Introduzione alla programmazione in C

Appunti delle lezioni di Tecniche di programmazione

Giorgio Grisetti
Luca Iocchi
Daniele Nardi
Fabio Patrizi
Alberto Pretto

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Facoltà di Ingegneria dell'Informazione, Informatica, Statistica
Università di Roma "La Sapienza"

Edizione 2018/2019

2019

Indice

3. Puntatori

3.1. Memoria, indirizzi e puntatori

La memoria di un elaboratore 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
...	...

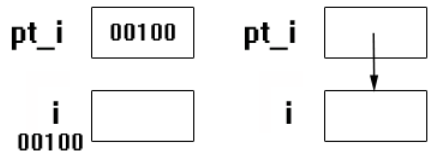
Una variabile identifica un certo numero di celle contigue, dipendente dal tipo di dato che memorizza (ad es., 1 cella per char, 4 celle per int). Accedendo ad una variabile, accediamo al contenuto delle celle identificate dalla variabile. In C il programmatore ha la possibilità di gestire gli indirizzi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

---

[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.

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.

stampa-puntatore.c

```
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' 15
```

L'operatore & si chiama operatore *indirizzo-di* e restituisce l'indirizzo della prima cella di memoria 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 indirizzamento indiretto

L'operatore di indirizzamento indiretto * (si può chiamare anche operatore di *dereferenziamento* o *contenuto-di*) permette di accedere al valore contenuto nella locazione di memoria identificata da un certo indirizzo.

Esempio: 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 i = *&j; equivale all'istruzione i = j;.

L'operatore di indirizzamento indiretto * si può usare anche nella parte sinistra dell'istruzione di assegnazione e in questo caso consente l'assegnazione del risultato dell'espressione a destra dell'operatore = nella zona di memoria puntata dal puntatore.
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: L'istruzione

```
int *p1;
```

dichiara p1 come variabile di tipo puntatore, specificando che il tipo contenuto nella locazione puntata è intero.
ATTENZIONE: La dichiarazione di una variabile puntatore non alloca memoria per la variabile puntata.

Esempio:

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

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:

```
int i;  // Alloca memoria per un intero
int *p; // Alloca memoria per 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:

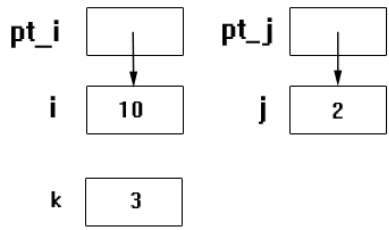
```
int *p1, p2;
```

p1 è di tipo puntatore a intero, mentre p2 è di tipo intero!

3.1.4. Esempio: uso di variabili puntatore

```
int i,j,k;
int *pt_i, *pt_j, *pt_k;
pt_i = &i;
pt_j = &j;
pt_k = &k;
*pt_i = 1;
*pt_j = 2;
*pt_k = *pt_i + *pt_j;
*pt_i = 10;
printf("i = %d\n", *pt_i);
printf("j = %d\n", *pt_j);
printf("k = %d\n", *pt_k);
printf("i = %d\n", i);
printf("j = %d\n", j);
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
j = 2
k = 3
i = 10
j = 2
k = 3
```

3.1.5. Condivisione di memoria

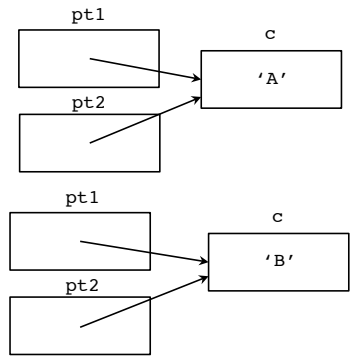
Un'area di memoria a cui fanno riferimento due o più puntatori è detta *condivisa*. In questo caso, le modifiche effettuate tramite un puntatore sono visibili tramite l'altro (e viceversa).

Esempio:

condivisione.c

```
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. Assegnazione

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

Esempio:

assegnazione-puntatore.c

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

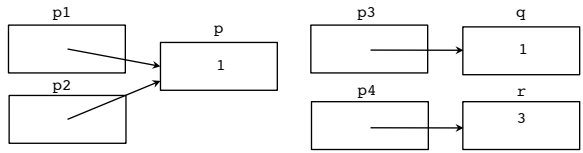
NOTA: L'assegnazione tra variabili puntatori e indirizzi di variabili di tipo non compatibile (come nell'esempio) provoca tipicamente solo un warning con il compilatore C, mentre è considerato un errore dal compilatore C++.

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:

```
int p=1, q=1, r=3;
int *p1=&p, *p2=&p, *p3=&q, *p4=&r;
if(p1 == p2)  // TRUE: p1 e 2 puntano alla stessa
    variabile
if(*p1 == *p2) // TRUE: variabili puntate da p1 e p2
    hanno stesso valore
if(p2 == p3)  // FALSE: p2 e p3 puntano a variabili
    diverse
if(*p2 == *p3)  // TRUE: variabili puntate da p2 e p3
    hanno stesso valore
if(p3 == p4)  // FALSE: p3 e p4 puntano a variabili
    diverse
if(*p3 == *p4)  // FALSE: 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 disponibile sui puntatori.

3.2.1. Somma di un valore intero ad un puntatore

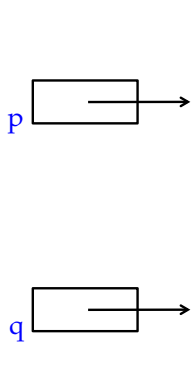
Esempio: Si consideri il seguente esempio

```
int i = 10;
int* p = &i;
int* q = p + 2;  // indirizzo di p + 2 * sizeof(int)
```

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	
0x00AF0001	00001010	Celle allocate per i (4 byte)
0x00AF0002	00000000	
0x00AF0003	00000000	
0x00AF0004	00000000	
...	...	
0x00AF0009	11011111	
...	...	

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

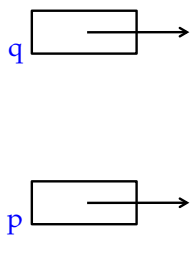
ATTENZIONE: la locazione a cui punta q dopo l'assegnazione potrebbe non essere valida (cioè non allocata).

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:

```
char c = 'd';
char *p = &c;
char *q = p - 4; // indirizzo di p - 4 * sizeof(char)
```



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

3.3. Puntatori a costanti

La specifica const può essere applicata anche a variabili di tipo puntatore e specifica che il contenuto della *locazione puntata* è costante (cioè non può essere modificato).

Esempio:

```
double pi = 3.5;
const double *pt = &pi;
(*pt)++; // ERRORE *pt e' costante
pt++;    // OK: pt non e' costante
```

3.4. Puntatori a puntatori

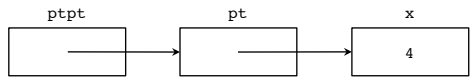
Come per ogni altro tipo si può definire un puntatore ad una variabile di tipo puntatore.

Esempio: Il seguente frammento di codice

```
int x;
int *pt;
int **ptpt;

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

stampa il valore di x, cioè 4.



3.5. Il valore 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 valida. 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 valida o meno.

Esempio:

```
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 genera un errore a tempo di esecuzione, in quanto non è possibile accedere ad un'area di memoria valida se il valore del puntatore è NULL.

3.6. Il tipo void*

Poiché la dimensione della 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:

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

Il tipo void* è compatibile con qualsiasi altro tipo puntatore.

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:

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

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

3.7. Conversione di puntatori

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

Esempio:

```
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 dereferenziamento permette quindi di accedere a tale valore.

3.8. Allocazione dinamica della memoria

L'*allocazione dinamica* della memoria è realizzata da opportune funzioni che gestiscono un'area di memoria chiamata *heap*. In quest'area, infatti,(a differenza dello stack (v.seguito), la allocazione/deallocazione della memoria ha luogo solo su esplicita richiesta del programma.

3.8.1. Funzione malloc

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;

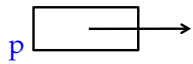
Semantica:
L'invocazione malloc(n):

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

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

Esempio:

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



indirizzo	contenuto	
0x014F2100	00100010	
0x014F2101	00010000	allocata da malloc (1 byte)
0x014F2102	11001110	
0x014F2103	11010010	
0x014F2104	11010010	
...	...	

Tipicamente la dimensione della memoria da allocare dinamicamente è calcolata con un espressione del tipo

```
n * sizeof(tipo)
```

specificando quindi il numero e il tipo delle informazioni da allocare.

Esempio:

```
int *a = (int *)malloc(10*sizeof(int));
```

alloca memoria per 10 interi.

3.8.2. Recupero della memoria

In generale, esistono diversi meccanismi che consentono di recuperare la memoria allocata dinamicamente quando questa non è più necessaria al programma:

Garbage collection: la memoria viene recuperata, ovvero rilasciata, da una speciale funzione (*garbage collector*) che, eseguita periodicamente senza l'intervento del programmatore, si occupa di identificare le aree di memoria allocate per le quali non sia disponibile un riferimento (puntatore) all'interno nel programma. Tale meccanismo non è disponibile in C (ma lo è, ad esempio, in Python o in Java).

Deallocazione esplicita: la memoria deve essere rilasciata esplicitamente dal programma tramite l'invocazione di una funzione specifica.

Si noti che l'uso del garbage collector potrebbe evitare il problema evidenziato nell'esempio precedente. Infatti, ad ogni iterazione, la memoria precedentemente allocata risulta inaccessibile e può essere rilasciata. In generale, tuttavia, esistono situazioni in cui la memoria può essere esaurita, anche in presenza di garbage collection.

3.8.2.1. Deallocazione

Il C delega al programmatore il compito di rilasciare la memoria inutilizzata. *R*ilasciare o *d*eallocare un'area di memoria significa renderla disponibile per usi futuri, in particolare, ad esempio, per l'allocazione dinamica di nuove variabili.

La funzione preposta a questo scopo ha la seguente intestazione:

```
void free(void* p)
```

Quando invocata con parametro un puntatore ad un'area di memoria allocata dinamicamente, free si occupa di rilasciare l'area precedentemente allocata. È necessario che il puntatore passato alla funzione free

sia stato restituito da una funzione di allocazione dinamica (malloc, realloc o calloc –v. seguito) oppure sia il puntatore NULL, caso in cui la funzione non ha effetto. In tutti gli altri casi, free ha un comportamento indefinito.

Esempio: Si consideri il seguente frammento di codice:

free.c

```
int *p = (int *) malloc(sizeof(int));
*p = 0;
int *q = (int *) malloc(sizeof(int));
* q = 10;
free (p);
int* r = (int *) malloc(sizeof(int));
```

Notiamo che l'indirizzo assegnato a q è sicuramente diverso da quello memorizzato in p, in quanto la memoria a cui p fa riferimento è già allocata e non può esserlo nuovamente. Dopo l'esecuzione di free(p), invece, la memoria a cui p faceva riferimento è diventata libera e può quindi essere riutilizzata per nuove allocazioni; ad esempio, potrebbe (ma non deve necessariamente) essere riusata nell'ultima invocazione di malloc, caso in cui il puntatore r finirebbe per contenere lo stesso valore di p.

Si osservi che dopo l'esecuzione di free(p) il valore di p rimane inalterato. In altre parole, p continua a puntare alla stessa locazione di memoria cui puntava inizialmente, sebbene questa sia stata rilasciata (e potenzialmente riallocata).

Chiaramente, l'accesso ad un'area di memoria deallocata deve essere evitato, in quanto tale area non fornisce nessuna garanzia sui dati che essa contiene.

Esempio: Si completi l'esempio precedente con il seguente frammento di codice:

```
*r = 100;
printf("%d\n",*p);
```

Poiché l'indirizzo contenuto in p è rimasto inalterato, p continua a puntare allo stesso indirizzo cui puntava prima dell'invocazione a free. Pertanto, se la successiva invocazione a malloc non ha riutilizzato la memoria rilasciata, la stampa di *p produce ancora 0. Tuttavia, l'indirizzo puntato da p è rimasto disponibile per nuove allocazioni. In particolare

esso potrebbe essere stato usato per allocare la variabile puntata da r che, a seguito dell'assegnazione *r = 100 contiene ora il valore 100. In tal caso, ovviamente, la stampa di *p produrrebbe 100.

Nell'esempio precedente, il valore assunto dalla variabile puntata da p è essenzialmente arbitrario, in quanto dipendente dalle scelte effettuate dal sistema e non dal programmatore. Nella pratica esistono situazioni in cui l'accesso ad una stessa variabile tramite puntatori diversi corrisponde ad una precisa scelta. In questi casi é necessario indicare esplicitamente questa volontà (e non sperare che il sistema si comporti come desiderato). Nel caso in esame, avremmo potuto esplicitamente assegnare a p il valore di r, tramite l'istruzione: p = r.

3.8.2.2. Puntatori *appesi*

Il problema dei *puntatori appesi* (*dangling pointers*) si verifica quando un puntatore si trova a fare riferimento ad un'area di memoria non allocata. Ciò può avvenire in maniera diretta, ovvero quando la memoria puntata da un puntatore viene deallocata tramite il puntatore stesso e l'indirizzo cui il puntatore fa riferimento non viene cambiato, oppure in maniera indiretta, per effetto della deallocazione di memoria a partire da un altro puntatore. Mostriamo di seguito un esempio del secondo caso.

Esempio:

dangling-pointer.c

```
int* q = (int *) malloc(sizeof(int));
int* p = q;
*q = 10;
free (p);
printf("%d\n",*q);
```

Per effetto del rilascio della memoria puntata dalla variabile p, la variabile q fa riferimento ad un'area di memoria non allocata, quindi suscettibile di modifiche arbitrarie.

Situazioni di questo genere devono essere evitate, in quanto fonti di errori difficili da individuare. Un approccio possibile, mostrato nel seguente esempio, consiste nell'assegnare il valore NULL ad un puntatore ogni volta che la memoria cui esso fa riferimento viene rilasciata (direttamente o indirettamente) e nel verificare che un puntatore faccia

riferimento ad un indirizzo non NULL prima accedere alla locazione da esso puntata.

Esempio:

fixed-dangling-pointer.c

```
int* q = (int *) malloc(sizeof(int));
int* p = q;
*q = 10;
free (p);
p = NULL;
q = NULL;
// ...
if (q != NULL)
    printf("%d\n",*q);
//...
```

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

Esempio:

finisci-memoria.c

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

int main(void){
  long int s = 0;
  long * p;
  while (1) {
    p = (long *) malloc(1000000); // 1 MiB
    *p = 10l;
    s++;
    printf("Allocati %d MiB\n", s);
  }
}
```

3.8.3. Tempo di vita delle variabili allocate dinamicamente

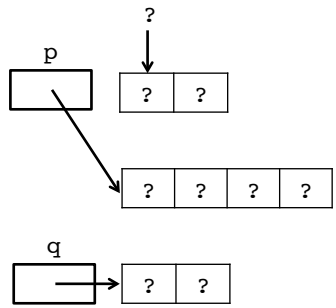
Il *tempo di vita* delle variabili allocate dinamicamente corrisponde al periodo che intercorre tra l'allocazione e la deallocazione della variabile. Se una variabile allocata dinamicamente non viene deallocata esplicitamente, il suo tempo di vita termina con il programma. Pertanto, il tempo di vita di una variabile allocata dinamicamente è noto solo a tempo di esecuzione.

Nonostante una variabile possa rimanere in vita per l'intera durata di un programma, si può verificare il caso in cui la corrispondente memoria non sia più accessibile. Questo accade quando non ci sono variabili di tipo puntatore che fanno riferimento a quell'area di memoria. Quando ciò si verifica, non vi è alcun modo di recuperare il riferimento ed accedere nudamente all'area di memoria: essa rimarrà inaccessibile per tutta la durata del programma ma, poiché ancora allocata, inutilizzabile per allocare nuova memoria. Tali situazioni sono ovviamente da evitare.

Esempio: Nel seguente frammento di codice, dopo la seconda linea, viene perso il riferimento alla prima area allocata, che rimane inutilmente allocata per tutta la durata dell'esecuzione del programma.

riferimento-perso.c

```
void* p = malloc(2);
p = malloc(4);
void* q = malloc(2);
```



Si osservi che oltre ad essere non più utilizzabile, la prima area di memoria, poichè già allocata, non può essere nemmeno sfruttata per allocare una nuova area.

3.9. Lettura tramite puntatori

I parametri di scanf, a partire dal secondo, denotano indirizzi di 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:

```
int x;
scanf("%d",&x);// &x indirizzo della locazione
               // contenente il valore inserito
```

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

Esempio:

scanf.c

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