

# Sistemi Operativi

## APPUNTI SUL LINGUAGGIO C



## Richiami al linguaggio C

# Il linguaggio C

Il C è un linguaggio di programmazione di tipo  
**imperativo**

un programma è una lista di istruzioni, di imperi  
che il calcolatore deve eseguire.

Versione standard del C: ANSI/ISO 9899

## Un semplice programma C

```
/* Programma che stampa un saluto */  
  
#include <stdio.h>  
  
main() {  
  
    printf("Hello World!\n");  
  
}
```

# Un semplice programma C

```
/* Programma che stampa un saluto */
```

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

```
main() {
```

```
printf("Hello World!\n");
```

```
}
```

L'istruzione è chiamata “**direttiva di compilazione**” e serve ad includere informazioni relative a una libreria predefinita del C che contiene le funzioni di input/output.

# Un semplice programma C

```
/* Programma che stampa un saluto */  
  
#include <stdio.h>  
  
main() {  
    printf("Hello World!\n");  
}
```

La parola **main()** identifica il programma principale e rappresenta il punto di ingresso del programma all'inizio della sua esecuzione

# Un semplice programma C

```
/* Programma che stampa un saluto */
```

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

```
main() {
```

```
printf("Hello World!\n");
```

```
}
```

Si possono  
specificare  
**commenti**  
racchiudendoli  
tra i simboli  
`/* e */`

**Ogni** istruzione è conclusa  
dal simbolo `<<;>>`

Le parentesi graffe delimitano un **blocco**

# Un semplice programma C

```
/* Programma che stampa un saluto */  
  
#include <stdio.h>  
  
main() {  
    printf("Hello World!\n");  
}
```

Per la stampa su video **printf** e tra apici la stringa costante

Esistono sequenze nelle stringhe che indicano caratteri speciali: per esempio **\n** indica il carattere di new line che quando viene incontrato sposta il carattere successivo alla riga seguente

# Caratteri speciali

Tipo di opzione	Descrizione
\n	Ritorno a capo
\t	Tabulazione orizzontale
\b	Tabulazione verticale
\a	Torna indietro di uno spazio
\f	Salto pagina



# Variabili in C

```
/* Programma per il calcolo del fattoriale */

#include <stdio.h>

main() {

    int n,fat;

    printf("Calcolo del fattoriale di:");
    scanf("%d",&n);

    fat = 1;

    while (n>1) {
        fat = fat * n;
        n = n-1;
    }

    printf("Risultato = %d\n",fat);

}
```

# Variabili in C

```
/* Programma per il calcolo del fattoriale */  
  
#include <stdio.h>  
  
main() {  
    int n,fat;  
  
    printf("Calcolo del fattoriale di:");  
    scanf("%d",&n);  
  
    fat = 1;
```

Una **variabile** viene dichiarata scrivendo il tipo seguito dal nome della variabile.  
Una variabile può essere inizializzata all'atto della sua dichiarazione (int a=1).

## Tipi di dato elementari

Tipi di dichiarazione	Rappresentazione
Char	Carattere (es. 'à')
Int	Numero intero (es. 3)
Short	Numero intero corto
Long	Numero intero lungo
Float	Numero reale "corto" (es 14.4)
Double	Numero reale "lungo"

In C **non esiste il tipo boolean**: Si usa la convenzione che lo zero rappresenta il valore **falso** e l'uno il valore **vero** (tutti i valori diversi da zero rappresentano il vero)

# Input/Output

```
/* Programma per il calcolo del fattoriale */  
  
#include <stdio.h>  
  
main() {  
  
    int n,fat;  
  
    printf("Calcolo del fattoriale di:");  
    scanf("%d",&n);  
  
    fat = 1;
```

La lettura e la stampa di variabili richiede spesso la specifica del loro formato: le istruzioni **printf**, **scanf** hanno in genere più argomenti.

## Argomenti di printf e scanf

```
printf ( "<stringa>" , <elenco argomenti> );
```

```
scanf ( "<stringa>" , <elenco argomenti> );
```

La lettura e la stampa di variabili richiede spesso la specifica del loro formato: le istruzioni **printf**, **scanf** hanno in genere più argomenti.

```
/* Programma per  
  
#include <stdio.h>  
  
main() {  
  
    int n,fat;  
  
    printf("Calcolo del fattoriale di:");  
    scanf("%d",&n);  
  
    fat = 1;  
  
    ----  
  
    printf("Risultato = %d\n",fat);
```

## Argomenti di printf e scanf

- Il primo è una stringa di caratteri (da stampare per la printf) nella quale ogni % indica il punto in cui vanno sostituiti, nell'ordine, gli argomenti che seguono;
- Il carattere che segue il simbolo % indica il tipo dell'argomento (d indica un valore intero);
- Gli altri argomenti specificano le variabili di input/output (quelle di input sono precedute dal simbolo speciale &).

**scanf ( “%d” , &n );**

Tipo di argomento da inserire

Assegna alla variabile **n**  
l'argomento inserito

Tipo di argomento da stampare

Stampa nel punto indicato  
il valore contenuto dalla  
variabile **fat**

**printf ( “ Risultato = %d \n ” , fat );**



# Argomenti di printf e scanf

Sintassi da utilizzare	Descrizione
%d	Dati di tipo int
%lf %l %f	Dati di tipo double Dati di tipo long Dati di tipo float
%c	Dati di tipo char
%s	Dati di tipo stringhe

# Assegnazione in C

```
/* Programma per il calcolo del fattoriale */  
  
#include <stdio.h>  
  
main() {  
  
    int n,fat;  
  
    printf("Calcolo del fattoriale di:");  
    scanf("%d",&n);  
  
    fat = 1;
```

L'istruzione di assegnazione si indica con il simbolo =

# Blocchi in C

```
while (n>1) {  
    fat = fat * n;  
    n = n-1;  
}
```

```
printf("Risultato = %d\n",fat);
```

```
}
```

Un'istruzione composta è delimitata da un blocco

# Operatori ed espressioni in C

Operazioni con gli int (interi)	Descrizione delle operazioni
* moltiplicazione	$4*5=20$ . Moltiplica i numeri inseriti
+ addizione	$2+10=12$ Somma i numeri inseriti
- sottrazione	$3-2=1$ Sottrae i numeri inseriti
/ divisione	$5/4=1$ Divide e il risultato è senza resto
% divisione con modulo	$10\%7=3$ Divide e come risultato abbiamo il resto

## Operatori ed espressioni in C

Operazioni con i double (reali)	Descrizione delle operazioni
* moltiplicazione	4.5*2.0=9.0 Moltiplica i numeri inseriti
+ addizione	2.0+10.2=12.2 Somma i numeri inseriti
- sottrazione	3.0-2.1=0.9 Sottrae i numeri inseriti
/ divisione	3.0/2.0=1.5

Tipo di espressione	Descrizione
x++	Incremento della variabile x di 1
y--	Decremento della variabile y di 1
a+=b e a*=b	a=a+b e a=a*b

# Operatori ed espressioni in C

Operatori relazionali	Descrizione
<code>x == y</code>	Testa se il valore di x è uguale a y
<code>x &gt; y</code>	Testa se x è maggiore di y
<code>x &gt;= y</code>	Testa se x è maggiore uguale di y
<code>x &lt; y</code>	Testa se x è minore di y
<code>x &lt;= y</code>	Testa se x è minore uguale di y
<code>x != y</code>	Testa se x è diverso da y

# Operatori ed espressioni in C

Operatori logici	Descrizione
&&	AND
	OR
!	NOT

Operatore condizionale	Descrizione
<op1>?<op2>:<op3>	Vale <op2> se <op1> è vero altrimenti <op3> Es. (a>b)?a:b calcola il massimo tra a e b

# Istruzioni di controllo condizionali

- Istruzione condizionale **if**

```
if ( <espr> ) <istr1>  
    else <istr2>
```

Se <espr> è vera viene eseguito <istr1> altrimenti verrà eseguito <istr2>

Es. **if** (a>b)

```
    printf("il maggiore è %d", a);  
else  
    printf("il maggiore è %d", b);
```



# Istruzioni di controllo condizionali

- Istruzione condizionale **switch**

```
switch ( <espr> ) {  
    case <costante1> : <istr1> [break]  
    case <costante2> : <istr2> [break]  
    ...  
    default : <istr>  
}
```

Se <espr> vale <costante n> vera viene eseguito <istr n>; in tutti gli altri casi (caso di default) verrà eseguito <istr>.

Per convenzione dopo l'istruzione che si vuole eseguire si usa il comando **break** per uscire dall'istruzione condizionale.

# Istruzioni di controllo condizionali

Es.

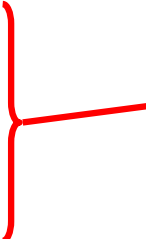
Vogliamo impostare il numero di giorni n di cui è fatto ogni mese:

```
switch ( mese ) {  
    case 2: n=28; break;  
    case 4 : case 6:  
    case 9 : case 11: n=30; break;  
    default : n=31;  
}
```

# Istruzioni di controllo iterative

- Istruzione **while**

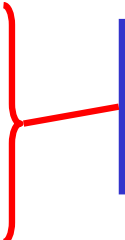
**while** ( <espr> )  
    <istr>



Prima valuta **espr** e poi esegue **istr**

oppure

**do** <istr>  
**while** ( <espr> )



Prima esegue **istr** poi valuta **espr**

Fino a che <espr> è vera viene eseguito <istr>.

# Istruzioni di controllo iterative

Es.

Calcoliamo il fattoriale di un numero i:

```
while ( i>1 ) {  
    fat *= i;  
    i--;  
}
```



```
do {  
    fat *= i;  
    i--;  
}  
while ( i>1 );
```

# Istruzioni di controllo iterative

- Istruzione **for**

```
for ( <espr_init>; <espr_test>; <espr_incr> )  
    <istr>
```

corrisponde a:

```
<espr_init>  
while ( <espr_test> ) {  
    <istr>  
    <espr_incr>;  
}
```

**Es.**      **for** ( i=n; i>1; i-- ) fat \*= i;

# Costanti simboliche in C

**#define** <nome> <valore>

Si definiscono facendo uso della direttiva **define** che va messa nell'intestazione del programma.

Es.

```
#define PIGRECO 3.14  
#define N 100  
#define TRUE 1  
#define FALSE 0
```

```
main() {  
    ...  
}
```

# Funzioni in C

Una **funzione** è un blocco di istruzioni, che ha parametri in ingresso (**parametri formali**) e restituisce un **risultato**.

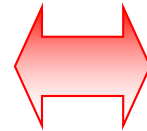
```
<tipo> <Nome funzione> (lista parametri) {  
    ...  
    blocco istruzioni  
    ...  
    return <risultato>  
}
```

Lista parametri: (<tipo> <nome>, <tipo> <nome>, ...)

## Funzioni in C

Es.

```
int Fatt(int n) {  
    int ris;  
    ris = 1;  
    while (n>=1) {  
        ris = ris*n;  
        n--;  
    }  
    return ris;  
}
```



```
int i,j,k;  
...  
i = Fatt(4);  
scanf("%d",&k);  
j = Fatt(K);
```

Una funzione viene attivata mediante una chiamata nella quale vengono passati i parametri attuali che devono corrispondere in numero, ordine e tipo ai parametri formali.



# Definizione e implementazione

In C si distingue tra **definizione** e **implementazione** di una funzione.

Una **definizione** specifica la struttura della funzione: numero, ordine e tipo dei parametri di ingresso/uscita. La definizione di una funzione è detta in C **prototipo** o **header**.

```
/* prototipi di funzioni */  
int Fatt(int);  
float radice(int);  
int somma(int, int);  
char primo(int);
```

## Definizione e implementazione

Una **implementazione** comporta invece la specifica completa della funzione.

```
/* implementazione della funzione somma */  
int somma(int a, int b) {  
    int sum;  
    sum = a+b;  
    return sum;  
}
```

Il compilatore C permette l'uso di una funzione solo se ha già incontrato (almeno) la sua definizione.

# Organizzazione dei programmi C

La programmazione C è **strutturata**: un programma complesso (e quindi **modulare**) è generalmente distribuito su più file.

Per includere un file in un altro si usa, per file nel direttorio di sistema (librerie standard), la direttiva di compilazione:

```
#include <..>
```

oppure per file nel direttorio corrente:

```
#include ".."
```

# Organizzazione dei programmi C

L'organizzazione standard è la seguente:

- Le definizioni vengono poste in file detti header che hanno estensione **.h**
- Le implementazioni vengono poste in file che hanno estensione **.c**

Quando si vuole usare in un programma **P** dati e/o funzioni memorizzate altrove, si includono nel file che contiene **P** le loro definizioni (contenute nei file header) e si rendono così visibili al programma **P**. Questo permette la compilazione separata.

Il linker ha poi il compito di associare nella maniera corretta le definizioni alle invocazioni.

## Struttura di un file header

```
// File nomefile.h  
#ifndef NOMEFILE_H  
#define NOMEFILE_H
```

```
<sequenza di definizioni di costanti>  
<sequenza di definizioni di tipi>  
<sequenza di definizioni di variabili>  
<sequenza di definizioni di funzioni>
```

```
#endif
```

L'identificazione **NOMEFILE\_H** è solo un nome che associamo alle definizioni contenute nel file e, insieme alle direttive condizionali, serve a impedire che qualche oggetto sia definito più di una volta.

# Organizzazione di programmi su più file

```
// operazioni.h
#ifndef OPERAZIONI_H
#define OPERAZIONI_H

...
/* definizione di f */
int f(char);
...
#endif
```

```
// operazioni.c
#include "operazioni.h"

...
/* implementazione di f */
int f(char a) {
...
}
```

```
// main.c
#include <iostream.h>
#include "operazioni.h"

...
main() {
...
a=f(b); /* uso di f */
...
}
```

# Puntatori

Permettono la gestione di strutture dinamiche (create e distrutte sotto il controllo dell'utente).

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

**Dichiarazione:** in C un puntatore si dichiara antepoendo al nome di una variabile il simbolo `*`.

```
int *i;  
char *c;  
float *n;
```

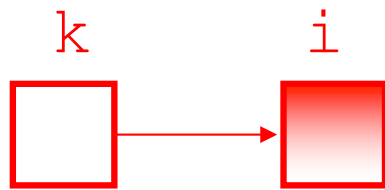
## Operazioni sui Puntatori

- l'operatore **&** applicato ad una variabile restituisce il puntatore ad essa;
- l'operatore **\*** applicato a un puntatore restituisce la variabile puntata

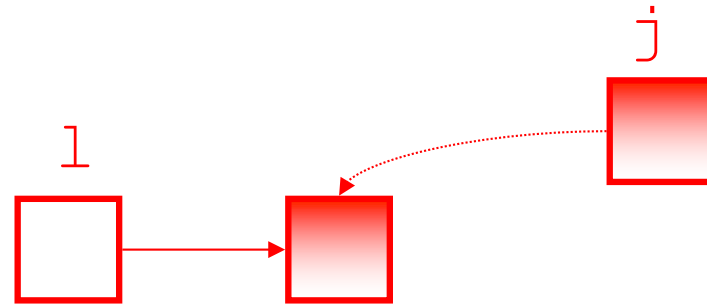
```
int i,j;  
int *k,*l;  
...  
k = &i; /* k punta alla variabile i */  
*l = j; /* nella variabile puntata da l  
        va il contenuto di j */  
*k = *l; /* nella variabile puntata da k  
        va il contenuto della variabile  
        puntata da l */  
k = l; /* k e l puntano alla stessa  
        variabile */
```



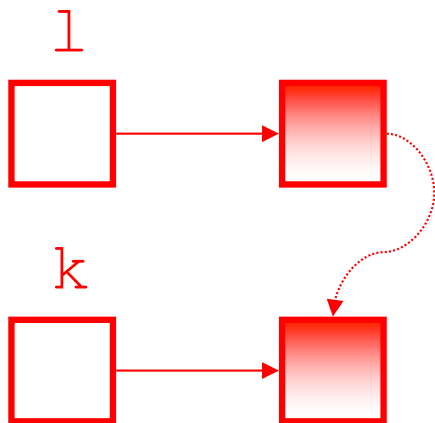
# Operazioni sui Puntatori



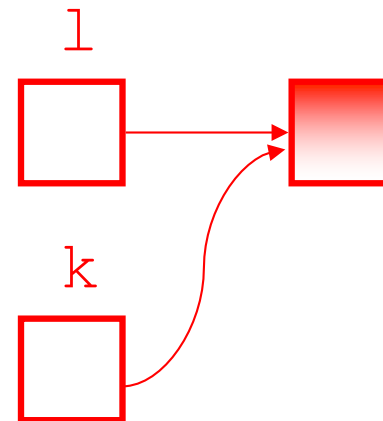
`k = &i;`



`*l = j;`



`*k = *l;`



`k = l;`

## Come ovviare ai limiti delle funzioni

**Procedure:** le definiamo come funzioni che restituiscono il tipo **void**

```
void PariDispari(int i) {  
    if ((i%2) == 0) printf("%d è un numero pari",i);  
    else printf("%d è un numero dispari",i);  
}
```

**Passaggi per riferimento:** non passiamo la variabile da modificare ma il suo puntatore.

```
void SommaProd(int i, int j, int *s, int *p) {  
    *s = i+j;  
    *p = i*j;  
}
```

# Array

Un **array** è una sequenza di elementi omogenei.

Un array viene dichiarato in C scrivendo, nell'ordine, il tipo degli elementi, il nome dell'array, e le sue dimensioni.

```
int c[12]; /*vettore*/
```

# Array

Un **vettore** è un gruppo di posizioni (o locazioni) di memoria correlate dal fatto che tutte hanno lo stesso nome e tipo di dato.

**Nome del  
puntatore (da  
notare che tutti  
gli elementi del  
puntatore hanno  
lo stesso nome,  
“c”)**

c[0]	4
c[1]	5
c[2]	0
c[3]	76
c[4]	7
c[5]	98
c[6]	6
c[7]	1
c[8]	2
c[9]	11
c[10]	90
c[11]	12

**Numero di  
posizione  
dell'elemento  
all'interno del  
vettore c**

## Regole generali su gli Array

c[0]	4
c[1]	5
c[2]	0
c[3]	76
c[4]	7
c[5]	98
c[6]	6
c[7]	1
c[8]	2
c[9]	11
c[10]	90
c[11]	12

- Il primo elemento di ogni vettore è ***l'elemento zero***.
- Il numero di posizione contenuto all'interno delle parentesi quadre è detto ***indice***.
- non c'è **controllo** sull'accesso a elementi **non esistenti**.

## Dichiarazione e inizializzazione dei vettori

I vettori occupano dello spazio in memoria. Il programmatore specificherà il tipo di ogni elemento e il numero di quelli richiesti da ognuno dei vettori, così che il computer possa riservare l'appropriata quantità di memoria.

**< tipo > < nome dell'array > [ dimensione ]**

**Uso: < nome dell'array > [ indice ]**

**Oss.** E' importante notare la differenza tra “*sesto elemento del vettore*” e “*l'elemento sei del vettore*”

```
c[5];  /* il sesto elemento */
```

```
c[6];  /* elemento sei del vettore */
```

# Dichiarazione e inizializzazione dei vettori

I vettori occupano dello spazio in memoria. Il programmatore specificherà il tipo di ogni elemento e il numero di quelli richiesti da ognuno dei vettori, così che il computer possa riservare l'appropriata quantità di memoria.

```
#define M 3
```

```
#define N 2
```

```
...
```

```
int c[M];
```

```
int l[N];
```

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

```
float r[] = {1.4, 3.2, 5.4 } /* viene allocato un vettore di tre  
                             elementi */
```

**È possibile inizializzare un array in fase di dichiarazione. In questo caso, per un vettore, la specifica delle dimensioni è opzionale.**

# I vettori multidimensionali

E' possibile definire strutture dati complesse: array multidimensionali, che si muovono su più indici.

< tipo > < nome dell'array > [ ] [ ] ... [ ]

```
int c[2][2];
```

```
int a[][] = { {1, 2}, {7, 12} };
```

Riga 0

Riga 1

Colonna 0   Colonna 1

<code>a[0][0]</code>	<code>a[0][1]</code>
<code>a[1][0]</code>	<code>a[1][1]</code>



1	2
7	12



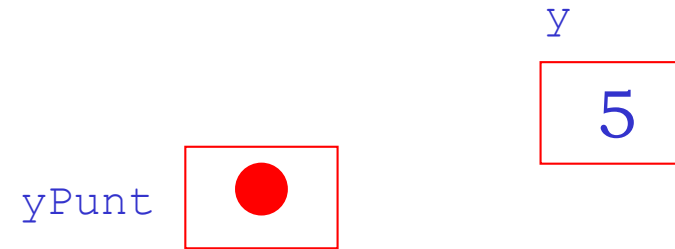
# Linguaggio C: Puntatori

- ✓ I puntatori sono il segreto della **potenza e la flessibilità** del C, perchè:
  - sono l'unico modo per effettuare alcune operazioni;
  - servono a produrre codici sorgenti compatti ed efficienti, anche se a volte difficili da leggere.
- ✓ In compenso, **la maggior parte degli errori** che i programmatori commettono in linguaggio C sono legati ai puntatori.
- ✓ In C ogni variabile è caratterizzata da due valori:
  - un **indirizzo della locazione di memoria** in cui sta la variabile,
  - ed il **valore contenuto** in quella locazione di memoria, che è il valore della variabile.
- ✓ Un **puntatore** e' un tipo di dato, è una **variabile** che contiene **l'indirizzo in memoria di un'altra variabile**, cioè un numero che indica in quale cella di memoria comincia la variabile puntata.

# Dinamica dei puntatori

```
int y=5;
```

```
int *yPunt;
```

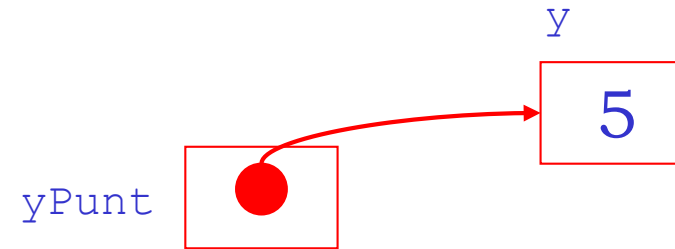


## Memoria



# Dinamica dei puntatori

```
int y=5;  
int *yPunt;  
yPunt = &y;
```



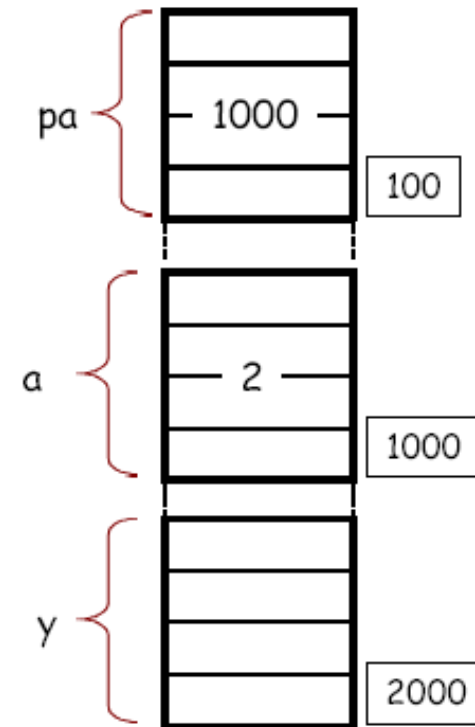
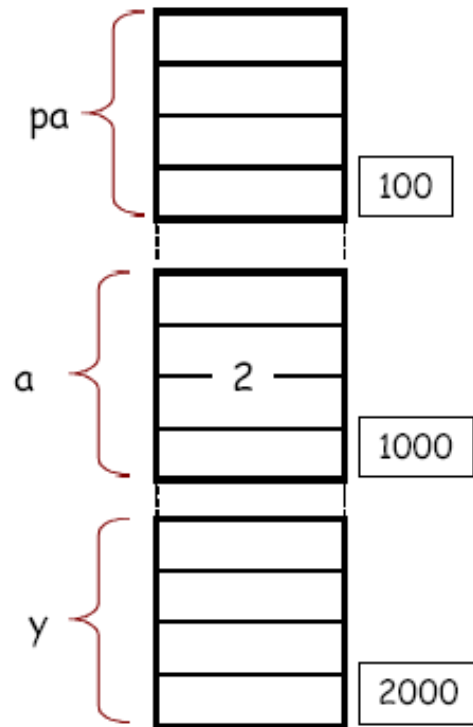
## Memoria



# Linguaggio C: Puntatori

```
int a=2, y;  
int *pa;
```

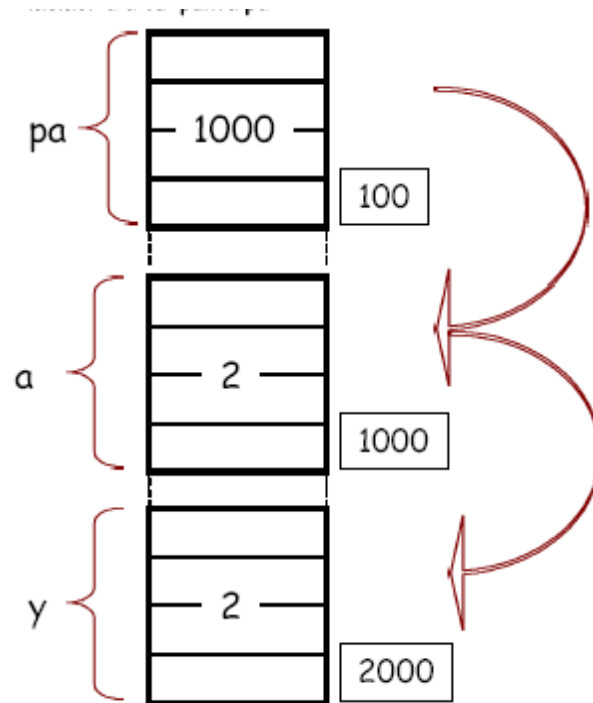
```
pa=&a;
```



# Linguaggio C: Puntatori

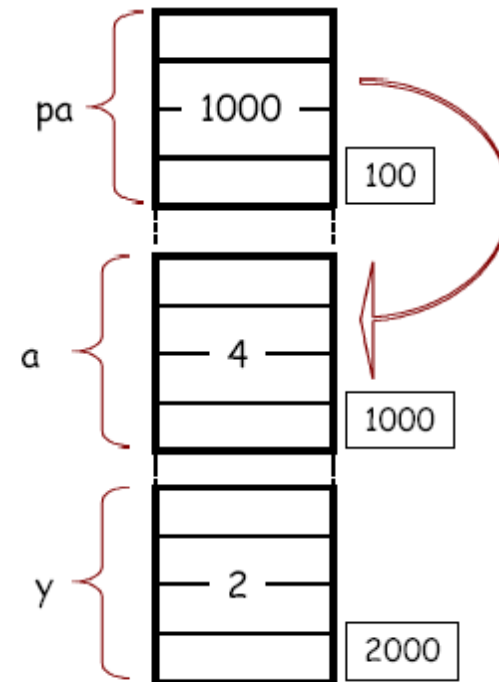
`y=*pa;`

Assegna ad y il contenuto della locazione di memoria a cui punta pa



`*pa=4;`

Assegna alla cella di memoria a cui punta pa il valore 4



# Linguaggio C: Puntatori

```
int Value(int n) {  
    return n*n*n; /* eleva al cubo la variabile locale n */  
}
```


```
void Reference(int *nP) {  
    *nP = *nP * *nP * *nP; /* eleva al cubo la variabile  
                             puntata */  
}
```

# Linguaggio C: Puntatori

number

5

```
int number = 5;
printf("%d", Value(number));
```



```
int Value(int n) {
```

```
    return n*n*n;
```

```
}
```


n

indefinita

number

5

```
int number = 5;
printf("%d", Value(number));
```



```
int Value(int n) {
```

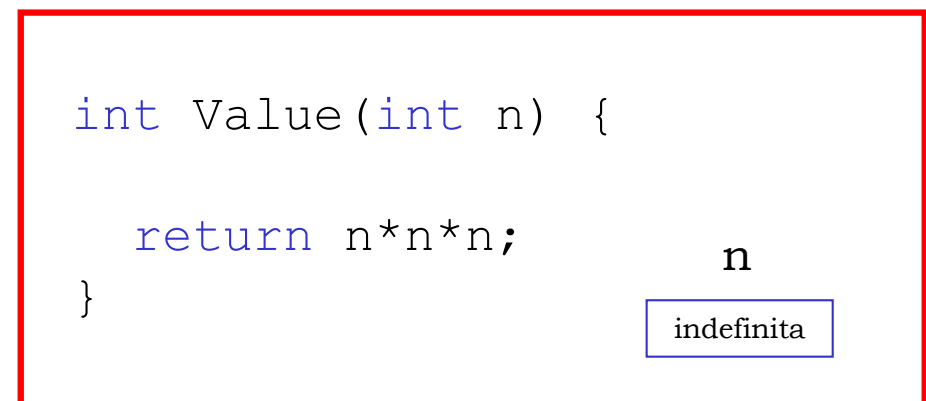
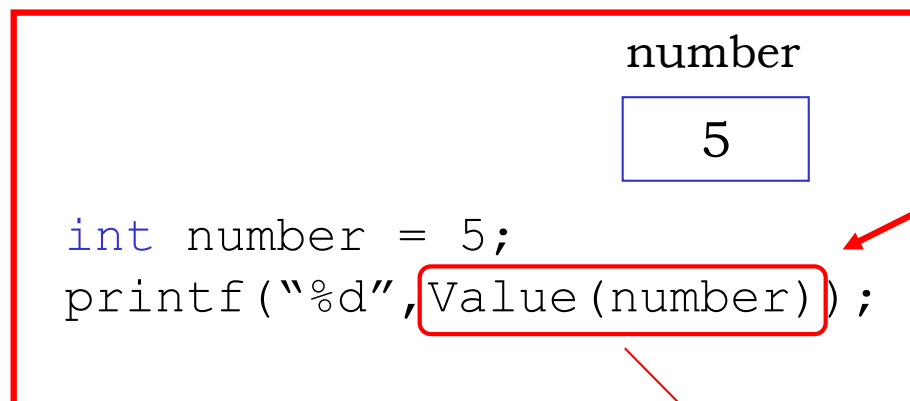
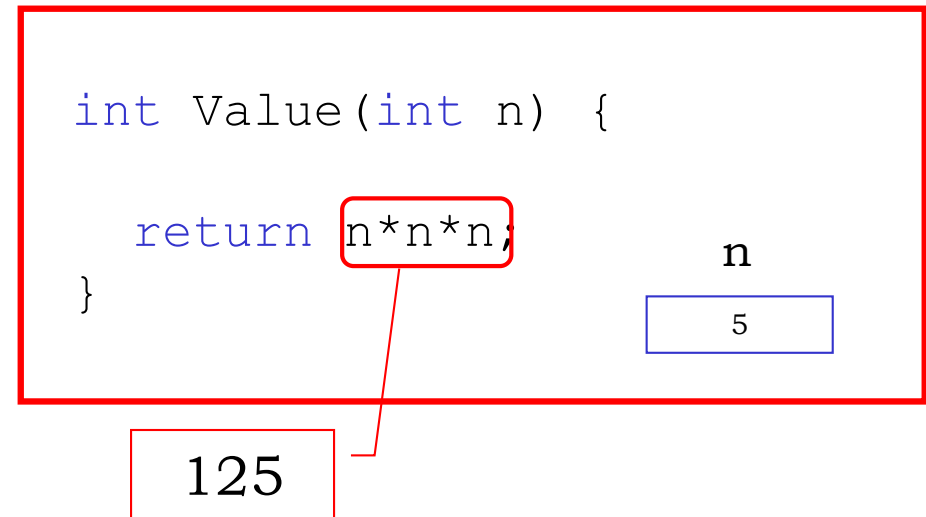
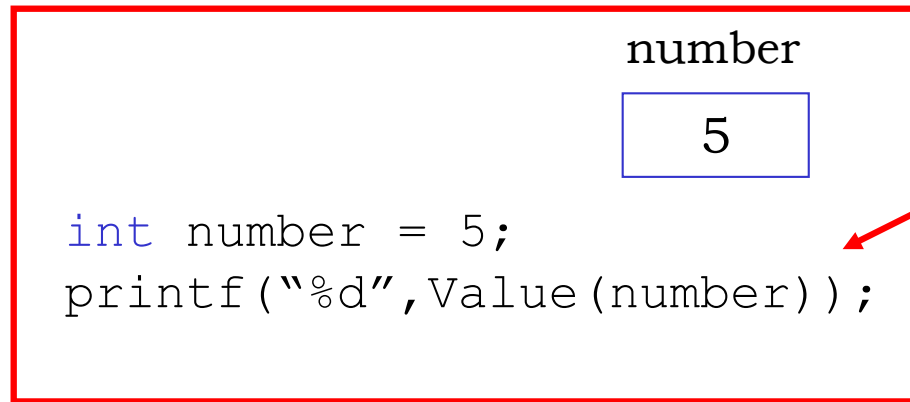
```
    return n*n*n;
```

```
}
```

n

5

# Linguaggio C: Puntatori





# Linguaggio C: Puntatori

number

5

```
int number = 5;  
printf("%d", Value(number));
```

```
int Value(int n) {
```

```
    return n*n*n;
```

```
}
```

n

indefinita

Stampa 125

# Linguaggio C: Puntatori

number

5

```
int number = 5; ←  
Reference(&number);
```

```
void Reference(int *nP) {  
    *nP = *nP * *nP * *nP;  
}
```

nP indefinita

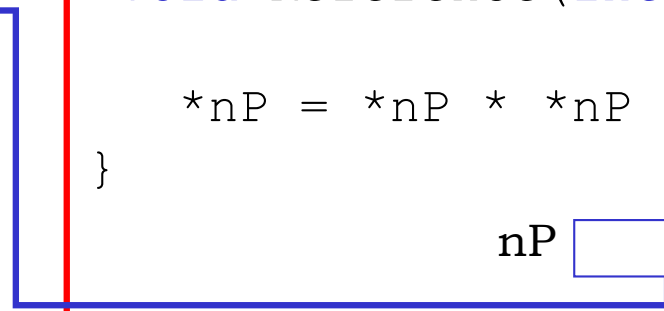
number

5

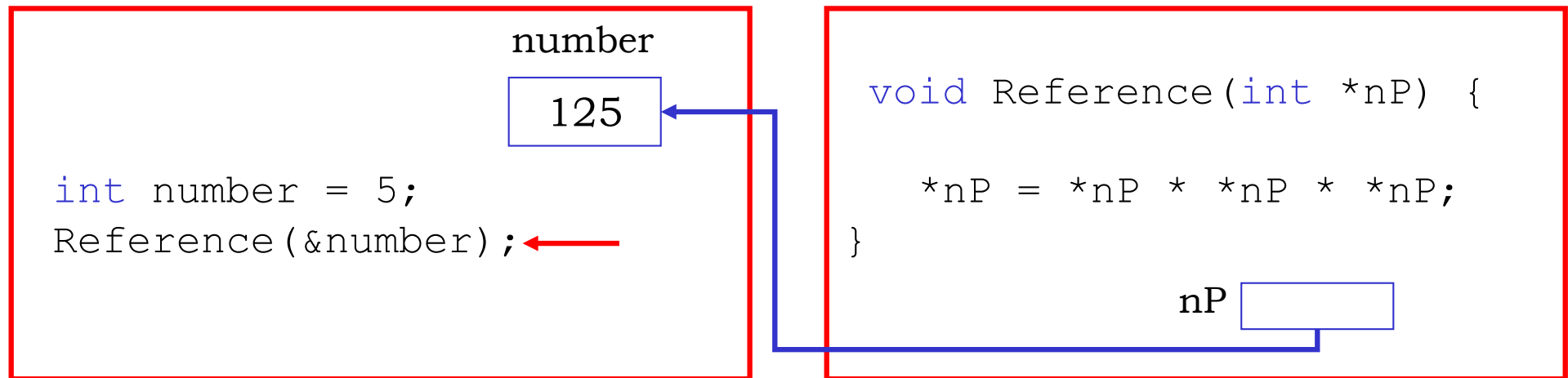
```
int number = 5;  
Reference(&number); ←
```

```
void Reference(int *nP) {  
    *nP = *nP * *nP * *nP;  
}
```

nP



# Linguaggio C: Puntatori



Number rimane con valore 125

# Puntatori: avvertenze

- ✓ Quando si usano i puntatori, è **molto facile fare confusione fra oggetti puntati e i loro puntatori**.
- ✓ Un puntatore è un indirizzo di memoria, mentre l'oggetto puntato è la zona di memoria che inizia con questo l'indirizzo, ed è grande quanto basta per contenere il tipo corrispondente.
- ✓ le variabili di tipo puntatore sono anche esse variabili, ossia zone di memoria.
- ✓ Ad esempio la differenza fra una variabile int e una variabile di tipo puntatore a intero è che la prima contiene un valore intero, mentre la seconda contiene un indirizzo, e in particolare l'indirizzo iniziale della zona di memoria associata a un intero.
- ✓ Quando un puntatore **viene dichiarato non punta a nulla**. Per poterlo utilizzare deve puntare a qualcosa.

Es. `int *p; *p=123; /*errore*/`

Usare:

```
int *p; int a;
```

```
p=&a; *p=123; /*corretto*/
```

# Aritmetica degli indirizzi

Si possono fare operazioni aritmetiche intere con i puntatori, ottenendo come risultato di far avanzare o riportare indietro il puntatore nella memoria, cioè di farlo puntare ad una locazione di memoria diversa.

Ovvero con i puntatori è possibile utilizzare due operatori aritmetici + e - , ed ovviamente anche ++ e --.

Il risultato numerico di un'operazione aritmetica su un puntatore è diverso a seconda del tipo di puntatore, o meglio a seconda delle dimensioni del tipo di dato a cui il puntatore punta.

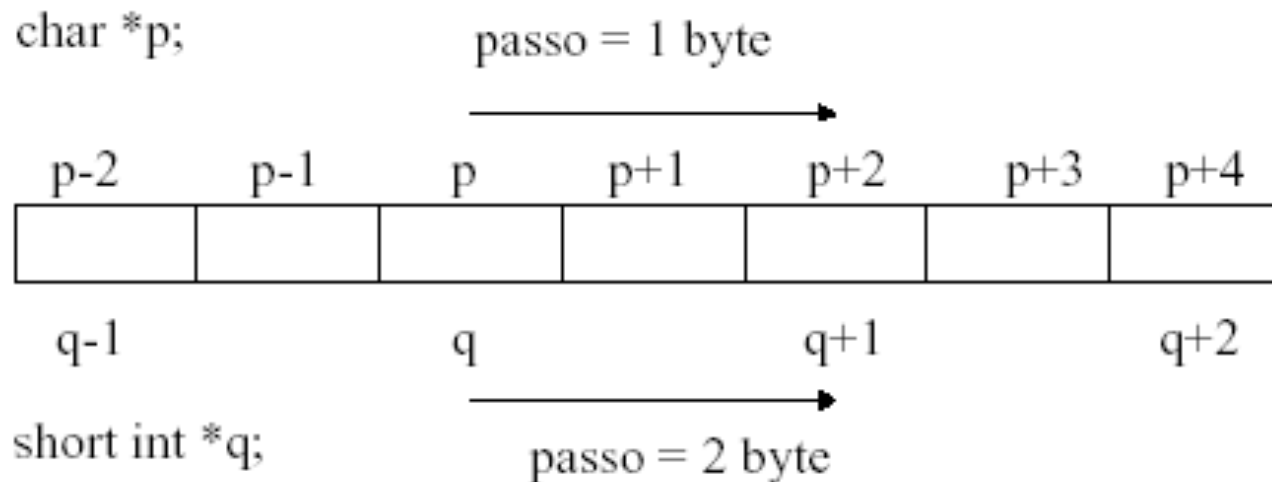
**il compilatore interpreta diversamente la stessa istruzione p++ a seconda del tipo di dato**, in modo da ottenere il comportamento seguente:

**Sommare un'unità ad un puntatore significa spostare in avanti in memoria il puntatore di un numero di byte corrispondenti alle dimensioni del dato puntato dal puntatore.**

# Aritmetica degli indirizzi

se  $p$  è un puntatore di tipo puntatore a char, **char \*p**, poichè il char ha dimensione 1, l'istruzione  $p++$  aumenta effettivamente di un'unità il valore del puntatore  $p$ , che punterà al successivo byte.

Invece se  $p$  è un puntatore di tipo puntatore a short int, **short int \*p**, poiché lo short int ha dimensione 2 byte, l'istruzione  $p++$  aumenterà effettivamente di 2 il valore del puntatore  $p$ , che punterà allo short int successivo a quello attuale.



# Array e Puntatori

Esiste un legame forte tra array e puntatori dovuta al fatto che:

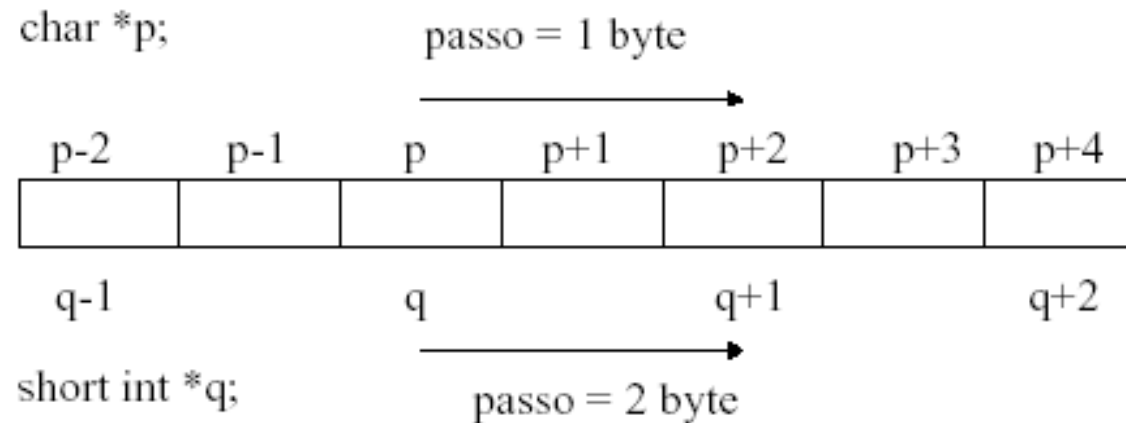
- gli elementi di un array vengono allocati in locazioni contigue della memoria principale;
- se si incrementa di uno un puntatore **p** a un tipo **T**, il valore di **p** viene incrementato di una quantità di byte pari alle dimensioni di **T** (aritmetica dei puntatori).

## **Regole che legano gli array con i puntatori:**

- il nome di un array coincide con l'indirizzo della prima componente del vettore;
- il puntatore ad un elemento dell'array si ottiene incrementando di uno il puntatore all'elemento precedente.

# Array e Puntatori

Se  $p$  è un puntatore ad un certo tipo (tipo  $*p$ ;) e contiene un certo valore **addr**, ovvero punta ad un certo indirizzo **addr**, l'espressione  $p[k]$  accede all'area di memoria che parte dal byte **addr+k**





# Array e Puntatori

Quindi, avendo definito

```
int a[5];
```

abbiamo che:

`a` e `&a[0]` sono la stessa cosa,

`*a` e `a[0]` sono la stessa cosa,

`*(a + 3)` e `a[3]` sono la stessa cosa

`(a + 2)` e `&a[2]` sono la stessa cosa.

# Array e Puntatori

Per i vettori (gli array monodimensionali di dati di tipo tipo) l'accesso ai dati avviene secondo queste modalità `vet[k]`, perchè in C, il nome di un array è TRATTATO dal compilatore come un puntatore *COSTANTE* alla prima locazione di memoria dell'array stesso .

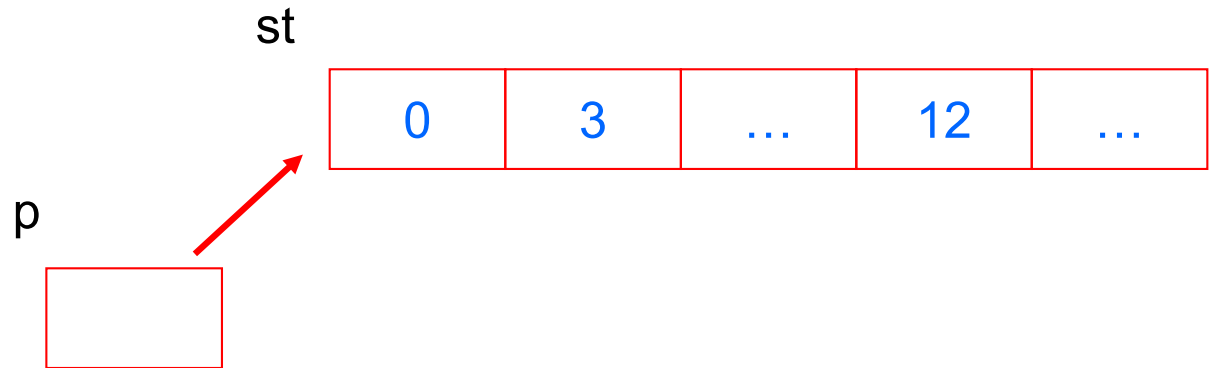
A differenza dei vettori però, l'area di memoria a cui il puntatore punta non viene allocata staticamente ma può essere allocata dinamicamente mediante alcune funzioni che richiedono al sistema operativo di riservare memoria e restituire un indirizzo a quell'area di memoria allocata, oppure può non essere allocata affatto.

# Array e Puntatori

```
char st[10];
```

```
char *p;
```

```
p = st;
```



Con questo assegnamento viene assegnato al puntatore `p` l'indirizzo della prima locazione di memoria del vettore. Da questo momento in avanti potremo accedere ai dati del vettore sia tramite `str`, sia tramite `p` esattamente negli stessi modi,

- o tramite l'indicizzazione tra parentesi quadre,
- o tramite l'aritmetica dei puntatori.

`str[10]` `*(str+10)` `p[10]` `*(p+10)` sono tutti modi uguali per accedere alla 11-esima posizione del vettore `str` puntato anche da `p`.

# Array e Puntatori

```
char st[10];  
char s = 'a';  
char *p;  
p = &s;
```

**st = p;**

NO

**ERRORE: NON E'  
POSSIBILE  
REINDIRIZZARE IL  
PUNTATORE DI UN  
ARRAY**

st



0

3

...

12

...

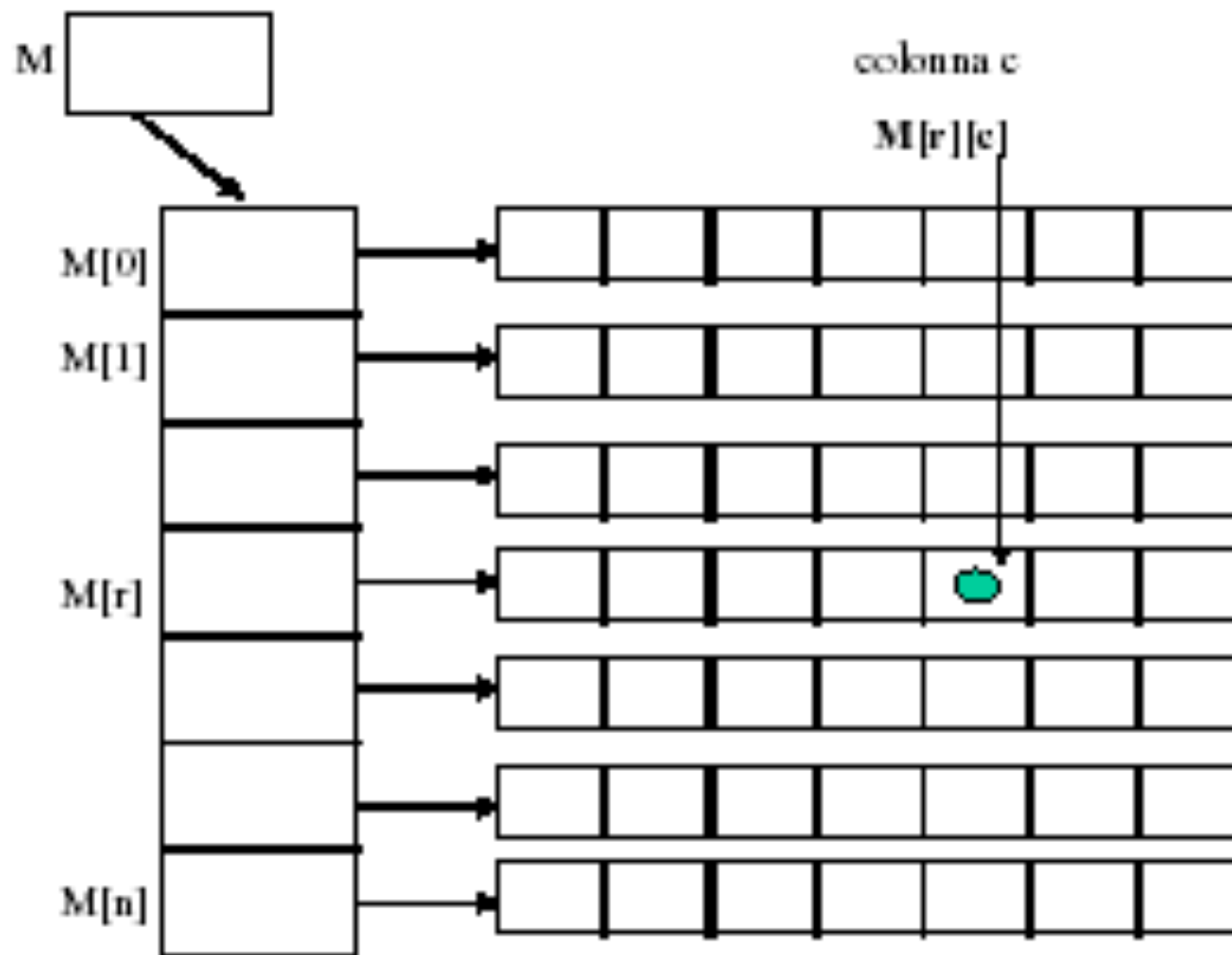
p



s

a

## I vettori multidimensionali



# Typedef

Il C permette di definire esplicitamente nomi nuovi per i tipi di dati, tramite la parola chiave **typedef**. L'uso di typedef consente di rendere il codice più leggibile.

Il formato dell'istruzione typedef è il seguente:

```
typedef < tipo > < nuovo_nome_tipo > ;
```

in questo modo assegnamo al tipo **tipo** il nuovo nome **nuovo\_nome\_tipo**.

Da questo momento in avanti potremo riferirci al tipo di dato **tipo** sia con il nome **tipo** sia con il nome **nuovo\_nome\_tipo**.

Es:

```
typedef int intero;  
intero i; /* definisce una variabile i di tipo int. */
```

## Struct (record)

Le *strutture* sono collezioni di variabili correlate sotto un unico nome. Le strutture possono contenere **variabili di diversi tipi** di dato (**contrariamente ai vettori**, che contengono soltanto elementi dello stesso tipo di dato)

Le strutture sono un *tipo di dato derivato* (sono cioè costruite usando oggetti di altri tipi).

## Struct (record)

In C i record si definiscono

*/\* definizione del tipo persona \*/*

**struct** persona {

**stringa** nome;

**stringa** cognome;

**stringa** CF;

**int** eta;

}

...

*/\* dichiarazione di variabili di tipo persona \*/*

**struct** persona pino, gino, nina;

...

La parola chiave **struct** introduce la definizione della struttura, e l'identificatore *persona* è la **structure tag** (l'etichetta della struttura)

Le variabili dichiarate all'interno delle parentesi graffe della definizione sono i *campi* della struttura.



## Struct (record)

In C i record si definiscono mediante il costrutto **struct**.

```
/* definizione del record persona */
```

```
struct persona {
```

```
stringa nome;
```

```
stringa cognome;
```

```
stringa CF;
```

```
int eta;
```

```
}
```

```
...
```

```
/* dichiarazione di variabili di tipo persona */
```

```
struct persona pino, gino, nina;
```

```
...
```

Per dichiarare le variabili di quel *tipo struttura* bisogna utilizzare l'etichetta della struttura con anteposta la parola chiave **struct**

## Struct (record)

Per accedere a un campo del record si usa l'operatore punto.

```
struct <nome_record> {  
    <tipo> nome1;  
    <tipo> nome2;  
    ...  
    <tipo> nome3;  
}
```

```
struct <nome_record> st;  
...  
st.nome2;  
...
```

## Struct (record)

Se abbiamo un puntatore al record possiamo accedere ai suoi campi attraverso l'operatore ->.

```
struct <nome_record> {  
    <tipo> nome1;  
    <tipo> nome2;  
    ...  
    <tipo> nome3;  
}
```

```
struct <nome_record> *st;  
...  
st->nome2;  
...
```

## Struct (record)

...

```
struct persona tizio;
```

```
struct persona *p;
```

```
struct persona amici[10];
```

...

```
scanf("%s", tizio.nome);
```

```
tizio.eta++;
```

```
printf("%s", amici[3].cognome);
```

...

```
p = &tizio;
```

```
printf("%s", p->cognome);
```

...

## Definizioni alternative di record

Ci sono vari modi per definire record oltre quello visto.

```
/* dichiarazione di variabili record contestuale alla  
definizione del record corrispondente */
```

```
struct contocorrente {  
    int numero;  
    char cognome[20];  
    char indirizzo[30];  
    long int saldo;  
} uno, due, tre;
```

## Definizioni alternative di record

Ci sono vari modi per definire record oltre quello visto.

```
/* dichiarazione di variabili record senza  
definizione del record corrispondente */
```

```
struct {  
    int codice;  
    char descrizione[20];  
    float dato;  
} a,b,c;
```

## Definizioni alternative di record

Ci sono vari modi per definire record oltre quello visto.

*/\* definizione di tipi di record mediante il  
costrutto typedef \*/*

**typedef struct** {

char nome[20];

char cognome[20];

char CF[16];

int eta;

} persona;

...

persona maria, marco;

...

## Proprietà dei record

È permessa la nidificazione di record.

```
struct persona {
```

```
char nome[20];
```

```
char cognome[20];
```

```
{ struct {
```

```
    int giorno;
```

```
    int mese;
```

```
    int anno;
```

```
} datanascita;
```

```
char indirizzo[20];
```

```
} squadra[20];
```

```
...
```

```
printf("%d", squadra[2].datanascita.anno);
```


```
...
```




## Proprietà dei record

È permesso l'autoriferimento mediante i puntatori.

```
struct persona {  
    char nome[20];  
    char cognome[20];  
    struct persona *coniuge;  
} io;
```



```
struct elemento {  
    int info;  
    struct elemento *next;  
} lista;
```



## Inizializzare i record

```
struct card {  
    char face[20];  
    char suit[20];  
}
```

...

```
struct card a = {"Three", "Hearts"};
```

...

## Copia di record

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

```
typedef struct {  
    int a;  
    int b;  
    char c;  
} prova;
```

```
int main(int argc, const char * argv[]){  
    prova p1,p2;  
    p1.a = 1;  
    p1.b = 2;  
    p1.c = 'a';  
    p2 = p1;  
    printf("%d %d %c\n", p2.a, p2.b, p2.c);  
    return 0;  
}
```

Stamperà:

**1 2 a**

## Copia di record

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

```
typedef struct {
    int a;
    int b;
    char c[20];
} prova;
```

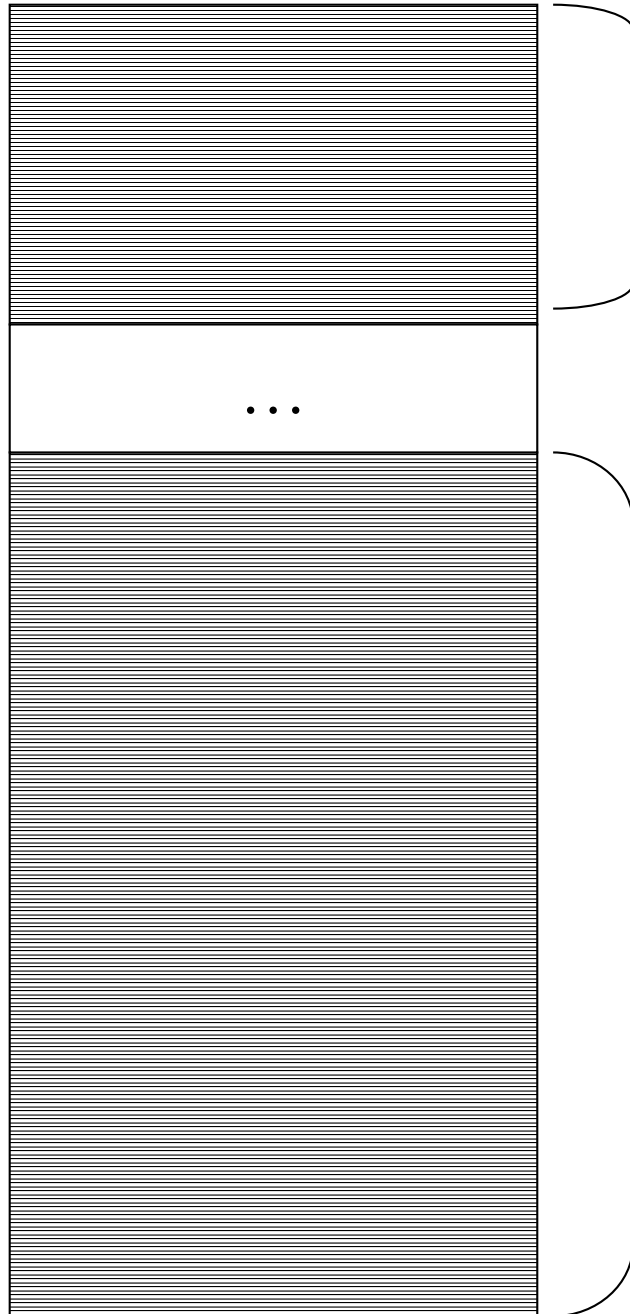
```
int main(int argc, const char * argv[]){
    prova p1,p2;
    p1.a = 1;
    p1.b = 2;
    strcpy(p1.c,"ciao");
    p2 = p1;
    printf("%d %d %s\n",p2.a,p2.b,p2.c);
    return 0;
}
```

Stamperà:  
**1 2 ciao**

# Gestione della memoria

Durante l'esecuzione di un programma C la memoria viene gestita in due maniere.

- **Gestione statica**: viene allocata dal sistema operativo un'area di memoria fissa per tutta l'esecuzione del programma.
- **Gestione dinamica**: vengono allocate due aree di memoria che vengono usate quando necessario e rese disponibili per successivi usi:
  - 1) lo **stack**: quando una funzione viene invocata vengono allocate automaticamente tutte le variabili locali e i parametri attuali sullo stack in un record di attivazione; successivamente, quando l'esecuzione della funzione termina, il record di attivazione viene cancellato e lo stack viene riportato nella situazione in cui era prima dell'invocazione;
  - 2) lo **heap**: la gestione viene lasciata al programmatore mediante creazione e distruzione dinamica di variabili (tramite puntatori).



**heap**

Gestito (dinamicamente)  
dall'utente

**stack**

Gestito (dinamicamente)  
dall'elaboratore

# Allocazione dinamica della memoria

L'allocazione dinamica della memoria consente una efficace gestione della memoria a tempo di esecuzione. Si usa generalmente per la gestione di dati di cui non è nota a priori la dimensione.

In ANSI C ci sono quattro funzioni per gestire dinamicamente

la memoria:

- **malloc**
- **calloc**
- **realloc**
- **free**

## Allocazione dinamica della memoria

Operativamente la gestione avviene mediante l'uso di puntatori e della costante simbolica predefinita **NULL** (che vale 0 ed è definita nella libreria **stdio.h**). La costante **NULL** serve a gestire la situazione in cui un puntatore non punta a nessuna locazione di memoria.



# Malloc

Alloca in maniera dinamica una zona di memoria della dimensione specificata in numero di byte. Restituisce un puntatore di tipo **void** all'area di memoria allocata. Se non c'è sufficiente quantità di memoria, restituisce **NULL**.

Viene sempre seguita da una operazione di "**casting**" per restituire un puntatore del tipo desiderato.

```
void *malloc( size_t size )
```

# Malloc

Si usa in genere insieme alla funzione C **sizeof** che restituisce la dimensione in numero di byte di una variabile o di un tipo.

```
size_t sizeof (<variabile o tipo>)
```

# Malloc

```
#include <stdio.h>
#include <stdlib.h>
...
typedef struct {
    char nome[20];
    char cognome[20];
    int telefono;
} persona;
....
/* definizione puntatori */
int *p, *q;
persona *tizio;
...
```

**/\* allocazione dinamica di memoria \*/**

`p=(int*)malloc(sizeof(int));`

`q=(int*)malloc(sizeof(*q));`

`tizio=(persona*)malloc(sizeof(persona));`

`if (tizio==NULL) printf("Out of memory");`

**Ho allocato una cella di memoria per contenere un intero**

**Ho allocato una cella di memoria per contenere un intero**

**Ho allocato una cella di memoria per contenere una struttura di tipo persona**

# Malloc

```
/* allocazione dinamica di memoria */  
  
p=(int*)malloc(sizeof(int));  
q=(int*)malloc(sizeof(*q));  
tizio=(persona*)malloc(sizeof(persona));  
if (tizio==NULL) printf("Out of memory");  
...  
  
/* uso delle variabili allocate */  
  
scanf("%d",p); scanf("%d",q);  
printf("%d\n",*p + *q);  
tizio->telefono=1733;  
printf("%d",tizio->telefono);  
...
```

## Calloc e Realloc

La **calloc** alloca in maniera dinamica una zona di memoria per memorizzare  $n$  oggetti della dimensione specificata.

La **realloc** rialloca uno spazio di memoria precedentemente allocato con una **calloc** o **malloc** (si usa in genere per modificare dimensione e/o tipo di un'area di memoria già allocata).

## Calloc e Realloc

In entrambi i casi viene restituito un puntatore di tipo **void** all'area di memoria (ri)allocata. Se non c'è sufficiente quantità di memoria, viene restituito **NULL**.

```
void *calloc( size_t n_elem, size_t elem_size )  
  
void *realloc( void *p, size_t size )
```

# Calloc e Realloc

...

```
typedef struct {  
    char nome[20];  
    char cognome[20];  
    int telefono;  
} persona;
```

```
typedef struct {  
    char nome[20];  
    char cognome[20];  
    int telefono;  
    int matricola;  
} studente;
```

....

```
persona *pp, *grp;  
studente *ps;
```



```
/* allocazione dinamica di una persona */
```

```
pp=(persona*)malloc(sizeof(persona));
```

```
pp->telefono=77;
```

```
/* allocazione dinamica di 10 persone */
```

```
grp=(persona*)calloc(10,sizeof(persona));
```

```
if (!grp) printf("Out of memory");
```

**Ho allocato 10 celle di memoria che conterranno strutture di tipo persona.**

```
/* riallocazione dinamica di pp */
```

```
ps=(studente*) realloc(pp,sizeof(studente));
```

```
ps->matricola=888;
```

```
printf("%d%d",ps->matricola,ps->telefono);
```

```
...
```

**Ho riallocato la cella di memoria che conteneva una struttura di tipo persona (puntata da pp).**

# Matrici in C

Allochiamo in modo dinamico una matrice: scriviamo un metodo “**leggi**” che allochi in memoria una matrice di interi M di n righe e m colonne (input) e la riempi facendoci passare gli elementi da tastiera.

```
int** M; int i, j;
```

```
M = (int**) calloc (n, sizeof (int*) );
```

```
for (i=0; i<n; i++)
```

```
    M[i] = (int*) calloc (m, sizeof (int) );
```

# Matrici in C

Allochiamo in modo dinamico una matrice: scriviamo un metodo “**leggi**” che allochi in memoria una matrice di interi M di n righe e m colonne (input) e la riempi facendoci passare gli elementi da tastiera.

```
int** leggi(int n, int m) {  
    int** M; int i, j;  
    M = (int**) calloc(n, sizeof(int*)) ;  
    for (i=0; i<n; i++)  
        M[i] = (int*) calloc(m, sizeof(int)) ;  
    for (i=0; i<n; i++)  
        for (j=0; j<m; j++)  
            scanf("%d", &M[i][j]) ;  
  
    return M;  
}
```

# Free

Rilascia una zona di memoria precedentemente allocata con una malloc, una calloc o una realloc.

**N.B.: andrebbe sempre usata prima della fine del programma su ogni variabile allocata dinamicamente.**

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

# Free

...

```
typedef struct {  
    char nome[20];  
    char cognome[20];  
    int telefono;  
} persona;
```

....

```
/* definizione puntatori */
```

```
int *p, *q;  
persona *pp, *grp;
```

...

# Free

```
/* allocazione dinamica di memoria */
```

```
p=(int*)malloc(sizeof(int));
```

```
q=(int*)malloc(sizeof(*q));
```

```
pp=(persona*)malloc(sizeof(persona));
```

```
grp=(persona*) calloc(10,sizeof(persona));
```

```
...
```

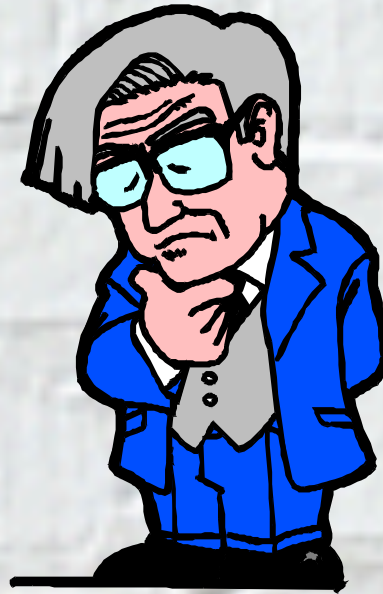
```
/* rilascio della memoria allocata */
```

```
free(p); free(q); free(pp); free(grp);
```

```
...
```

# Sistemi Operativi

## APPUNTI SUL LINGUAGGIO C



Richiami al linguaggio C