

Puntatori

Matteo Spanio

9 aprile 2024

In questo capitolo vengono introdotti i puntatori, un concetto fondamentale per la programmazione in C. Si descrive cosa siano, la loro sintassi e i principali utilizzi: l'indirizzamento di variabili, l'accesso a variabili tramite puntatore, cosa abbiamo in comune con gli array e le stringhe e come vengono passati ai parametri di funzione.

Il termine *puntatore* produce sempre un po' di confusione tra i principianti del C, in realtà, una volta capito il concetto, si tratta di una feature molto potente e utile del linguaggio. Non sono effettivamente complicati, ma è facile fare confusione, spero che questa guida possa aiutare a chiarire le idee...

Quando si dichiara una variabile si comunica al compilatore il suo tipo e il suo nome prima del suo utilizzo, questo è fondamentale perché la dichiarazione permette al compilatore di allocare un blocco di memoria per immagazzinare la variabile. Quindi:

Ad ogni variabile (e non solo) è associato un indirizzo di memoria. Questo indirizzo è un numero che identifica la posizione della variabile in memoria.

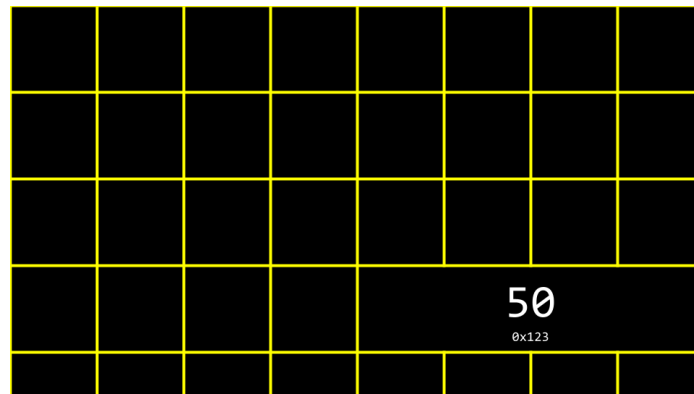


Figura 1: Rappresentazione in memoria di una variabile.

Cos'è un puntatore?

Un **puntatore** è una variabile che contiene l'informazione per accedere ad un'altra variabile, ossia il suo **indirizzo**.

Quindi un puntatore è semplicemente un indirizzo di memoria contenente una variabile.

Se si dichiara una variabile e un puntatore ad essa si può accedere alla variabile in due modi:

1. Utilizzando la variabile stessa
2. Utilizzando il puntatore

Un esempio potrebbe essere:

```
#include <stdio.h>

int main(void)
{
    int a;
    int *ptr_to_a;                                ①

    ptr_to_a = &a;                                ②

    a = 5;
    printf("The value of a is %d\n", a);

    *ptr_to_a = 6;                                ③
    printf("The value of a is %d\n", a);
    printf("The value of ptr_to_a is %p\n", ptr_to_a);
    printf("It stores the value %d\n", *ptr_to_a);
    printf("The address of a is %p\n", &a);
}
```

- ① Indica la dichiarazione di un puntatore a un intero, il carattere `*` è usato per indicare che `ptr_to_a` è un puntatore, il tipo di variabile a cui punta è indicato prima del `*`.
- ② In C il simbolo `&` prima di un nome di variabile è usato per ottenere l'indirizzo di memoria di quella variabile e, come detto prima, un puntatore è un indirizzo di memoria. Quindi `ptr_to_a` contiene l'indirizzo di `a`.
- ③ Dopo che ad `a` è stato assegnato il valore 5, viene usato ancora il simbolo `*`, questa volta però ha un significato diverso: una volta che il puntatore è dichiarato, il simbolo `*` è usato per accedere al valore della variabile a cui il puntatore punta. Quindi `*ptr_to_a` è il valore di `a`. È un modo alternativo per impostare il valore di `a` a 6.

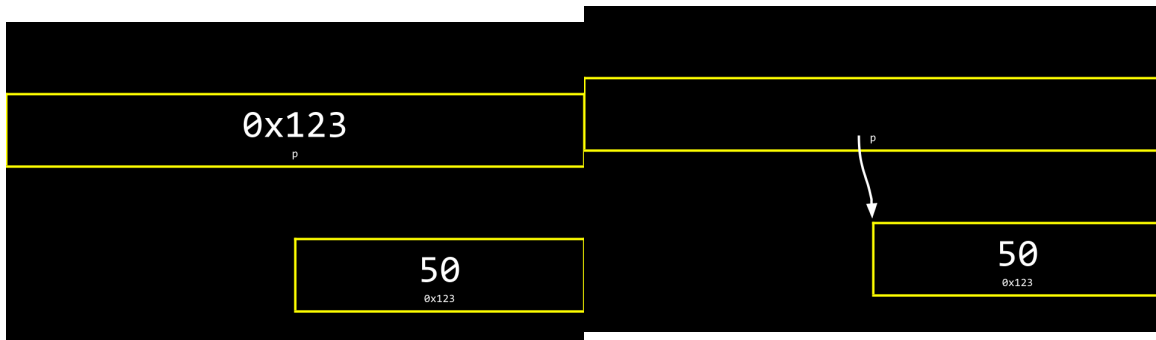
💡 Consiglio

Un sistema un po' stravagante ma efficace per imparare il significato degli operatori `*` e `&` nel contesto dei puntatori è pensare a `*` come “*valore puntato da*” e `&` come “*indirizzo di*”. Una volta imparato questo, il resto è molto più semplice.

Si prenda in considerazione il seguente listato:

```
#include <stdio.h>
int main(void)
{
    int n = 50;
    int *p = &n;
    printf("%d\n", p); // stampa l'indirizzo di n
    printf("%d\n", *p); // stampa il valore di n
}
```

Come prima, viene dichiarata la variabile `n` e subito dopo si dichiara un puntatore `p` che punta a `n`. L'immagine Figura 2a aiuta a visualizzare la situazione in memoria, mentre l'immagine Figura 2b spiega graficamente il concetto di un puntatore che *punta a* una variabile, spesso si indica graficamente con una freccia che parte dal puntatore e punta alla variabile.



(a) Rappresentazione della memoria per il listato. (b) Rappresentazione di puntatore che *punta a* una variabile.

Figura 2: Visualizzazione dello stato della memoria con i puntatori.

🔥 Attenzione

Applicare l'operatore `*` ad un puntatore non inizializzato è un errore:

```
int *p;
printf("%d\n", *p); // errore!
```

Assegnare un valore a un puntatore non inizializzato è un errore:

```
int *p; // p non è inizializzato
*p = 5; // errore!
```

Dichiarazione di un puntatore

I puntatori si possono dichiarare insieme ad altre variabili e possono essere di qualsiasi tipo:

```
int *p, x, *y, a[10];
float *f;
char *c;
```

In questo caso `p` e `y` sono puntatori a interi, `x` è un intero e `a` è un array di 10 interi. Si noti che il carattere `*`, seppure faccia parte del tipo della variabile, è posto vicino al nome della variabile, questo per evitare di creare confusione proprio in una dichiarazione multipla come nella riga di codice sopra. Lo stesso codice si sarebbe potuto scrivere:

```
int* p, x, *y, a[10];
float* f;
char* c;
```

Il secondo esempio crea maggiore confusione alla maggior parte delle persone, quindi è meglio evitare di dichiarare variabili in questo modo.

Esempio di utilizzo

Spesso un esempio chiarisce meglio il concetto, di seguito si introducono due esempi per chiarire il concetto di puntatore, e la sintassi

```
int i, j, *p;           ①
i = 5;                  ②
p = &i;                 ③
j = *p;                 ④
printf("%d\n", j); // stampa 5  ⑤
```

- ① Si dichiarano tre variabili: `i`, `j` e `p`, `p` è un puntatore a un intero.
- ② Si assegna il valore 5 a `i`.
- ③ Si assegna l'indirizzo di `i` a `p`, l'operatore `&` è usato per ottenere l'indirizzo di `i`.
- ④ Si assegna il valore puntato da `p` a `j`, l'operatore `*` è usato per accedere al valore puntato da `p`.
- ⑤ Si stampa il valore di `j`, che è 5.

```
// esempio 2
int *p, *q, i;
p = &i;
*p = 6;
q = p;
(*q)++;
printf("%d\n", i); // stampa 7
```

Il secondo esempio è un po' più complesso, ma il concetto è lo stesso. Si dichiarano due puntatori `p` e `q` e una variabile `i`. Si assegna l'indirizzo di `i` a `p`, si assegna il valore 6 a `i` tramite `*p`, si assegna `p` a `q` e si incrementa il valore puntato da `q` di 1. L'espressione `(*q)++` è equivalente a `*q = *q + 1`, servono le parentesi perché l'operatore `++` ha una precedenza maggiore rispetto all'operatore `*`.

Puntatori come parametri di funzioni

In C non è possibile restituire più di un valore da una funzione.

```
void incrementa(int a, int b) {
    a++;
    b++;
    // alla fine della funzione i valori di a e b vengono persi
}
```

I puntatori permettono di aggirare questa limitazione. Possono essere passati come argomenti di funzioni:

```
void incrementa(int *a, int *b) {
    *a = *a + 1;
    *b = *b + 1;
}
```

Ci si basa unicamente sui side effects, è possibile quindi restituire più di un valore.

La chiamata di funzione avviene così:

```
int i = 1;
int j = 10;
incrementa(&i, &j);
printf("%d\n", i); // stampa 2
printf("%d\n", j); // stampa 11
```

Abbiamo già visto questa sintassi con `scanf`.

Esempio di utilizzo di puntatori con funzioni

Scrivere una funzione che scambi il valore di due variabili.

Il prototipo della funzione è: `void swap(int *a, int *b);`

Verrebbe spontaneo risolvere il problema senza usare i puntatori:

```
void swap(int a, int b) {  
    int temp = a;           ①  
    a = b;                  ②  
    b = temp;               ③  
    printf("a = %d, b = %d\n", a, b);  
}
```

- ① Si salva il valore di `a` in una variabile temporanea.
- ② Si assegna il valore di `b` a `a`.
- ③ Si assegna il valore salvato in precedenza a `b`.

All'ultima riga il codice effettivamente ha scambiato i valori di `a` e `b`, ma si tratta di variabili locali alla funzione, quindi i valori di `a` e `b` all'esterno della funzione non sono stati modificati. Per rendere la modifica permanente si devono usare i puntatori:

```
void swap(int *a, int *b) {  
    int temp = *a;  
    *a = *b;  
    *b = temp;  
    printf("a = %d, b = %d\n", *a, *b);  
}
```

La soluzione è identica alla precedente, ma questa volta i valori di `a` e `b` vengono modificati direttamente, ciò vuol dire che i valori di `a` e `b` all'esterno della funzione saranno modificati.

Puntatori e array

In realtà, in C, un array è un puntatore al primo elemento dell'array.

```
int a[10];
```

`a` è un puntatore al primo elemento dell'array, quindi è possibile scrivere `*a` per accedere al primo elemento dell'array.

L'operatore `[]` permette di scorrere la memoria a partire dal primo elemento dell'array.

Gli array possono quindi essere trattati come puntatori:

```
int a[10], *p;  
p = a;
```

p punta al primo elemento dell'array.

Attenzione

In generale, in C, si possono trattare gli array come puntatori, ma non si può trattare un puntatore come un array (o almeno non sempre). Gli array sono costanti, mentre i puntatori no.

Gli elementi dell'array si trovano in posizioni di memoria contigue. Nella pratica, si usa `a[0]`, ma è solo uno zucchero sintattico per `*(a + 0)`, in cui 0 è l'offset rispetto al primo elemento dell'array. `*(a + 1)` è il secondo elemento dell'array, `*(a + 2)` è il terzo elemento e così via.

Quindi l'operazione `p + 1` punta al secondo elemento dell'array.

```
int a[] = {1, 2, 3, 4, 5};  
int *p = a;  
  
p + 1; // è l'indirizzo del secondo elemento dell'array  
p + 2; // è l'indirizzo del terzo elemento dell'array  
// per accedere al valore si usa *p  
*(p + 1); // è il valore del secondo elemento dell'array  
p[1]; // è il valore del secondo elemento dell'array
```

Nota

C capisce da solo che p è un puntatore ad un array di interi, quindi `p + 1` punta al secondo elemento dell'array che si trova dopo un salto di 4 byte (la dimensione di un intero).

```
int a[10], *p;  
p = a;  
for (int i = 0; i < 10; i++) {  
    *(p + i) = i;  
    // oppure  
    // p[i] = i;  
}
```

La sintassi `p[i]` è uno zucchero sintattico per `*(p + i)`.

Esercizio

Scrivere una funzione che trovi il minimo e il massimo in un array di interi.

Il prototipo della funzione è: `void minmax(const int *a, int n, int *min, int *max);`

Puntatori di puntatori

Un puntatore può puntare ad un altro puntatore.

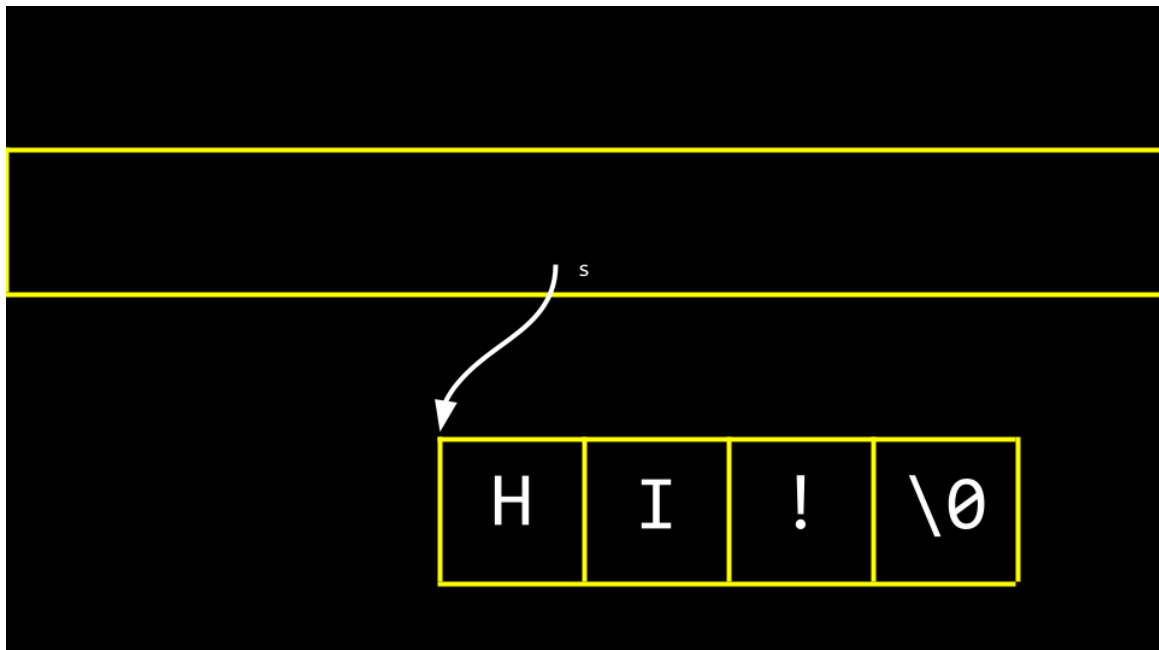
```
int i = 5;  
int *p = &i;  
int **q = &p;
```

Può sembrare strano, ma è una pratica molto comune.

Le stringhe in C sono array di caratteri terminati da un carattere nullo.

```
char s[] = "Hi!";  
// char *s = "Hi!";
```

`s` è un puntatore al primo carattere della stringa.



Il caso più comune di puntatore a puntatore è quello delle stringhe.

Quando si deve memorizzare un array di stringhe si usa un array di puntatori a caratteri.


```
int main(int argc, char **argv) {  
    // argv è un array di puntatori a caratteri  
}  
// oppure  
int main(int argc, char *argv[]) {  
    // argv è un array di puntatori a caratteri  
}
```