

3. Puntatori

Sommario

- Memoria, indirizzamento e puntatori
- Tipo `void *` e conversione di puntatori

3.1. Memoria, indirizzi e puntatori

La memoria di una macchina è organizzata in celle contigue, tipicamente da 1 byte, ciascuna con un proprio indirizzo ¹.

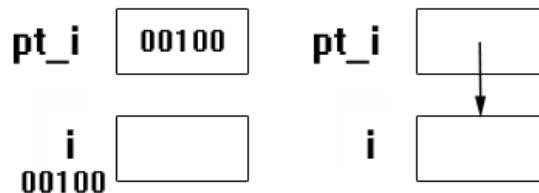
indirizzo	contenuto
0xF0000000	00100010
0xF0000001	00010000
0xF0000002	11111110
0xF0000003	11011010
...	...

Una variabile identifica un certo numero di celle contigue, dipendente dal tipo di dato che memorizza (ad es., 1 cella per `char`, 4 celle per `int`). Accedendo ad una variabile, accediamo al contenuto delle celle identificate dalla variabile. In C il programmatore ha la possibilità di gestire gli indirizzi attraverso delle variabili che vengono definite di tipo *puntatore*.

I valori delle variabili di tipo puntatore sono *indirizzi di memoria*, cioè valori numerici che fanno riferimento a specifiche locazioni di memoria.

¹ Gli indirizzi di una macchina a 32 bit vengono rappresentati per brevità con 8 cifre esadecimali, ciascuna rappresentativa del gruppo di 4 bit corrispondente alla sua rappresentazione in sistema binario. Il prefisso `0x` indica che l'indirizzo riportato è rappresentato, appunto, nel sistema esadecimale.

Normalmente non interessa conoscere lo specifico indirizzo contenuto in una variabile di tipo puntatore, mentre è molto importante conoscere a quali variabili il puntatore fa riferimento ed il valore in essa contenuto. È quindi sufficiente rappresentare graficamente l'indirizzamento usando una freccia.



3.1.1. Operatore *indirizzo-di*

Vediamo innanzitutto come ottenere dei valori di tipo puntatore, cioè degli indirizzi.

```
int i = 15;
printf("L'indirizzo di i e' %p\n", &i);
printf("mentre il valore di i e' %d\n", i);
```

stampa

```
L'indirizzo di i e' 0xbffff47c
mentre il valore di i e' 1
```

L'operatore `&` si chiama operatore *indirizzo-di* e restituisce l'indirizzo della prima cella del gruppo identificato dalla variabile a cui viene applicato.

NOTA: Per la specifica di formato dei puntatori si usa il carattere `p`, indipendentemente dal tipo di dato a cui il puntatore fa riferimento.

3.1.2. Operatore di *dereferenziazione* (indirizzamento indiretto)

L'operatore `*` permette di accedere al valore contenuto nella locazione identificata da un certo indirizzo.

Esempio 3.1. Si consideri il seguente frammento di codice

```
int j = 1;
int i = *&j;
```

L'espressione `&j` restituisce l'indirizzo della locazione puntata da `j`. L'operatore `*` restituisce, invece, il valore contenuto nella locazione di memoria puntata dal suo argomento (cioè il risultato dell'espressione `&j`). Pertanto, l'istruzione `int i = *&j;` equivale all'istruzione `i = j;`.

NOTA: l'operatore (unario) di indirizzamento indiretto `*` non deve essere confuso con l'operatore di moltiplicazione (binario).

3.1.3. Variabili di tipo puntatore

Per memorizzare gli indirizzi occorre dichiarare delle variabili di tipo *puntatore*. Nella dichiarazione è necessario specificare che tipo di dato è contenuto nella locazione puntata.

Esempio 3.2.

L'istruzione

```
int *p1;
```

Dichiara `p1` come variabile di tipo puntatore, specificando che il tipo contenuto nella locazione puntata è intero.

NOTA: La dichiarazione di una variabile puntatore non “crea” la variabile puntata.

Esempio 3.3.

```
int *p1;  
*p1 = 10; // ERRORE: l'indirizzo contenuto in p1  
          // non corrisponde a memoria allocata per un intero
```

Per accedere ad una locazione di memoria, è necessario che essa sia allocata. In caso contrario viene generato un errore a tempo di esecuzione.

Come visto, una dichiarazione di variabile alloca memoria.

Esempio 3.4.

```
int i; // Alloca memoria per un intero  
int *p; // Crea la variabile puntatore  
p = &i; // assegna a p l'indirizzo della locazione  
        // associata ad i  
*p = 10; // OK  
printf("i = %d\n", i); // Stampa 10.
```

ATTENZIONE alle dichiarazioni multiple!

Esempio 3.5.

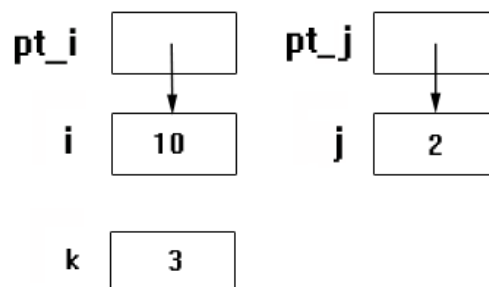
```
int *p1, p2;
```

p2 non è di tipo puntatore, ma semplicemente intero!

3.1.4. Esempio: uso di variabili puntatore

```
int i,j,k;  
int *pt_i, *pt_j;  
pt_i = &i;  
pt_j = &j;  
i = 1;  
j = 2;  
k = *pt_i + *pt_j;  
*pt_i = 10;  
printf("i = %d\n", i);  
printf("k = %d\n", k);
```

In questo frammento di programma, le variabili di tipo `int` `i`, `j`, `k`, vengono utilizzate tramite puntatori alle locazioni di memoria ad esse associate al momento della dichiarazione.



Il programma stampa:

```
i = 10
```

```
k = 3
```

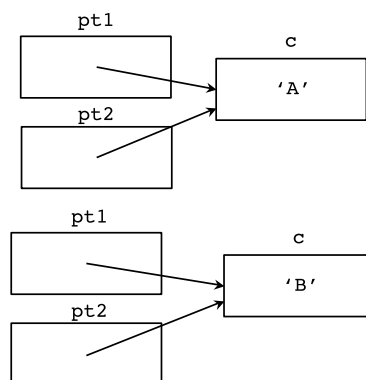
3.1.5. Condivisione di memoria

Un'area di memoria a cui fanno riferimento due o più puntatori è detta *condivisa*. In questo caso, le modifiche effettuate tramite un puntatore sono visibili tramite l'altro (e viceversa).

Esempio 3.6.

```
char c = 'A';
char* pt1 = &c;
char* pt2 = pt1;
printf("*pt1 = %c\n",*pt1);
printf("*pt2 = %c\n",*pt2);
*pt2 = 'B'; // Modifica tramite pt2
printf("%c\n",c); // Ovviamente, c e' cambiata!
printf("*pt1 = %c\n",*pt1); // anche pt1 vede la modifica
```

Le figure seguenti mostrano lo stato della memoria prima e dopo la modifica effettuata tramite pt2.



3.1.6. Assegnamento

A variabili di tipo puntatore possono essere assegnati valori corrispondenti ad indirizzi di memoria.

Esempio 3.7.

```
int* p;
int q = 10;
char c = 'x';
p = &q; // OK
p = &c; // WARNING! p punta a int mentre c e' char
```

3.1.7. Uguaglianza tra puntatori

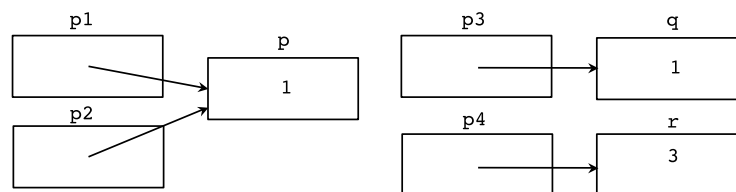
Quando si usa l'operatore di uguaglianza (==) tra puntatori occorre prestare particolare attenzione. L'operatore, infatti, verifica l'uguaglianza tra gli indirizzi contenuti nelle variabili puntatore, non tra il contenuto delle variabili puntate.

Esempio 3.8.

```

int p=1, q=1, r=3;
int *p1=&p, *p2=&p, *p3=&q, *p4=&r;
if(p1 == p2){...}// TRUE: p1 e p2 puntano alla stessa variabile
if(*p1 == *p2){...}// TRUE: Le variabili puntate da p1 e p2 hanno
// stesso valore
if(p2 == p3){...}// FALSE: p2 e p3 puntano a variabili diverse
if(*p2 == *p3){...}// TRUE!: Le variabili puntate da p2 e p3 hanno
// stesso valore
if(p3 == p4){...}// FALSE: p3 e p4 puntano a variabili diverse
if(*p3 == *p4){...}// FALSE!: Le variabili puntate da p3 e p4 hanno
// valori diversi

```



3.2. Aritmetica dei puntatori

Alle variabili di tipo puntatore è possibile aggiungere o sottrarre valori interi. Tali operazioni hanno una *semantica diversa* rispetto al caso degli interi. Inoltre, queste due operazioni, insieme alla differenza tra puntatori (discussa più avanti) rappresentano le uniche operazioni aritmetiche disponibili sui puntatori.

3.2.1. Somma di un valore intero ad un puntatore

Esempio 3.9. Si consideri il seguente esempio

```

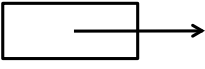
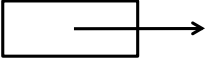
int i = 10;
int* p = &i;
int* q = p + 2;

```

Al termine dell'esecuzione, a *q* viene assegnato l'indirizzo della locazione di memoria ottenuta valutando l'espressione *p + 2* come segue:

- si considera l'indirizzo a cui *p* punta (&*i*);
- si considera la dimensione *N* in byte del tipo a cui la variabile *p* punta (4 byte per *int*);
- si moltiplica il valore sommato a *p*, cioè 2, per *N*;

- si restituisce il valore dell'indirizzo puntato da p , incrementato del valore ottenuto sopra.

	indirizzo	contenuto	
	0x00AF0000	00100010	
p 	0x00AF0001	00001010	Celle allocate per i (4 byte)
	0x00AF0002	00000000	
	0x00AF0003	00000000	
	0x00AF0004	00000000	
	
q 	0x00AF0009	11011111	
	

Si noti che N dipende dal tipo di p , NON di q .

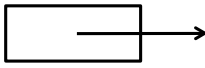
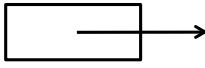
ATTENZIONE: la locazione a cui punta q dopo l'assegnamento potrebbe non essere valida (cioè non allocata).

3.2.2. Sottrazione di un valore intero da un puntatore

Il caso della sottrazione di un valore intero da un puntatore è duale rispetto a quanto visto per la somma, con l'indirizzo di partenza decrementato del valore intero moltiplicato per N .

Esempio 3.10.

```
char c = 'd';
char *pc = &c;
char *p = pc - 4;
```

	indirizzo	contenuto	
	0x1BA01004	00101110	
p 	0x1BA01005	00000000	
	0x1BA01006	10100010	
	0x1BA01007	00011010	
	0x1BA01008	00001010	
pc 	0x1BA01009	01000000	1 byte allocato per c
	

3.3. Puntatori costanti

La specifica `const` può essere applicata anche a variabili di tipo puntatore.

Esempio 3.11.

```
double pi = 3.5;
double const *pt = &pi;
(*pt)++; // OK (pi vale 4.5)
pt++; // NO: pt e' costante!
```

In questo caso, la specifica `const` si applica alla variabile puntatore, non alla locazione puntata (il cui contenuto può quindi cambiare).

3.4. Puntatori a costanti

Si può dichiarare un puntatore a valori costanti.

```
const int k = 3;
const int *pt; // dichiarazione di puntatore a costante
pt = &k; // ok
pt++; // ammesso sebbene non sia noto a cosa punti pt
int *pti;
int w;
pti = &k; // no: k e' costante
pt = &w; // no: w e' variabile
```

Si noti che se la penultima istruzione fosse legale, potremmo modificare, tramite il puntatore `pti`, il valore assegnato ad una costante.

3.5. Puntatori a puntatori

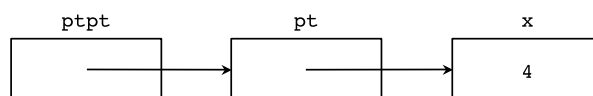
Come per ogni altro tipo si può definire un puntatore ad una variabile di tipo puntatore.

Esempio 3.12. Il seguente frammento di codice

```
double x;
double * pt;
double ** ptpt;

x = 4;
pt = &x;
ptpt = &pt;
printf("%lf\n", **ptpt);
```


stampa il valore di x.



3.6. Il puntatore NULL

Le variabili di tipo puntatore possono assumere anche il valore speciale NULL. Tale valore specifica che la variabile non punta ad alcuna locazione di memoria. Si noti la differenza tra una variabile (puntatore) il cui valore è NULL, che risulta pertanto inizializzata, ed un variabile non inizializzata, il cui valore non è noto. Il confronto con NULL può essere usato per verificare se una variabile di tipo puntatore punta ad una locazione di memoria.

Esempio 3.13.

```
int *pt = NULL;
if (pt!=NULL)
    *pt=10;
```

In questo caso il ramo if non viene eseguito. L'istruzione `*pt=10;` eseguita al momento in cui `pt` vale NULL genererebbe un errore a tempo di esecuzione.

3.7. Il tipo void*

Poiché la memoria allocata per i puntatori è fissa (rappresenta un indirizzo) si può omettere la specifica del tipo della variabile puntata, dichiarando un puntatore di tipo `void*`.

Esempio 3.14.

```
void* pt;
int i;
pt = &i;
```

I puntatori di tipo `void` sono utilizzati quando il tipo dei valori da trattare non è noto a priori. Un caso tipico si ha nell'uso delle istruzioni per l'allocazione dinamica di memoria (v. seguito). Si osservi che il contenuto della locazione puntata da un puntatore `void *` non può essere assegnato direttamente ad un'altra variabile (di qualsiasi tipo).

Esempio 3.15.

```
void* pt;
...
int j = *pt; //ERRORE!
```

Per poter effettuare questa assegnazione, è necessario prima eseguire una conversione.

3.8. Conversione di puntatori

Anche le variabili di tipo puntatore possono essere convertite (automaticamente o mediante casting).

Esempio 3.16.

```
void* pt;
...
int* pti = pt; // Conversione automatica
int* pti2 = (int*) pt; // Casting esplicito
```

Dopo la conversione è possibile assegnare il contenuto della locazione puntata ad una variabile (di tipo opportuno).

```
int j = *pti2; // pti2 e' un puntatore a int
```

... o più brevemente:

```
int j = *((int *)pt); // ((int *)pt) e' un puntatore a int
```

Si noti che l'espressione `((int *)pt)` restituisce l'indirizzo della locazione puntata da `pt`, convertita in riferimento a valore di tipo intero. L'uso dell'operatore di dereferenziazione permette quindi di accedere a tale valore.

3.9. Allocazione di memoria

La funzione `malloc`, definita nella libreria `stdlib.h`, permette al programmatore di allocare dinamicamente spazio di memoria.

Invocazione malloc

Sintassi:

```
malloc(n);
```

- *n* è la dimensione in byte della memoria da allocare;

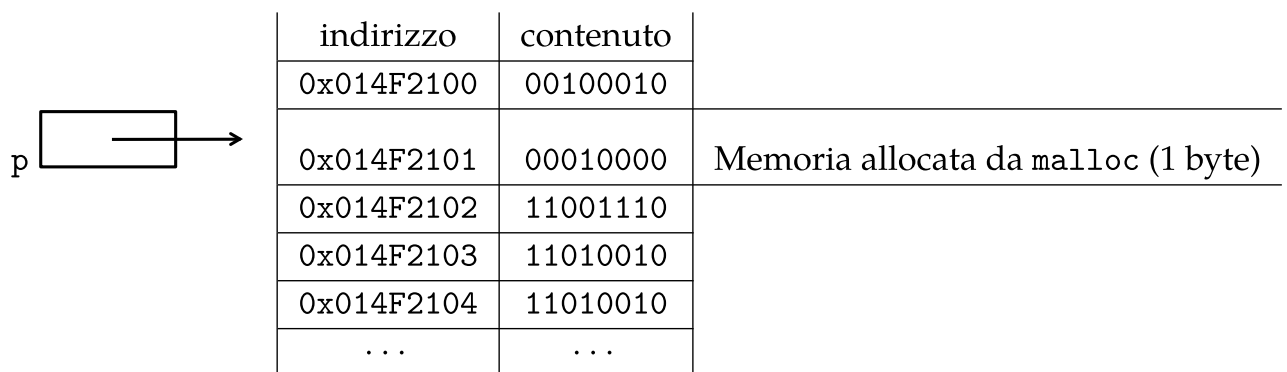
L'invocazione `malloc(n)`:

- alloca n byte contigui di memoria;
- restituisce un puntatore di tipo `void*`, contenente l'indirizzo alla prima cella allocata.

Per accedere correttamente alla memoria allocata, il puntatore restituito deve essere convertito nel tipo opportuno.

Esempio 3.17.

```
#include<stdlib.h>
...
char* p = (char*) malloc(sizeof(char));
...
```



indirizzo	contenuto
0x014F2100	00100010
0x014F2101	00010000
0x014F2102	11001110
0x014F2103	11010010
0x014F2104	11010010
...	...

Memoria allocata da malloc (1 byte)

3.9.1. Deallocazione di memoria

La memoria allocata e non più utilizzata deve essere *rilasciata*. Una delle funzioni C che permette di rilasciare la memoria associata ad un puntatore è `free`.

Esempio 3.18.

```
#include<stdlib.h>
char* p = (char*) malloc(sizeof(char));
...
free(p); // rilascia la memoria puntata da p
```

Si noti che la funzione `free` può essere invocata solo su un puntatore che fa riferimento ad una locazione precedentemente allocata dinamicamente (ovvero, nel nostro caso, tramite `malloc`). L'invocazione su locazioni allocate diversamente (ad esempio, ottenute tramite riferimento a variabili dichiarate) genera un comportamento indefinito.

Una volta rilasciata, la memoria libera può essere nuovamente allocata, ad esempio tramite un'altra invocazione di `malloc`.

Si noti che dopo il rilascio, `p` punta ad una locazione libera.

Omettere il rilascio della memoria non più utilizzata può comportare la saturazione della memoria disponibile.

Esempio 3.19.

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

int main(void){
    long int x = 0;
    printf("size long int %ld\n", sizeof(long));
    printf("size x di tipo long int %ld\n", sizeof(x));
    long * p;
    while (1) {
        p = (long *) malloc(1000000000000001);
        *p = 101;
        x++;
        printf("x e' %ld\n", x);
    }
}
```

3.10. Ancora sulla funzione `scanf`

I parametri di `scanf()`, a partire dal secondo, denotano puntatori alla memoria in cui memorizzare i valori letti. Come visto, data una variabile `x`, il rispettivo indirizzo di memoria può essere ottenuto tramite l'operatore `&`.

Esempio 3.20.

```
int x;
scanf("%d",&x); // &x e' un puntatore alla locazione
                // contenente il valore assegnato ad x
```

Quando il riferimento è memorizzato in una variabile puntatore, può essere usato direttamente.

Esempio 3.21.

```
int* p = (int*) malloc(sizeof(int));  
printf("Inserisci un valore intero\n");  
scanf("%d",p);  
printf("Il valore inserito e': %d\n", *p);
```