io

per semplicità userò l'editor nano dalla riga di comando:



```
flaviocopes — fish /Users/flaviocopes — -fish — 59x8
→ ~ gcc
clang: error: no input files
→ ~ nano hello.c
```

Digita il programma:



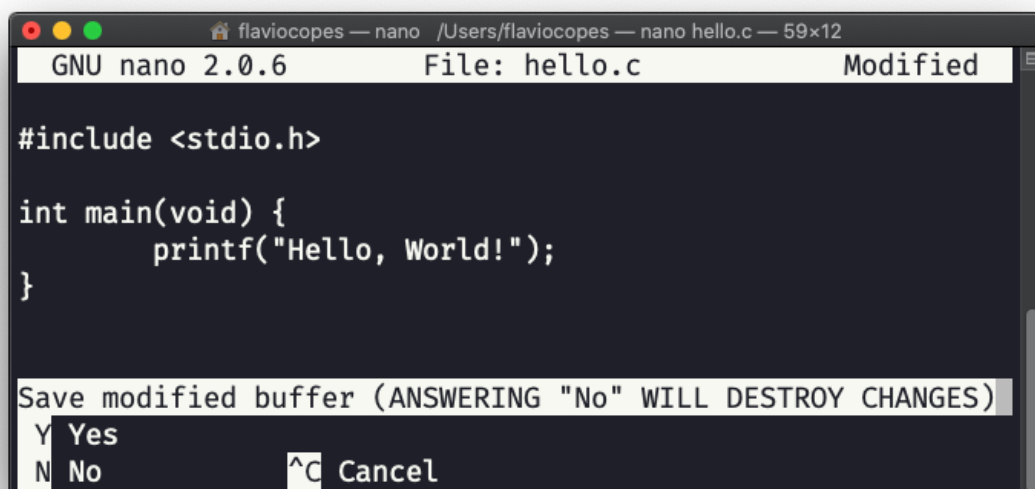
```
flaviocopes — nano /Users/flaviocopes — nano hello.c — 59x12
GNU nano 2.0.6 File: hello.c

#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}

^G Get He ^O WriteO ^R Read F ^Y Prev P ^K Cut Te ^C Cur Pos
^X Exit ^J Justif ^W Where ^V Next P ^U UnCut ^T To Spell
```

Ora premi ctrl-X per uscire:



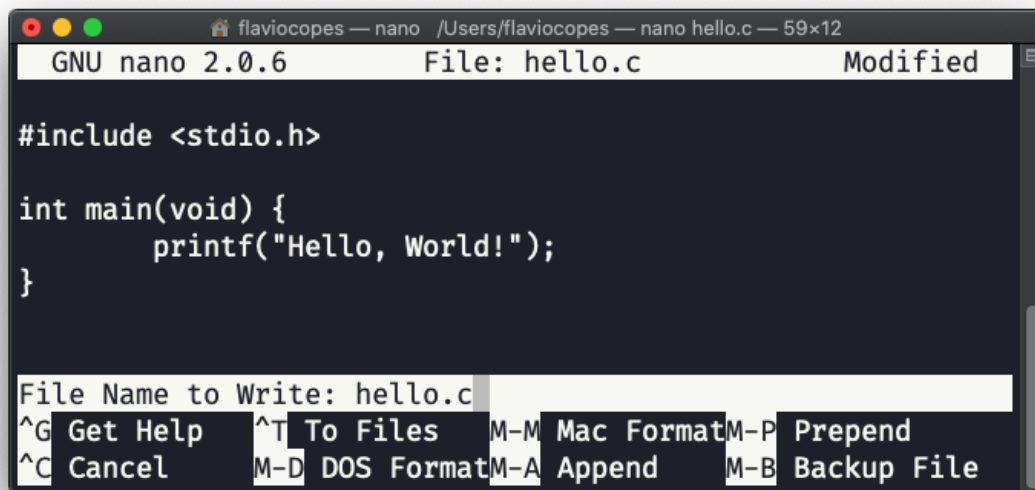
```
flaviocopes — nano /Users/flaviocopes — nano hello.c — 59x12
GNU nano 2.0.6 File: hello.c Modified

#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}

Save modified buffer (ANSWERING "No" WILL DESTROY CHANGES)
Y Yes
N No ^C Cancel
```

Conferma premendo il tasto y, quindi premi invio per confermare il nome del file:



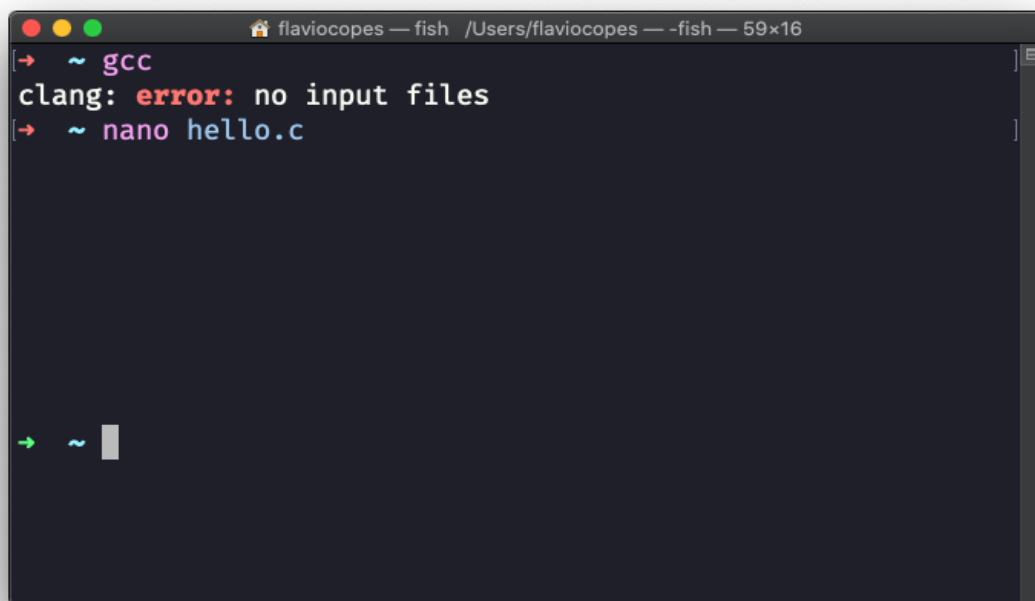
```
GNU nano 2.0.6      File: hello.c      Modified

#include <stdio.h>

int main(void) {
    printf("Hello, World!");
}

File Name to Write: hello.c
^G Get Help      ^T To Files      M-M Mac Format  M-P Prepend
^C Cancel        M-D DOS Format  M-A Append     M-B Backup File
```

Ecco fatto, ora torniamo al terminale:

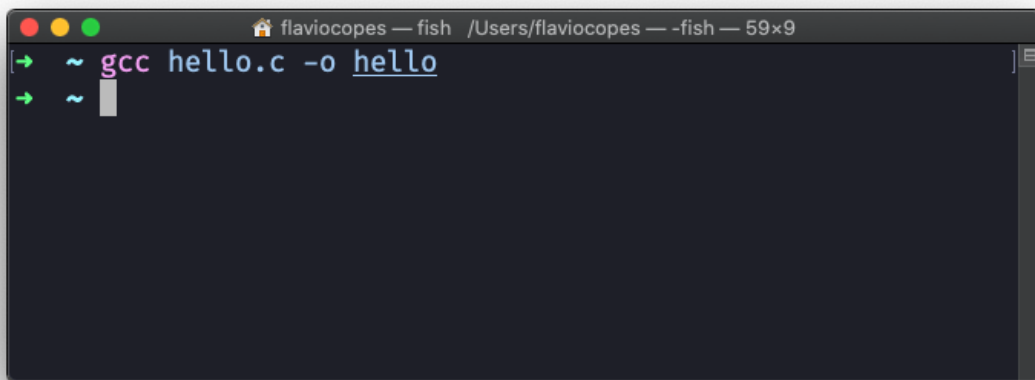


```
flaviocopes — fish /Users/flaviocopes — -fish — 59x16
[→ ~ gcc
clang: error: no input files
[→ ~ nano hello.c

[→ ~
```

Ora digita
gcc hello.c -o hello

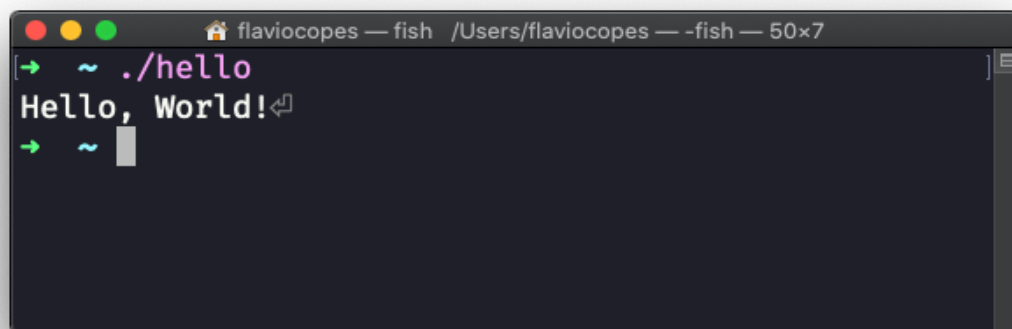
Il programma non dovrebbe darti errori:



```
flaviocopes — fish /Users/flaviocopes — -fish — 59x9
→ ~ gcc hello.c -o hello
→ ~
```

ma dovrebbe averti generato un eseguibile di nome hello. Ora digita
./hello

per eseguirlo:

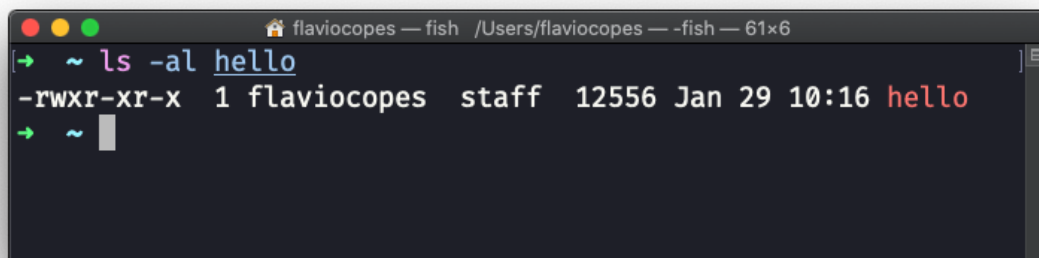


```
flaviocopes — fish /Users/flaviocopes — -fish — 50x7
→ ~ ./hello
Hello, World!
→ ~
```

Aggiungo ./ al nome del programma per dire al terminale che il comando si trova nella cartella
corrente

Fantastico!

Ora se chiami `ls -al hello`, puoi vedere che il programma ha una dimensione di soli 12 KB:



```
flaviocopes — fish /Users/flaviocopes — -fish — 61x6
→ ~ ls -al hello
-rwxr-xr-x  1 flaviocopes  staff  12556 Jan 29 10:16 hello
→ ~
```

Questo è uno dei vantaggi di C: è altamente ottimizzato e questo è anche uno dei motivi per cui
è così buono per i dispositivi embedded che hanno una quantità molto limitata di risorse.

Variabili e tipi

C è un linguaggio tipizzato staticamente.

Ciò significa che ogni variabile ha un tipo associato e questo tipo è noto al momento della compilazione.

Questo è molto diverso da come lavori con le variabili in Python, JavaScript, PHP e altri linguaggi interpretati.

Quando crei una variabile in C, devi specificare il tipo di variabile nella dichiarazione.

In questo esempio inizializziamo una variabile age di tipo `int`:

```
int age;
```

Un nome di variabile può contenere qualsiasi lettera maiuscola o minuscola, può contenere cifre e il trattino basso, ma non può iniziare con una cifra. AGE e Age10 sono nomi di variabili validi, 1age non lo è.

Puoi anche inizializzare una variabile nella dichiarazione, specificando il valore iniziale:

```
int age = 37;
```

Una volta dichiarata una variabile, puoi usarla nel codice del tuo programma. Puoi modificarne il valore in qualsiasi momento, utilizzando ad esempio l'operatore `=`, come in `age = 100`; (a condizione che il nuovo valore sia dello stesso tipo).

In questo caso:

```
#include <stdio.h>
```

```
int main(void) {  
    int age = 0;  
    age = 37.2;  
    printf("%u", age);  
}
```

il compilatore genererà un avviso in fase di compilazione e convertirà il numero decimale in un valore intero.

I tipi di dati incorporati in C sono `int`, `char`, `short`, `long`, `float`, `double`, `long double`.

Scopriamo di più su questi tipi di dato.

Numeri interi

C fornisce i seguenti tipi per definire valori interi:

- `char`
- `int`
- `short`
- `long`

Nella maggior parte dei casi, utilizzerai un `int` per memorizzare un numero intero. Ma in alcuni

casi, potresti voler scegliere una delle altre 3 opzioni.

Il tipo `char` è comunemente usato per memorizzare le lettere della tabella ASCII, ma può essere utilizzato per contenere numeri interi piccoli da -128 a 127. Occupa almeno 1 byte.

`int` occupa almeno 2 byte. `short` occupa almeno 2 byte. `long` occupa almeno 4 byte.

Come puoi vedere, non ci vengono garantiti gli stessi valori per ambienti diversi. Abbiamo solo un'indicazione. Il problema è che i numeri esatti che possono essere memorizzati in ciascun tipo di dati dipendono dall'implementazione e dall'architettura.

Ci viene garantito che `short` non è più lungo di `int`. E ci viene garantito che `long` non è più breve di `int`.

La specifica standard ANSI C determina i valori minimi di ogni tipo e grazie ad esso possiamo almeno sapere qual è il valore minimo che possiamo aspettarci di avere a nostra disposizione.

Se stai programmando C in ambiente Arduino, una scheda diversa avrà limiti diversi.

Su una scheda Arduino Uno, `int` occupa uno spazio di memoria di 2 byte, che va da -32,768 a 32,767. Su un Arduino MKR 1010, `int` occupa uno spazio di memoria di 4 byte, che va da -2,147,483,648 a 2,147,483,647. Una bella differenza.

Su tutte le schede Arduino, `short` occupa uno spazio di memoria di 2 byte, che va da -32,768 a 32,767. `long` occupa uno spazio di memoria di 4 byte, che va da -2,147,483,648 a 2,147,483,647.

Interi senza segno

Per tutti i tipi di dati di cui sopra, possiamo anteporre `unsigned` per iniziare l'intervallo da 0, invece di un numero negativo. Questo potrebbe avere senso in molti casi.

- `unsigned char` andrà da 0 ad almeno 255
- `unsigned int` andrà da 0 ad almeno 65,535
- `unsigned short` andrà da 0 ad almeno 65,535
- `unsigned long` andrà da 0 ad almeno 4,294,967,295

Il problema dell'overflow

Dati tutti questi limiti, potrebbe sorgere una domanda: come possiamo assicurarci che i nostri numeri non superino il limite? E cosa succede se superiamo il limite?

Se hai un numero `unsigned int` a 255 e lo incrementi, otterrai 256 come valore finale. Come previsto. Se hai un numero `unsigned char` a 255 e lo incrementi, otterrai 0 come valore finale. Si resetta partendo dal valore iniziale possibile.

Se hai un numero `unsigned char` a 255 e aggiungi 10, otterrai il numero 9:

```
#include <stdio.h>
```

```
int main(void) {  
    unsigned char j = 255;  
    j = j + 10;  
    printf("%u", j); /* 9 */  
}
```

Se non si dispone di un valore con segno, il comportamento non è definito. Fondamentalmente

ti darà un numero enorme che può variare, come in questo caso:

```
#include <stdio.h>

int main(void) {
    char j = 127;
    j = j + 10;
    printf("%u", j); /* 4294967177 */
}
```

In altre parole, C non ti protegge dall'andare oltre i limiti di un tipo. Devi occupartene tu stesso.

Avvisi quando si dichiara il tipo sbagliato

Quando dichiari la variabile e la iniziizzi con il valore sbagliato, il compilatore gcc (quello che probabilmente stai usando) dovrebbe avvisarti:

```
#include <stdio.h>
```

```
int main(void) {
    char j = 1000;
}
```

```
hello.c:4:11: warning: implicit conversion
    from 'int' to
      'char' changes value from 1000 to -24
    [-Wconstant-conversion]
    char j = 1000;
           ~    ^~~~
```

1 warning generated.

E ti avverte anche negli assegnamenti diretti:

```
#include <stdio.h>
```

```
int main(void) {
    char j;
    j = 1000;
}
```

Ma non se aumenti il numero usando, ad esempio, +=:

```
#include <stdio.h>
```

```
int main(void) {
    char j = 0;
    j += 1000;
}
```

Numeri in virgola mobile

I tipi a virgola mobile possono rappresentare un insieme di valori molto più ampio rispetto agli interi e possono anche rappresentare frazioni, cosa che gli interi non possono fare.

Usando i numeri in virgola mobile, rappresentiamo i numeri come cifre decimali moltiplicati per potenze di 10.

Potresti vedere numeri in virgola mobile scritti come

- 1.29e-3
- -2.3e+5

e in altri modi apparentemente strani.

I seguenti tipi:

- float
- double
- long double

sono usati per rappresentare numeri con punti decimali (tipi a virgola mobile). Tutti possono rappresentare sia numeri positivi che negativi.

I requisiti minimi per qualsiasi implementazione C sono che float può rappresentare un intervallo compreso tra 10^{-37} e 10^{+37} e viene generalmente implementato utilizzando 32 bit. double può rappresentare un insieme più grande di numeri. long double può contenere ancora più numeri.

Le cifre esatte, come per i valori interi, dipendono dall'implementazione.

Su un moderno Mac, un float è rappresentato in 32 bit e ha una precisione di 24 bit significativi. 8 bit vengono utilizzati per codificare l'esponente.

Un numero double è rappresentato in 64 bit, con una precisione di 53 bit significativi. 11 bit vengono utilizzati per codificare l'esponente.

Il tipo long double è rappresentato in 80 bit, ha una precisione di 64 bit significativi. 15 bit vengono utilizzati per codificare l'esponente.

Sul tuo computer, come puoi determinare la dimensione specifica dei tipi? Puoi scrivere un programma per farlo:

```
#include <stdio.h>

int main(void) {
    printf("char size: %lu bytes\n", sizeof(char));
    printf("int size: %lu bytes\n", sizeof(int));
    printf("short size: %lu bytes\n", sizeof(short));
    printf("long size: %lu bytes\n", sizeof(long));
    printf("float size: %lu bytes\n", sizeof(float));
    printf("double size: %lu bytes\n",
        sizeof(double));
    printf("long double size: %lu bytes\n",
        sizeof(long double));
}
```

Nel mio sistema, un moderno Mac, stampa del programma è:

```
char size: 1 bytes
int size: 4 bytes
short size: 2 bytes
```

long size: 8 bytes
float size: 4 bytes
double size: 8 bytes
long double size: 16 bytes

Costanti

Parliamo ora di costanti.

Una costante viene dichiarata in modo simile alle variabili, tranne per il fatto che è anteposta la parola chiave `const` e devi sempre specificare un valore.

Come questo:

```
const int age = 37;
```

Questo è C perfettamente valido, sebbene sia comune dichiarare costanti maiuscole, in questo modo:

```
const int AGE = 37;
```

È solo una convenzione, ma può aiutarti molto durante la lettura o la scrittura di un programma C poiché migliora la leggibilità. Nome maiuscolo significa costante, nome minuscolo significa variabile.

Il nome delle costanti segue le stesse regole per i nomi delle variabili: può contenere qualsiasi lettera maiuscola o minuscola, può contenere cifre e il trattino basso, ma non può iniziare con una cifra. `AGE` e `Age10` sono nomi di variabili validi, `1AGE` non lo è.

Un altro modo per definire le costanti è usare questa sintassi:

```
#define AGE 37
```

In questo caso, non è necessario aggiungere un tipo, non è necessario anche il segno di uguale = e si omette il punto e virgola alla fine.

Il compilatore C dedurrà il tipo dal valore specificato, in fase di compilazione.

Operatori

C ci offre un'ampia varietà di operatori che possiamo utilizzare per operare sui dati.

In particolare, possiamo identificare diversi gruppi di operatori:

- operatori aritmetici
- operatori di confronto
- operatori logici
- operatori di assegnazione composti
- operatori bit per bit
- operatori di puntatore
- operatori di struttura
- operatori vari

In questa sezione li esamineremo tutti in dettaglio, usando 2 variabili immaginarie `a` e `b` come

esempi.

Manterrò fuori da questo elenco gli operatori bit per bit, gli operatori di struttura e gli operatori di puntatore , per semplificare le cose

Operatori aritmetici

In questo macro gruppo separerò gli operatori binari e gli operatori unari.

Gli operatori binari funzionano utilizzando due operandi:

OPERATORE	NOME	ESEMPIO
=	Assegnazione	a = b
+	Addizione	a + b
-	Sottrazione	a - b
*	Moltiplicazione	a * b
/	Divisione	a / b
%	Modulo	a % b

Gli operatori unari accettano solo un operando:

OPERATORE	NOME	ESEMPIO
+	Più unario	+a
-	Meno unario	-a
++	Incremento	a++o++a

-- Decremento a--o--a

La differenza tra a++ ed ++a è che a++ incrementa la variabile a dopo averla usata. ++a incrementa la variabile a prima di usarla.

Per esempio:

```
int a = 2;
int b;
b = a++ /* b è 2, a è 3 */
b = ++a /* b è 4, a è 4 */
```

Lo stesso vale per l'operatore di decremento.

Operatori di confronto

OPERATORE	NOME	ESEMPIO
==	Operatore di uguaglianza	a == b
!=	Operatore non uguale	a != b
>	Più grande di	a > b
<	Minore di	a < b
>=	Maggiore o uguale a	a >= b
<=	Minore o uguale a	a <= b

Operatori logici

- ! NOT (esempio: !a)
- && AND (esempio: a && b)
- || OR (esempio: a || b)

Questi operatori sono ottimi quando si lavora con valori booleani.

Operatori composti di assegnazione

Questi operatori sono utili per eseguire un compito e allo stesso tempo eseguire un'operazione aritmetica:

OPERATORE	NOME	ESEMPIO
<code>+=</code>	Assegnazione somma	<code>a += b</code>
<code>-=</code>	Assegnazione differenza	<code>a -= b</code>
<code>*=</code>	Assegnazione prodotto	<code>a *= b</code>
<code>/=</code>	Assegnazione quoziente	<code>a /= b</code>
<code>%=</code>	Assegnazione modulo	<code>a %= b</code>

L'operatore ternario

L'operatore ternario è l'unico operatore in C che funziona con 3 operandi ed è un modo breve per esprimere i condizionali.

Ecco come appare:

```
<condizione> ? <espressione> : <espressione>
```

Esempio:

```
a ? b : c
```

Se `a` viene valutato `true`, l'istruzione `b` viene eseguita, altrimenti viene eseguita l'istruzione `c`. L'operatore ternario ha la stessa funzionalità di un condizionale `if/else`, tranne per il fatto che è più breve da esprimere e può essere integrato in un'espressione.

sizeof

L'operatore `sizeof` restituisce la dimensione dell'operando passato. Puoi passare una variabile o anche un tipo di dato.

Esempio di utilizzo:

```
#include <stdio.h>

int main(void) {
    int age = 37;
    printf("%ld\n", sizeof(age));
    printf("%ld", sizeof(int));
}
```

Precedenza dell'operatore

Con tutti quegli operatori (e altri, che non ho trattato in questo post, inclusi bit per bit, operatori di struttura e operatori di puntatore), dobbiamo prestare attenzione quando li

utilizziamo insieme in un'unica espressione.

Supponiamo di avere questa operazione:

```
int a = 2;
int b = 4;
int c = b + a * a / b - a;
```

Qual è il valore di c? Ci aspettiamo che l'addizione venga eseguita prima della moltiplicazione e della divisione?

C'è una serie di regole che ci aiutano a risolvere questo enigma.

In ordine dalla precedenza più bassa alla precedenza più alta, abbiamo:

- l'operatore di assegnazione =
- gli operatori **binari** + e -
- gli operatori * e /
- gli operatori unari + e -

Gli operatori hanno anche una regola di associatività, che è sempre da sinistra a destra, ad eccezione degli operatori unari e dell'assegnazione.

In:

```
int c = b + a * a / b - a;
```

Per prima cosa eseguiamo $a * a / b$, che, essendo da sinistra a destra, possiamo separare in $a * a$ e il risultato per $/ b$: $2 * 2 = 4$, $4 / 4 = 1$.

Quindi possiamo eseguire la somma e la sottrazione: $4 + 1 - 2$. Il valore di c è 3.

In ogni caso, tuttavia, voglio assicurarmi che tu sappia che puoi usare le parentesi per rendere più facile la lettura e la comprensione di qualsiasi espressione simile a quella appena vista.

Le parentesi hanno una priorità maggiore su qualsiasi altra cosa.

L'espressione precedente può essere riscritta come:

```
int c = b + ((a * a) / b) - a;
```

e non dobbiamo pensarci così tanto.

Condizionali

Qualsiasi linguaggio di programmazione offre ai programmatori la possibilità di eseguire delle scelte.

Vogliamo fare X in alcuni casi e Y in altri casi.

Vogliamo controllare i dati e fare scelte in base allo stato di tali dati.

C ci fornisce 2 modi per farlo.

La prima è l'istruzione `if`, con il suo aiutante `else`, e la seconda è l'istruzione `switch`.

if

In un'istruzione `if`, puoi verificare che una condizione sia vera, quindi eseguire il blocco fornito tra parentesi graffe:

```
int a = 1;

if (a == 1) {
```



```
    /* fa qualcosa */  
}
```

Puoi aggiungere un blocco `else` per eseguire un blocco diverso se la condizione originale risulta falsa:

```
int a = 1;  
  
if (a == 2) {  
    /* fa qualcosa */  
} else {  
    /* fa qualcos'altro */  
}
```

Fai attenzione a una fonte comune di bug: usa sempre l'operatore di confronto `==` nelle valutazioni e non l'operatore di assegnazione `=`. In caso contrario, il controllo condizionale `if` sarà sempre vero, a meno che l'argomento non sia `0`, ad esempio se fai:

```
int a = 0;  
  
if (a = 0) {  
    /* mai invocato */  
}
```

Perché accade questo? Perché il controllo condizionale cercherà un risultato booleano (il risultato di un confronto) e il numero `0` corrisponde sempre a un valore falso. Tutto il resto è vero, compresi i numeri negativi.

Puoi avere più blocchi `else` impilando insieme più istruzioni `if`:

```
int a = 1;  
  
if (a == 2) {  
    /* fa qualcosa */  
} else if (a == 1) {  
    /* */  
} else {  
    /* fa qualcos'altro ancora */  
}
```

switch

Se sei costretto a ricorrere a troppi blocchi `if / else / if` per eseguire un controllo, forse perché hai necessità di controllare il valore esatto di una variabile, allora `switch` può esserti molto utile.

Puoi fornire una variabile come condizione e una serie di punti di ingresso `case` per ogni valore che ti aspetti:

```
int a = 1;  
  
switch (a) {  
    case 0:  
        /* fa qualcosa */  
}
```

```

        break;
    case 1:
        /* fa qualcos'altro */
        break;
    case 2:
        /* fa qualcos'altro */
        break;
}

```

Abbiamo bisogno di una parola chiave `break` alla fine di ogni caso per evitare che il caso successivo venga eseguito quando quello precedente finisce. Questo effetto "a cascata" può essere utile in alcune situazioni.

Puoi aggiungere un `case` "acchiappa-tutto" alla fine, etichettato con `default`:

```

int a = 1;

switch (a) {
    case 0:
        /* fa qualcosa */
        break;
    case 1:
        /* fa qualcos'altro */
        break;
    case 2:
        /* fa qualcos'altro */
        break;
    default:
        /* gestisce tutti gli altri casi */
        break;
}

```

Cicli

C offre tre modi per eseguire un ciclo: **ciclo for**, **ciclo while** e **ciclo do while**. Tutti consentono di iterare sugli array, ma con alcune differenze. Vediamoli nel dettaglio.

Cicli For

Il primo e probabilmente il modo più comune per eseguire un ciclo è il **ciclo for**.

Usando la parola chiave `for` possiamo definire le *regole* del ciclo in anticipo, e quindi fornire il blocco che verrà eseguito ripetutamente.

Come questo:

```

for (int i = 0; i <= 10; i++) {
    /* istruzioni da ripetere */
}

```

Il blocco `(int i = 0; i <= 10; i++)` contiene i 3 dettagli di implementazione del ciclo:

- la condizione iniziale (`int i = 0`)
- il test (`i <= 10`)
- l'incremento (`i++`)

Definiamo prima una variabile di ciclo, in questo caso denominata `i`. `i` è un nome di variabile comune da utilizzare per i cicli, insieme a `j` per i cicli nidificati (un ciclo all'interno di un altro ciclo). È solo una convenzione.

La variabile viene inizializzata al valore 0 e viene eseguita la prima iterazione. Quindi viene incrementato come indicata nella parte di incremento (`i++` in questo caso, incrementando di 1) e tutto il ciclo si ripete fino ad arrivare al numero 10.

All'interno del blocco principale del ciclo possiamo accedere alla variabile `i` per sapere a quale iterazione ci troviamo. Questo programma dovrebbe stampare 0 1 2 3 4 5 5 6 7 8 9 10:

```
for (int i = 0; i <= 10; i++) {
    /* istruzioni da ripetere */
    printf("%u ", i);
}
```

I cicli possono anche iniziare da un numero alto e passare a un numero inferiore, in questo modo:

```
for (int i = 10; i > 0; i--) {
    /* istruzioni da ripetere */
}
```

Puoi anche incrementare la variabile del ciclo di 2 o un altro valore:

```
for (int i = 0; i < 1000; i = i + 30) {
    /* istruzioni da ripetere */
}
```

Cicli While

I cicli **while** sono più semplici da scrivere di un ciclo `for`, perché quest'ultimo richiede un po' più di lavoro da parte tua.

Invece di definire tutti i dati del ciclo in anticipo quando avvii il ciclo, come fai nel ciclo `for`, nel ciclo `while` devi semplicemente controllare una condizione:

```
while (i < 10) {

}
```

Ciò presuppone che `i` sia già definito e inizializzato con un valore.

E questo ciclo sarà un **ciclo infinito** a meno che tu non incrementi la variabile `i` in un punto all'interno del ciclo. Un ciclo infinito è dannoso perché bloccherà il programma, non consentendo che accada nient'altro.

Questo è ciò di cui hai bisogno per un ciclo `while` "corretto":

```
int i = 0;

while (i < 10) {
```

```

/* fa qualcosa */

    i++;
}

```

C'è un'eccezione a questo, e lo vedremo tra un minuto. Prima, lascia che ti presenti `do while`.

Cicli Do while

Anche se i cicli sono fantastici, potrebbero esserci momenti in cui devi fare una cosa in particolare: vuoi sempre eseguire un blocco e poi *forse* ripeterlo nuovamente.

Questo viene fatto usando la parola chiave `do while`. In un certo senso è molto simile a un ciclo `while`, ma leggermente diverso:

```

int i = 0;

do {
    /* fa qualcosa */

    i++;
} while (i < 10);

```

Il blocco che contiene il commento `/* fa qualcosa */` viene sempre eseguito almeno una volta, indipendentemente dal risultato del controllo della condizione in basso.

Quindi, finché `i` non è inferiore a 10, ripeteremo il blocco.

Uscire da un ciclo usando break

In C, per tutti i cicli, abbiamo un modo per uscire da un loop in qualsiasi momento, immediatamente, indipendentemente dalle condizioni impostate per il ciclo.

Ciò viene fatto usando la parola chiave `break`.

Questo è utile in molti casi. Potresti voler controllare il valore di una variabile, ad esempio:

```

for (int i = 0; i <= 10; i++) {
    if (i == 4 && unaVariabile == 10) {
        break;
    }
}

```

Avere questa opzione per uscire da un ciclo è particolarmente interessante per i cicli `while` (e anche `do while`), perché possiamo creare loop apparentemente infiniti che terminano quando si verifica una condizione. Lo definisci all'interno del blocco del ciclo:

```

int i = 0;
while (1) {
    /* fa qualcosa */

    i++;
    if (i == 10) break;
}

```

È piuttosto comune avere questo tipo di ciclo in C.

Array

Un array è una variabile che memorizza più valori.

Ogni valore nell'array, in C, deve avere lo **stesso tipo**. Ciò significa che avrai array di valori `int`, array di valori `double` e così via.

Puoi definire un array di valori `int` in questo modo:

```
int prices[5];
```

Devi sempre specificare la dimensione dell'array. C non fornisce array dinamici pronti all'uso (per ottenere questo devi usare una struttura di dati come un elenco collegato).

Puoi usare una costante per definire la dimensione:

```
const int SIZE = 5;  
int prices[SIZE];
```

Puoi inizializzare un array al momento della definizione, in questo modo:

```
int prices[5] = { 1, 2, 3, 4, 5 };
```

Ma puoi anche assegnare un valore dopo la definizione, in questo modo:

```
int prices[5];
```

```
prices[0] = 1;  
prices[1] = 2;  
prices[2] = 3;  
prices[3] = 4;  
prices[4] = 5;
```

Oppure, più pratico, usando un ciclo:

```
int prices[5];
```

```
for (int i = 0; i < 5; i++) {  
    prices[i] = i + 1;  
}
```

E puoi fare riferimento a un elemento nell'array utilizzando le parentesi quadre dopo il nome della variabile dell'array, aggiungendo un numero intero per determinare il valore dell'indice.

Come questo:

```
prices[0]; /* valore dell'elemento dell'array: 1 */  
prices[1]; /* valore dell'elemento dell'array: 2 */
```

Gli indici degli array iniziano da 0, quindi un array con 5 elementi, come l'array `prices` di sopra, avrà elementi che vanno da `prices[0]` a `prices[4]`.

La cosa interessante degli array in C è che tutti i suoi elementi sono memorizzati in sequenza, uno dopo l'altro. Qualcosa che normalmente non accade con i linguaggi di programmazione di livello superiore.

Un'altra cosa interessante è questa: il nome della variabile dell'array, nell'esempio sopra

`prices`, è un **puntatore** al primo elemento dell'array. In quanto tale può essere utilizzato come un normale puntatore.

I puntatori li tratteremo a breve.

Stringhe

In C, le stringhe sono un tipo speciale di array: una stringa è un array di valori `char`:

```
char name[7];
```

Ho introdotto il tipo `char` quando ho introdotto i tipi, ma in breve, è comunemente usato per memorizzare le lettere della tabella ASCII.

Una stringa può essere inizializzata come si inizializza un normale array:

```
char name[7] = { "F", "l", "a", "v", "i", "o" };
```

O più convenientemente con una stringa letterale (detta anche stringa costante), una sequenza di caratteri racchiusa tra virgolette:

```
char name[7] = "Flavio";
```

Puoi stampare una stringa con `printf()` usando `%s`:

```
printf("%s", name);
```

Hai notato come "Flavio" sia lungo 6 caratteri, ma ho definito un array di lunghezza 7? Come mai? Questo perché l'ultimo carattere in una stringa deve essere un valore `0`, il terminatore di stringa, e dobbiamo prevedere uno spazio per esso.

Questo è importante da tenere a mente soprattutto quando si manipolano le stringhe.

Parlando di manipolazione delle stringhe, c'è un'importante libreria standard fornita in C:

`string.h`.

Questa libreria è essenziale perché astrae molti dei dettagli di basso livello del lavoro con le stringhe e ci fornisce una serie di utili funzioni.

Puoi caricare la libreria nel tuo programma aggiungendo in alto:

```
#include <string.h>
```

E una volta fatto, hai accesso a:

- `strcpy()` per copiare una stringa su un'altra stringa
- `strcat()` per aggiungere una stringa a un'altra stringa
- `strcmp()` per confrontare due stringhe per l'uguaglianza
- `strncmp()` per confrontare i primi `n` caratteri di due stringhe
- `strlen()` per calcolare la lunghezza di una stringa

e molti, molti altri.

Puntatori

I puntatori sono una delle parti più confuse/difficili di C, secondo me. Soprattutto se sei nuovo alla programmazione, ma anche se provieni da un linguaggio di programmazione di livello superiore come Python o JavaScript.

In questa sezione voglio introdurli nel modo più semplice possibile ma non stupido.

Un puntatore è l'indirizzo di un blocco di memoria che contiene una variabile.

Quando dichiari un numero intero come questo:

```
int age = 37;
```

Possiamo usare l'operatore & per ottenere il valore dell'indirizzo di memoria della variabile:

```
printf("%p", &age); /* 0x7ffeef7dcb9c */
```

Ho usato il formato %p specificato in printf() per stampare il valore dell'indirizzo.

Possiamo assegnare l'indirizzo a una variabile:

```
int *address = &age;
```

Usando int *address nella dichiarazione, non stiamo dichiarando una variabile intera, ma piuttosto un **puntatore a un intero**.

Possiamo usare l'operatore puntatore * per ottenere il valore della variabile a cui punta un indirizzo:

```
int age = 37;
int *address = &age;
printf("%u", *address); /* 37 */
```

Questa volta stiamo usando di nuovo l'operatore puntatore, ma poiché non essendo una dichiarazione, qui sta a significare "il valore della variabile a cui punta questo puntatore".

In questo esempio dichiariamo una variabile age e utilizziamo un puntatore per inizializzare il valore:

```
int age;
int *address = &age;
*address = 37;
printf("%u", *address);
```

Quando lavori con C, scoprirai che molte cose sono basate su questo semplice concetto. Quindi assicurati di familiarizzare un po' con lui esercitandoti da solo con gli esempi sopra.

I puntatori sono una grande opportunità perché ci costringono a pensare agli indirizzi di memoria e a come sono organizzati i dati.

Gli array sono un esempio. Quando dichiari un array:

```
int prices[3] = { 5, 4, 3 };
```

La variabile prices è in realtà un puntatore al primo elemento dell'array. In questo caso puoi ottenere il valore del primo elemento usando la funzione printf():

```
printf("%u", *prices); /* 5 */
```

La cosa interessante è che possiamo ottenere il secondo elemento aggiungendo 1 al puntatore prices:

```
printf("%u", *(prices + 1)); /* 4 */
```

E così via per tutti gli altri valori.

Possiamo anche fare molte belle operazioni di manipolazione delle stringhe, poiché le stringhe sono fondamentalmente degli array.

Abbiamo anche molti altri esempi di applicazioni, incluso passare il riferimento di un oggetto o di una funzione in giro per evitare di consumare più risorse nel copiarli.

Funzioni

Le funzioni sono il modo in cui possiamo strutturare il nostro codice in subroutine potendo:

1. dargli un nome
2. chiamare quando ne abbiamo bisogno

Già nel tuo primissimo programma, in "Hello, World!", fai subito uso delle funzioni C:

```
#include <stdio.h>
```

```
int main(void) {  
    printf("Hello, World!");  
}
```

La funzione `main()` è una funzione molto importante, in quanto è il punto di ingresso per un programma C.

Ecco un'altra funzione:

```
void doSomething(int value) {  
    printf("%u", value);  
}
```

Le funzioni hanno 4 aspetti importanti:

1. hanno un nome, quindi possiamo invocarli ("chiamare") in seguito
2. specificano un valore da restituire
3. possono avere argomenti
4. hanno un corpo, compreso tra parentesi graffe

Il corpo della funzione è l'insieme di istruzioni che vengono eseguite ogni volta che invochiamo una funzione.

Se la funzione non ha un valore da restituire, è possibile utilizzare la parola chiave `void` prima del nome della funzione. Altrimenti si specifica il tipo di valore restituito dalla funzione (`int` per un numero intero, `float` per un valore in virgola mobile, `const char *` per una stringa, ecc.).

Non puoi restituire più di un valore da una funzione.

Una funzione può avere argomenti. Sono facoltativi. Se non li ha, tra parentesi inseriamo `void`, in questo modo:

```
void doSomething(void) {  
    /* ... */  
}
```

In questo caso, quando invochiamo la funzione la chiameremo senza nulla tra parentesi:

```
doSomething();
```

Se abbiamo un parametro, specifichiamo il tipo e il nome del parametro, in questo modo:


```
void doSomething(int value) {
    /* ... */
}
```

Quando invochiamo la funzione, passeremo quel parametro tra parentesi, in questo modo:
doSomething(3);

Possiamo avere più parametri, e in tal caso li separiamo usando una virgola, sia nella dichiarazione che nell'invocazione:

```
void doSomething(int value1, int value2) {
    /* ... */
}
```

```
doSomething(3, 4);
```

I parametri vengono passati per **copia**. Ciò significa che se modifichi value1, il suo valore viene modificato localmente. Il valore al di fuori della funzione, dove è stato passato nella chiamata, non cambia.

Se passi un **puntatore** come parametro, puoi modificare quel valore di variabile perché ora puoi accedervi direttamente usando il suo indirizzo di memoria.

Non è possibile definire un valore predefinito per un parametro. C++ può farlo (e così possono fare i programmi in linguaggio per Arduino), ma C non può.

Assicurati di definire la funzione prima di chiamarla, altrimenti il compilatore genererà un avviso e un errore:

```
➔ ~ gcc hello.c -o hello; ./hello
hello.c:13:3: warning: implicit declaration of
      function 'doSomething' is invalid in C99
      [-Wimplicit-function-declaration]
      doSomething(3, 4);
      ^
hello.c:17:6: error: conflicting types for
      'doSomething'
void doSomething(int value1, char value2) {
      ^
hello.c:13:3: note: previous implicit declaration
      is here
      doSomething(3, 4);
      ^
1 warning and 1 error generated.
```

L'avviso che ricevi riguarda l'ordine propedeutico della scrittura del codice, di cui ho già parlato. Invece l'errore riguarda un'altra cosa, comunque correlata all'avviso. Poiché C non "vede" la dichiarazione della funzione prima dell'invocazione, deve fare delle ipotesi. E assume che la funzione restituisca int. La funzione tuttavia restituisce void, da cui l'errore.

Se modifichi la definizione della funzione in:

```
int doSomething(int value1, int value2) {  
    printf("%d %d\n", value1, value2);  
    return 1;  
}
```

otterresti solo l'avviso e non l'errore:

→ ~ gcc hello.c -o hello; ./hello

```
hello.c:14:3: warning: implicit declaration of  
      function 'doSomething' is invalid in C99  
      [-Wimplicit-function-declaration]  
      doSomething(3, 4);  
      ^  
1 warning generated.
```

In ogni caso, assicurati di dichiarare la funzione prima di utilizzarla. Sposta la funzione in alto o aggiungi il prototipo della funzione in un file di intestazione.

All'interno di una funzione, puoi dichiarare variabili.

```
void doSomething(int value) {  
    int doubleValue = value * 2;  
}
```

Una variabile viene creata nel punto di chiamata della funzione e viene distrutta al termine della funzione. Non è visibile dall'esterno.

All'interno di una funzione, puoi chiamare la funzione stessa. Questo si chiama **ricorsione** ed è qualcosa che offre opportunità peculiari.

Input e output

C è un piccolo linguaggio e il suo "nucleo" non include alcuna funzionalità di input/output (I/O). Questo non è qualcosa di esclusivo di C, ovviamente. È comune che il core del linguaggio sia agnostico rispetto all'I/O.

Nel caso di C, Input/Output viene fornito dalla C Standard Library tramite un insieme di funzioni definite nel file di intestazione `stdio.h`.

Puoi importare questa libreria usando:

```
#include <stdio.h>
```

sopra il tuo file C.

Questa libreria ci fornisce, tra molte altre funzioni:

- `printf()`
- `scanf()`
- `sscanf()`
- `fgets()`
- `fprintf()`

Prima di descrivere cosa fanno queste funzioni, voglio dedicare un minuto a parlare dei **flussi di I/O**.

Abbiamo 3 tipi di flussi I/O in C:

- `stdin`(ingresso standard)
- `stdout`(uscita standard)
- `stderr`(errore standard)

Con le funzioni I/O lavoriamo sempre con i flussi. Uno stream è un'interfaccia di alto livello che può rappresentare un dispositivo o un file. Dal punto di vista del C, non abbiamo alcuna differenza nella lettura da un file o nella lettura da riga di comando: è comunque un flusso di I/O.

Questa è una cosa da tenere a mente.

Alcune funzioni sono progettate per funzionare con uno stream specifico, come `printf()`, che usiamo per stampare i caratteri su `stdout`. Usando la sua controparte più generica `fprintf()`, possiamo specificare su quale flusso scrivere.

Ora parliamo di `printf()`, anche se lo abbiamo incontrato più volte da quando abbiamo iniziato.

`printf()` è una delle prime funzioni che utilizzerai durante l'apprendimento della programmazione in C.

Nella sua forma di utilizzo più semplice, gli passi una stringa letterale:

```
printf("hey!");
```

e il programma stamperà il contenuto della stringa sullo schermo.

È possibile stampare il valore di una variabile. Ma è un po' più complicato perché devi aggiungere un carattere speciale, un segnaposto, che cambia a seconda del tipo di variabile. Ad esempio usiamo `%d` per un numero intero in base 10 con segno:

```
int age = 37;
```

```
printf("La mia età è %d", age);
```

Possiamo stampare più di una variabile usando le virgole:

```
int age_yesterday = 37;
```

```
int age_today = 36;
```

```
printf("Ieri la mia età era %d e oggi è %d", age_yesterday, age_today);
```

Esistono altri identificatori di formato come `%d`:

- `%c` per un carattere
- `%s` per un carattere
- `%f` per i numeri in virgola mobile
- `%p` per i puntatori

e molti altri.

Possiamo utilizzare i caratteri di escape in `printf()`, come `\n` che possiamo usare per fare in modo che l'output crei una nuova riga.

`scanf()`

`printf()` viene utilizzato come funzione di output. Voglio introdurre ora una funzione di input, così da completare il concetto di I/O: `scanf()`.

Questa funzione viene utilizzata per ottenere un valore dall'utente che esegue il programma, dalla riga di comando.

Dobbiamo prima definire una variabile che conterrà il valore che otteniamo dall'input:

```
int age;
```

Quindi chiamiamo `scanf()` con 2 argomenti: il formato (tipo) della variabile e l'indirizzo della variabile:

```
scanf("%d", &age);
```

Se vogliamo ottenere una stringa come input, ricorda che il nome della stringa è un puntatore al primo carattere, quindi non è necessario il carattere `&` prima di esso:

```
char name[20];
```

```
scanf("%s", name);
```

Ecco un piccolo programma che usa sia `printf()` che `scanf()`:

```
#include <stdio.h>
```

```
int main(void) {  
    char name[20];  
    printf("Inseriscie il tuo nome: ");  
    scanf("%s", name);  
    printf("hai inserito %s", name);  
}
```

Ambito delle variabili

Quando definisci una variabile in un programma C, a seconda di dove la dichiari, avrà un **ambito** diverso.

Ciò significa che sarà disponibile in alcuni luoghi, ma non in altri.

Il luogo in cui avviene la dichiarazione determina 2 tipi di variabili:

- **variabili globali**
- **variabili locali**

Questa è la differenza: una variabile dichiarata all'interno di una funzione è una variabile locale, come questa:

```
int main(void) {  
    int age = 37;  
}
```

Le variabili locali sono accessibili solo dall'interno della funzione e quando la funzione termina,

termina anche la loro esistenza. Vengono cancellate dalla memoria (con alcune eccezioni). Una variabile definita al di fuori di una funzione è una variabile globale, come in questo esempio:

```
int age = 37;

int main(void) {
    /* ... */
}
```

Le variabili globali sono accessibili da qualsiasi funzione del programma, e sono disponibili per l'intera esecuzione del programma, fino al suo termine.

Ho detto che le variabili locali non sono più disponibili al termine della funzione.

Il motivo è che le variabili locali sono dichiarate nello **stack**, per impostazione predefinita, a meno che non le allochi esplicitamente nell'heap usando i puntatori. Ma poi devi gestire tu stesso la memoria.

Variabili statiche

All'interno di una funzione, puoi inizializzare una **variabile statica** usando la parola chiave `static`.

Ho detto "all'interno di una funzione" perché le variabili globali sono statiche per impostazione predefinita, quindi non è necessario aggiungere la parola chiave.

Cos'è una variabile statica? Una variabile statica viene inizializzata su 0 se non viene specificato alcun valore iniziale e mantiene il valore tra le chiamate di funzione.

Considera questa funzione:

```
int incrementAge() {
    int age = 0;
    age++;
    return age;
}
```

Se chiamiamo `incrementAge()` una volta, otterremo 1 come valore restituito. Se lo chiamiamo più di una volta, otterremo sempre 1 come valore di ritorno, perché `age` è una variabile locale e viene reinizializzata a 0 ad ogni singola chiamata di funzione.

Se cambiamo la funzione in:

```
int incrementAge() {
    static int age = 0;
    age++;
    return age;
}
```

Ora ogni volta che chiamiamo questa funzione, otterremo un valore incrementato:

```
printf("%d\n", incrementAge());
printf("%d\n", incrementAge());
printf("%d\n", incrementAge());
```

ci darà

1
2
3

Possiamo anche omettere l'inizializzazione di age a 0 in `static int age = 0`; e scrivere semplicemente `static int age`; perché le variabili statiche vengono automaticamente impostate su 0 quando vengono create.

Possiamo anche avere array statici. In questo caso, ogni singolo elemento dell'array viene inizializzato a 0:

```
int incrementAge() {  
    static int ages[3];  
    ages[0]++;  
    return ages[0];  
}
```

Variabili globali

In questa sezione voglio approfondire la differenza tra **variabili globali e locali**.

Una **variabile locale** è definita all'interno di una funzione ed è disponibile solo all'interno di tale funzione.

Come questo:

```
#include <stdio.h>  
  
int main(void) {  
    char j = 0;  
    j += 10;  
    printf("%u", j); //10  
}
```

j non è disponibile da nessuna parte al di fuori della funzione main.

Una **variabile globale** è definita al di fuori di qualsiasi funzione, in questo modo:

```
#include <stdio.h>  
  
char i = 0;  
  
int main(void) {  
    i += 10;  
    printf("%u", i); //10  
}
```

È possibile accedere a una variabile globale da qualsiasi funzione del programma. L'accesso non si limita alla lettura del valore: la variabile può essere aggiornata da qualsiasi funzione.

Per questo motivo, le variabili globali sono un modo che abbiamo per condividere gli stessi dati tra le funzioni.

La principale differenza con le variabili locali è che la memoria allocata per le variabili viene

liberata una volta terminata la funzione.
Le variabili globali vengono liberate solo al termine del programma.

Definizioni di tipo

La parola chiave `typedef` in C consente di definire nuovi tipi.
Partendo dai tipi integrati in C, possiamo creare i nostri tipi, usando questa sintassi:
`typedef existingtype NEWTYPE`

Il nuovo tipo che creiamo di solito è, per convenzione, maiuscolo.
Questo per distinguerlo più facilmente, e riconoscerlo immediatamente come tipo.
Ad esempio possiamo definire un nuovo tipo `NUMBER` che è un `int`:
`typedef int NUMBER`

e una volta fatto, puoi definire nuove variabili `NUMBER`:
`NUMBER one = 1;`

Ora potresti chiederti: perché? Perché non utilizzare semplicemente il tipo integrato `int`?
Bene, `typedef` diventa davvero utile se abbinato a due cose: tipi enumerati e strutture .

Tipi enumerati

Usando le parole chiave `typedef` e `enum` possiamo definire un tipo che può avere un valore o un altro.

È uno degli usi più importanti della parola chiave `typedef`.
Questa è la sintassi di un tipo enumerato:

```
typedef enum {  
    //... valore  
} TYPENAME;
```

Il tipo enumerato che creiamo è solitamente, per convenzione, maiuscolo.
Qui c'è un semplice esempio:

```
typedef enum {  
    true,  
    false  
} BOOLEAN;
```

C viene fornito con un tipo `bool`, quindi questo esempio non è molto pratico, ma serve a farti un'idea.

Un altro esempio è definire i giorni feriali:

```
typedef enum {  
    lunedì,  
    martedì,  
    mercoledì,  
    giovedì,  
    venerdì,
```

```
    sabato,  
    domenica  
} WEEKDAY;
```

Ecco un semplice programma che utilizza questo tipo enumerato:

```
#include <stdio.h>
```

```
typedef enum {  
    lunedì,  
    martedì,  
    mercoledì,  
    giovedì,  
    venerdì,  
    sabato,  
    domenica  
} WEEKDAY;
```

```
int main(void) {  
    WEEKDAY day = lunedì;  
  
    if (day == lunedì) {  
        printf("È lunedì!");  
    } else {  
        printf("Non è lunedì");  
    }  
}
```

Ogni elemento nella definizione enum è accoppiato internamente a un numero intero. Quindi in questo esempio lunedì è 0, martedì è 1 e così via.

Ciò significa che l'istruzione condizionale sarebbe potuta essere `if (day == 0)` invece di `if (day == lunedì)`, ma è molto più semplice per noi umani ragionare con i nomi piuttosto che con i numeri, quindi è una sintassi molto conveniente.

Strutture

Usando la parola chiave `struct` possiamo creare strutture di dati complesse usando i tipi C di base.

Una struttura è una raccolta di valori di diversi tipi. Gli array in C sono limitati a un tipo, quindi le strutture possono rivelarsi molto interessanti in molti casi d'uso.

Questa è la sintassi di una struttura:

```
struct <nomestruttura> {  
    //...variabili  
};
```

Esempio:

```
struct persona {  
    int eta;
```



```
    char *nome;
};
```

Puoi dichiarare variabili che hanno come tipo quella struttura aggiungendole dopo la parentesi graffa di chiusura, prima del punto e virgola, in questo modo:

```
struct persona {
    int eta;
    char *nome;
} flavio;
```

O più variabili, come in questo caso:

```
struct persona {
    int eta;
    char *nome;
} flavio, popolazione[20];
```

In questo caso dichiaro una singola variabile per sona denominata `flavio` e un array di 20 persona denominata `popolazione`.

Possiamo anche dichiarare le variabili in un secondo momento, usando questa sintassi:

```
struct persona {
    int eta;
    char *nome;
};
```

```
struct persona flavio;
```

Possiamo inizializzare una struttura al momento della dichiarazione:

```
struct persona {
    int eta;
    char *nome;
};
```

```
struct persona flavio = { 37, "Flavio" };
```

e una volta definita una struttura, possiamo accedere ai valori in essa contenuti usando un punto:

```
struct persona {
    int eta;
    char *nome;
};
```

```
struct person flavio = { 37, "Flavio" };
printf("%s, eta %u", flavio.nome, flavio.eta);
```

Possiamo anche cambiare i valori usando la sintassi col punto:

```
struct persona {
```

```

    int eta;
    char *nome;
};

struct persona flavio = { 37, "Flavio" };

flavio.eta = 38;

```

Le strutture sono molto utili perché possiamo passarle come parametri di funzione, o restituire valori, incorporando varie variabili al loro interno. Ogni variabile ha un'etichetta.

È importante notare che le strutture vengono **passate tramite copia**, a meno che ovviamente non si passi un puntatore a una struttura, nel qual caso viene passato per riferimento.

Utilizzando typedef possiamo semplificare il codice quando si lavora con le strutture.

Diamo un'occhiata a un esempio:

```

typedef struct {
    int eta;
    char *nome;
} PERSONA;

```

La struttura che creiamo utilizzando typedef è solitamente, per convenzione, maiuscola.

Ora possiamo dichiarare nuove variabili PERSONA come questa:

```
PERSONA flavio;
```

e possiamo inizializzarle nella dichiarazione in questo modo:

```
PERSONA flavio = { 37, "Flavio" };
```

Parametri della riga di comando

Nei tuoi programmi C, potresti aver la necessità di dover accettare i parametri dalla riga di comando al lancio del comando stesso.

Per casi semplici, tutto ciò che devi fare per ottenere ciò è cambiare l'aspetto della funzione main() da

```
int main(void)
```

a

```
int main (int argc, char *argv[])
```

argc è un numero intero che contiene il numero di parametri forniti nella riga di comando.

argv è un array di stringhe.

All'avvio del programma, ci vengono forniti gli argomenti in quei 2 parametri.

Nota che c'è sempre almeno un elemento nell'array argv: il nome del programma

Prendiamo l'esempio del compilatore C che utilizziamo per eseguire i nostri programmi, in questo modo:

```
gcc hello.c -o hello
```

Se questo fosse il nostro programma, argc avrà il valore 4 e argv sarà un array contenente

- gcc
- hello.c
- -o
- hello

Scriviamo un programma che stampi gli argomenti che riceve:

```
#include <stdio.h>
```

```
int main (int argc, char *argv[]) {  
    for (int i = 0; i < argc; i++) {  
        printf("%s\n", argv[i]);  
    }  
}
```

Se il nome del nostro programma è hello e lo eseguiamo in questo modo: ./hello, otterremmo questo come output:

```
./hello
```

Se passiamo alcuni parametri casuali, come questo: ./hello a b c otterremmo questo output sul terminale:

```
./hello  
a  
b  
c
```

Questo sistema funziona alla grande per esigenze semplici. Per esigenze più complesse, ci sono pacchetti comunemente usati come **getopt**.

File di intestazione

Programmi semplici possono essere inseriti in un unico file. Ma quando il tuo programma diventa più grande è impossibile tenerlo tutto in un solo file.

È possibile spostare parti di un programma in un file separato. Quindi crei un **file di intestazione**.

Un file di intestazione sembra un normale file C, tranne per il fatto che termina con .h invece di .c. Invece delle implementazioni delle tue funzioni e delle altre parti di un programma, contiene le **dichiarazioni**.

Hai già utilizzato i file di intestazione quando hai usato per la prima volta la funzione printf() o un'altra funzione di I/O, e per fare ciò hai dovuto digitare:

```
#include <stdio.h>
```

per usarlo.

#include è una direttiva del preprocessore.

Il preprocessore va a cercare il file stdio.h nella libreria standard perché hai usato le parentesi attorno ad esso. Per includere i tuoi file di intestazione, utilizzerai le virgolette, in

questo modo:

```
#include "myfile.h"
```

Quanto sopra cercherà `myfile.h` nella cartella corrente.

Puoi anche utilizzare una struttura di cartelle per le librerie:

```
#include "myfolder/myfile.h"
```

Diamo un'occhiata a un esempio. Questo programma calcola gli anni da un determinato anno:

```
#include <stdio.h>
```

```
int calculateAge(int year) {  
    const int CURRENT_YEAR = 2020;  
    return CURRENT_YEAR - year;  
}  
  
int main(void) {  
    printf("%u", calculateAge(1983));  
}
```

Supponiamo di voler spostare la funzione `calculateAge` in un file separato.

creo un file `calculate_age.c`:

```
int calculateAge(int year) {  
    const int CURRENT_YEAR = 2020;  
    return CURRENT_YEAR - year;  
}
```

E un file `calculate_age.h` in cui inserisco la *funzione prototipo*, che è la stessa della funzione nel file `.c`, tranne il corpo:

```
int calculateAge(int year);
```

Ora nel file `.c` principale possiamo andare a rimuovere la definizione della funzione `calculateAge()` e possiamo importare `calculate_age.h`, che renderà disponibile la funzione `calculateAge()`:

```
#include <stdio.h>  
#include "calculate_age.h"  
  
int main(void) {  
    printf("%u", calculateAge(1983));  
}
```

Non dimenticare che per compilare un programma composto da più file, devi elencarli tutti nella riga di comando, in questo modo:

```
gcc -o main main.c calculate_age.c
```

E con configurazioni più complesse, è necessario un Makefile per dire al compilatore come compilare il programma.

Il preprocessore

Il preprocessore è uno strumento che ci aiuta molto durante la programmazione in C. Fa parte dello standard C, proprio come il linguaggio, il compilatore e la libreria standard.

Analizza il nostro programma e si assicura che il compilatore ottenga tutto ciò di cui ha bisogno prima di procedere con il processo.

Cosa fa in pratica?

Ad esempio, cerca tutti i file di intestazione che includi con la direttiva `#include`.

Esamina anche ogni costante che hai definito utilizzando `#define` e la sostituisce con il suo valore effettivo.

Questo è solo l'inizio. Ho citato queste 2 operazioni perché sono le più comuni. Il preprocessore può fare molto di più.

Hai notato che `#include` e `#define` hanno un `#` all'inizio? Questo è comune a tutte le direttive del preprocessore. Se una riga inizia con `#`, viene curata dal preprocessore.

Condizionali

Una delle cose che possiamo fare è usare i condizionali per cambiare il modo in cui il nostro programma verrà compilato, a seconda del valore di un'espressione.

Ad esempio possiamo verificare se la costante `DEBUG` è 0:

```
#include <stdio.h>

const int DEBUG = 0;

int main(void) {
    #if DEBUG == 0
        printf("NON sto eseguendo il debug\n");
    #else
        printf("Sto eseguendo il debug\n");
    #endif
}
```

Costanti simboliche

Possiamo definire una **costante simbolica** :

```
#define VALUE 1
#define PI 3.14
#define NAME "Flavio"
```

Quando utilizziamo `NAME` o `PI` o `VALUE` nel nostro programma, il preprocessore sostituisce il suo nome con il valore prima di eseguire il programma.

Le costanti simboliche sono molto utili perché possiamo dare nomi ai valori senza creare variabili in fase di compilazione.

Macro

Con `#define` possiamo anche definire una **macro**. La differenza tra una macro e una costante simbolica è che una macro può accettare un argomento e in genere contiene codice, mentre una costante simbolica è un valore:

```
#define POWER(x) ((x) * (x))
```

Notare le parentesi attorno agli argomenti: questa è una buona pratica per evitare problemi quando la macro viene sostituita nel processo di precompilazione.

Quindi possiamo usarlo nel nostro codice in questo modo:

```
printf("%u\n", POWER(4)); //16
```

La grande differenza con le funzioni è che le macro non specificano il tipo dei loro argomenti o valori restituiti, il che potrebbe essere utile in alcuni casi.

Le macro, tuttavia, sono limitate a definizioni di una sola riga.

ifdef

Possiamo verificare se una costante simbolica o una macro è definita usando `#ifdef`:

```
#include <stdio.h>
#define VALUE 1

int main(void) {
#ifdef VALUE
    printf("Il valore è definito\n");
#else
    printf("Il valore non è definito\n");
#endif
}
```

Abbiamo a disposizione anche `#ifndef` per verificare il contrario (macro non definita).

Possiamo anche usare `#if defined` e `#if !defined` per fare lo stesso compito.

È comune avvolgere un blocco di codice in un blocco come questo:

```
#if 0
```

```
#endif
```

per impedirne temporaneamente l'esecuzione o per utilizzare una costante simbolica `DEBUG`:

```
#define DEBUG 0
```

```
#if DEBUG
    //codice inviato solo al compilatore
    //se DEBUG non è 0
#endif
```

Costanti simboliche predefinite che puoi usare

Il preprocessore definisce anche un numero di costanti simboliche che puoi usare, identificate dai 2 trattini bassi prima e dopo il nome, tra cui:

- `__LINE__` si traduce nella riga corrente nel file del codice sorgente
- `__FILE__` si traduce nel nome del file
- `__DATE__` si traduce nella data di compilazione, nel formato Mmm gg aaaa
- `__TIME__` si traduce nel tempo di compilazione, nel formato hh:mm:ss

Conclusion

Grazie mille per aver letto questo manuale! Spero che ti ispiri a saperne di più su C.

Per ulteriori tutorial in inglese, dai un'occhiata al mio blog flaviocopes.com.

Invia qualsiasi feedback, segnalazione di errore o opinione a hey@flaviocopes.com

E ricorda: [puoi ottenere una versione in inglese in PDF ed ePub di questo Manuale per principianti C](#)

Potete contattarmi su Twitter [@flaviocopes](https://twitter.com/flaviocopes).

Articoli più cliccati

Unire CSV con Python
Il comando Git push
Centrare immagini in CSS
I codici Alt
Tenere a bada il footer
Cosa è API?
Chiave licenza Windows 10
Live Server non funziona
Formattare in Markdown
Guida allo stile di fCC
Valori unici Python
Array.find() in JS
position in CSS
Destrutturazione in JS
length in JS
Boilerplate HTML5
Manuale JS
Git e GitHub
about:blank
LS in Linux
Immagine di sfondo REACT
Vim su Windows in PowerShell
Processi Child di Node.js
Algoritmi di forza bruta
@property in Python
Algoritmo di Dijkstra
Come annullare un Git Add
Ripristinare schede su Chrome
Link nuova scheda in HTML
Eliminare un branch di Git