

Tipi di dato semplici

- ▶ Abbiamo visto nei primi esempi che il C tratta vari **tipi di dato**
⇒ le dichiarazioni associano variabili e costanti al corrispondente **tipo**
- ▶ Per **tipo di dato** si intende un insieme di **valori** e un insieme di **operazioni** che possono essere applicate ad essi.

Esempio:

I numeri interi $\{\dots, -2, -1, 0, 1, 2, \dots\}$ e le usuali operazioni aritmetiche (somma, sottrazione, ...)

- ▶ Ogni tipo di dato ha una propria **rappresentazione** in memoria (codifica binaria) che utilizza un certo numero di celle di memoria.
- ▶ Il meccanismo dei tipi ci consente di trattare le informazioni in maniera **astratta**, cioè prescindendo dalla sua rappresentazione **concreta**.

L'uso di variabili con tipo ha importanti conseguenze quali:

- ▶ per ogni variabile è possibile determinare a priori l'insieme dei valori ammissibili e l'insieme delle operazioni ad essa applicabili
- ▶ per ogni variabile è possibile determinare a priori la quantità di memoria necessaria per la sua rappresentazione
- ▶ è possibile rilevare a priori (a tempo di compilazione) errori nell'uso delle variabili all'interno di operazioni non lecite per il tipo corrispondente

Esempio: Nell'espressione $y + 3$ se la variabile y non è stata dichiarata di tipo numerico si ha un errore (almeno dal punto di vista concettuale) rilevabile a tempo di compilazione (cioè senza eseguire il programma).

Classificazione dei tipi

- ▶ **Tipi semplici:** consentono di rappresentare informazioni semplici
Esempio: una temperatura, una misura, una velocità, ecc.
- ▶ **Tipi strutturati:** consentono di rappresentare informazioni costituite dall'aggregazione di varie componenti
Esempio: una data, una matrice, una fattura, ecc.
- ▶ Un valore di un tipo semplice è logicamente **indivisibile**, mentre un valore di un tipo strutturato può essere **scomposto** nei valori delle sue componenti
Esempio: un valore di tipo **data** è costituito da tre valori (semplici)
- ▶ Il C mette a disposizione un insieme di tipi predefiniti (tipi **built-in**) e dei meccanismi per definire nuovi tipi (tipi **user-defined**)
Nota: con **T** identificatore di tipo, nel seguito indichiamo con **sizeof(T)** lo spazio (in byte) necessario per la memorizzazione di valori di tipo **T** (vedremo che **sizeof** è una funzione C).

Tipi semplici built-in

- ▶ interi
- ▶ reali
- ▶ caratteri

Per ciascun tipo consideriamo i seguenti **aspetti**:

1. intervallo di definizione (se applicabile)
2. notazione (sintassi) per le costanti
3. operatori
4. predicati (operatori di confronto)
5. formati di ingresso/uscita

Tipi interi: interi con segno

- ▶ 3 tipi:
 - `short`
 - `int`
 - `long`
- ▶ **Intervallo di definizione:** da -2^{n-1} a $2^{n-1}-1$, dove n dipende dal compilatore
- ▶ Vale:
 - `sizeof(short) ≤ sizeof(int) ≤ sizeof(long)`
 - `sizeof(short) ≥ 2` (ovvero, almeno 16 bit)
 - `sizeof(long) ≥ 4` (ovvero, almeno 32 bit)
- ▶ Compilatore `gcc`: `short`: 16 bit, `int`: 32 bit, `long`: 32 bit
- ▶ I valori limite sono contenuti nel file `limits.h`, che definisce le costanti: `SHRT_MIN`, `SHRT_MAX`, `INT_MIN`, `INT_MAX`, `LONG_MIN`, `LONG_MAX`

Notazione per le costanti: in decimale: 0, 10, -10, ...

- ▶ Per distinguere `long` (solo nel codice): `10L` (oppure `10l`, ma `l` sembra `1`).

Operatori: +, -, *, /, %, ==, !=, <, >, <=, >=

N.B.: l'operatore di uguaglianza si rappresenta con `==` (mentre `=` è utilizzato per il comando di assegnamento!)

Ingresso/uscita: tramite `printf` e `scanf`, con i seguenti specificatori di formato (dove `d` indica "decimale"):

`%hd` per `short`

`%d` per `int`

`%ld` per `long` (con `l` minuscola)

Tipi interi: interi senza segno

- ▶ 3 tipi:

 - `unsigned short`

 - `unsigned int`

 - `unsigned long`

- ▶ **Intervallo di definizione:** da 0 a 2^n-1 , dove n dipende dal compilatore.

Il numero n di bit è lo stesso dei corrispondenti interi con segno.

- ▶ Le costanti definite in `limits.h` sono:

 - `USHRT_MAX`, `UINT_MAX`, `ULONG_MAX` (n.b. il minimo è sempre 0)

Notazione per le costanti:

- ▶ decimale: come per interi con segno
 - ▶ esadecimale: `0xA`, `0x2F4B`, ...
 - ▶ ottale: `012`, `027513`, ...
- ▶ Nel codice si possono far seguire le cifre del numero dallo specificatore `u` (ad esempio `10u`).

Ingresso/uscita: tramite `printf` e `scanf`, con i seguenti specificatori di formato:

`%u` per numeri in decimale

`%o` per numeri in ottale

`%x` per numeri in esadecimale con cifre `0, ..., 9, a, ..., f`

`%X` per numeri in esadecimale con cifre `0, ..., 9, A, ..., F`

Per interi `short` si antepone `h`

`long` si antepone `l` (minuscola)

Operatori: tutte le operazioni vengono fatte modulo 2^n .

Caratteri

- ▶ Servono per rappresentare caratteri alfanumerici attraverso opportuni **codici**, tipicamente il codice **ASCII** (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange).
- ▶ Un codice associa ad ogni carattere un intero:

Esempio: Codice ASCII:

carattere:	'0'	...	'9'	','	':'	'<'
intero (in decimale):	48	...	57	58	59	60

carattere:	'a'	...	'z'	'{'	' '	'}'
intero (in decimale):	97	...	122	123	124	125

carattere:	'A'	...	'Z'	'['	'\'	']'
intero (in decimale):	65	...	90	91	92	93

- ▶ In C i caratteri possono essere **usati come gli interi** (un carattere coincide con il codice che lo rappresenta).

Intervallo di definizione: dipende dal compilatore

- ▶ Vale: `sizeof(char) ≤ sizeof(int)`

Tipicamente i caratteri sono rappresentati con 8 bit.

Operatori: sono gli stessi di `int` (operazioni effettuate utilizzando il codice del carattere).

Costanti: `'A'`, `'#'`, ...

Esempio:

```
char x, y, z;
```

```
x = 'A';
```

```
y = '\n';
```

```
z = '#';
```

Come non va usato il codice

- ▶ Confrontiamo:

```

char x, y, z;           char x, y, z;
x = 'A';               x = 65;   /* codice ASCII di 'A' */
y = '\n';              y = 10;  /* codice ASCII di '\n' */
z = '#';               z = 35;  /* codice ASCII di '#' */

```

- ▶ Non è sbagliato, però è **pessimo stile** di programmazione.
- ▶ Non è detto che il codice dei caratteri sia quello ASCII.
 ⇒ Il programma **non sarebbe portabile**.
- ▶ Vedremo presto un modo per sfruttare l'ordinamento tra caratteri molto utile.

Ingresso/uscita: tramite `printf` e `scanf`, con specificatore di formato `%c`

Attenzione: in ingresso non vengono saltati gli spazi bianchi e gli a capo

Esempio:

```
int i, j;
printf("Immetti due interi\n");
scanf("%d%d", &i, &j);
printf("%d %d\n", i, j);
```

Immetti due interi
> 18 25↵
18 25

```
int i, j;
char c;
printf("Immetti due interi\n");
scanf("%d%c%d", &i, &c, &j);
printf("%d %d %d\n", i, c, j);
```

Immetti due interi
> 18 25↵
18 32 25

- ▶ 32 è il codice ASCII del carattere ' ' (spazio)

- ▶ Funzioni per la stampa e la lettura di un singolo carattere:
`putchar(c)`; ... stampa il carattere memorizzato in `c`
`c = getchar()`; ... legge un carattere e lo assegna alla variabile `c`

Esempio:

```
char c;  
putchar('A');  
putchar('\n');  
c = getchar();  
putchar(c);
```

Tipi reali

I reali vengono rappresentati in virgola mobile (floating point).

- ▶ 3 tipi:

`float`

`double`

`long double`

- ▶ **Intervallo di definizione:**

	sizeof	cifre significative	min esp.	max esp.
<code>float</code>	4	6	-37	38
<code>double</code>	8	15	-307	308
<code>long double</code>	12	18	-4931	4932

- ▶ Le grandezze precedenti dipendono dal compilatore e sono definite nel file `float.h`.
- ▶ Deve comunque valere la relazione:

$$\text{sizeof(float)} \leq \text{sizeof(double)} \leq \text{sizeof(long double)}$$

Costanti: con punto decimale o notazione esponenziale

Esempio:

```
double x, y, z, w;  
x = 123.45;  
y = 0.0034;    /* oppure y = .0034 */  
z = 34.5e+20;  /* oppure z = 34.5E+20 */  
w = 5.3e-12;
```

► Nei programmi, per denotare una costante di tipo

► `float`, si può aggiungere `f` o `F` finale

Esempio: `float x = 2.3e5f;`

► `long double`, si può aggiungere `L` o `l` finale

Esempio: `long double x = 2.34567e520L;`

Operatori: come per gli interi (tranne `"%"`)

Ingresso/uscita: tramite `printf` e `scanf`, con diversi specificatori di formato

Output con `printf` (per float):

- ▶ `%f ...` notazione in virgola fissa
`%8.3f ...` 8 cifre complessive, di cui 3 cifre decimali

Esempio:

```
float x = 123.45;
printf("|%f| |%8.3f| |%-8.3f|\n", x, x, x);
```

```
|123.449997| | 123.450| |123.450 |
```

- ▶ `%e` (oppure `%E`) ... notazione esponenziale
`%10.3e ...` 10 cifre complessive, di cui 3 cifre decimali

Esempio:

```
double x = 123.45;
printf("|%e| |%10.3e| |%-10.3e|\n", x, x, x);
```

```
|1.234500e+02| | 1.234e+02| |1.234e+02 |
```

Input con scanf (per float):

si può usare indifferentemente %f o %e.

Riassunto degli specificatori di formato per i tipi reali:

	float	double	long double
printf	%f, %e	%f, %e	%Lf, %Le
scanf	%f, %e	%lf, %le	%Lf, %Le

Conversioni di tipo

Situazioni in cui si hanno conversioni di tipo

- ▶ quando in un'espressione compaiono operandi di tipo diverso
- ▶ durante un'assegnamento $x = y$, quando il tipo di y è diverso da quello di x
- ▶ esplicitamente, tramite l'operatore di **cast**
- ▶ nel passaggio dei parametri a funzione (più avanti)
- ▶ attraverso il valore di ritorno di una funzione (più avanti)

Una conversione può o meno coinvolgere un **cambiamento nella rappresentazione** del valore.

da **short** a **long** (dimensioni diverse)

da **int** a **float** (anche se stessa dimensione)

Conversioni implicite tra operandi di tipo diverso nelle espressioni

Quando un'espressione del tipo $x \text{ op } y$ coinvolge operandi di tipo diverso, avviene una conversione implicita secondo le seguenti regole:

1. ogni valore di tipo `char` o `short` viene convertito in `int`
2. se dopo il passo 1. l'espressione è ancora eterogenea si converte l'operando di tipo inferiore facendolo divenire di tipo superiore secondo la seguente gerarchia:

`int` → `long` → `float` → `double` → `long double`

Esempio: `int x; double y;`

Nel calcolo di $(x+y)$:

1. `x` viene convertito in `double`
2. viene effettuata la somma tra valori di tipo `double`
3. il risultato è di tipo `double`

Conversioni nell' assegnamento

Si ha in `x = exp` quando i tipi di `x` e `exp` non coincidono.

- ▶ La conversione avviene **sempre** a favore del tipo della variabile a sinistra:

se si tratta di una **promozione** non si ha perdita di informazione

se si ha una **retrocessione** si può avere perdita di informazione

Esempio:

```
int i;  
float x = 2.3, y = 4.5;  
i = x + y;  
printf("%d", i); /* stampa 6 */
```

- ▶ Se la conversione non è possibile si ha errore.

Conversioni esplicite (operatore di `cast`)

Sintassi: `(tipo) espressione`

- Converte il valore di `espressione` nel corrispondente valore del `tipo` specificato.

Esempio:

```
int somma, n;
float media;
...
media = somma / n;           /* divisione tra interi */
media = (float)somma / n;   /* divisione tra reali */
```

- L'operatore di cast "`(tipo)`" ha precedenza più alta degli operatori binari e associa da destra a sinistra. Dunque

`(float) somma / n`

equivale a

`((float) somma) / n`

Input/output

- ▶ Come già detto, input e output non sono parte integrante del C
- ▶ L'interazione con l'ambiente è demandato alla libreria standard
⇒ un insieme di funzioni a uso dei programmi C
- ▶ La libreria `stdio.h` implementa un semplice **modello** di ingresso e uscita di dati testuali
- ▶ un testo è trattato come un successione (**stream**) di caratteri, ovvero
⇒ una sequenza di caratteri organizzata in righe, ciascuna terminata da “\n”
- ▶ al momento dell'esecuzione, al programma vengono connessi automaticamente 3 stream:
 - ▶ **standard input**: di solito la tastiera
 - ▶ **standard output**: di solito lo schermo
 - ▶ **standard error**: di solito lo schermo

Input/output (cont.)

- ▶ Compito della libreria è fare in modo che tutto il trattamento dei dati in ingresso e uscita si conformi a questo modello
 - ⇒ il programmatore non si deve preoccupare di come ciò sia effettivamente realizzato
- ▶ Ogni volta che si effettua una operazione di **lettura** attraverso `getchar` viene acquisito il **prossimo** carattere dallo standard input e viene restituito il suo valore
(analogamente per `scanf` che comporta l'acquisizione di uno o più caratteri a seconda delle specifiche di formato presenti . . .)
- ▶ Ogni volta che si effettua una operazione di scrittura (attraverso `putchar` o `printf`) tutti i valori coinvolti vengono convertiti in sequenze di caratteri e quest'ultime vengono accodate allo standard output.
- ▶ Tipicamente il sistema operativo consente di reindirizzare gli stream standard, ad esempio su uno o più file.

Formattazione dell'output con `printf`

- ▶ Riepilogo specificatori di formato principali:
 - ▶ interi: `%d`, `%o`, `%u`, `%x`, `%X`
per `short`: si antepone `h`
per `long`: si antepone `l` (minuscola)
 - ▶ reali: `%e`, `%f`, `%g`
per `double`: non si antepone nulla
per `long double`: si antepone `L`
 - ▶ caratteri: `%c`
 - ▶ stringhe: `%s` (le vedremo più avanti)
 - ▶ puntatori: `%p` (li vedremo più avanti)
- ▶ Flag: messi subito dopo il `"%"`
 - ▶ `"-"`: allinea a sinistra
 - ▶ altri flag (non ci interessano)
- ▶ Sequenze di escape: `\%`, `\'`, `\"`, `\\`, `\a`, `\b`, `\n`, `\t`, ...

Formattazione dell'input con `scanf`

- ▶ Specificatori di formato: come per l'output, tranne che per i reali
 - ▶ `double`: si antepone `l`
 - ▶ `long double`: si antepone `L`
- ▶ **Soppressione dell'input:** mettendo `*` subito dopo `%`
Non ci deve essere un argomento corrispondente allo specificatore di formato.

Esempio: Lettura di una data in formato `gg/mm/aaaa` oppure `gg-mm-aaaa`.

```
int g, m, a;  
scanf("%d%*c%d%*c%d%*c", &g, &m, &a);
```

Tipi di dato strutturati: Array

- ▶ I tipi di dato visti finora sono tutti semplici: `int`, `char`, `float`, ...
- ▶ ma i dati manipolati nelle applicazioni reali sono spesso complessi (o **strutturati**)
- ▶ Gli **array** sono uno dei tipi di dato strutturati
 - ▶ sono composti da **elementi omogenei** (tutti dello stesso tipo)
 - ▶ ogni elemento è identificato all'interno dell'array da un **numero d'ordine** detto **indice** dell'elemento
 - ▶ il numero di elementi dell'array è detto **lunghezza** (o **dimensione**) dell'array
- ▶ Consentono di rappresentare tabelle, matrici, matrici n-dimensionali, ...

Array monodimensionali (o vettori)

- ▶ Supponiamo di dover rappresentare e manipolare la classifica di un campionato cui partecipano 16 squadre.
- ▶ È del tutto naturale pensare ad una **tabella**

Classifica

Squadra A	Squadra B	...	Squadra C
1° posto	2° posto		16° posto

che evolve con il procedere del campionato

Classifica

Squadra B	Squadra A	...	Squadra C
1° posto	2° posto		16° posto

Sintassi: dichiarazione di variabile di tipo vettore

```
tipo-elementi nome-array [lunghezza];
```

Esempio: `int vet[6];`

dichiara un vettore di 6 elementi, ciascuno di tipo intero.

- ▶ All'atto di questa dichiarazione vengono riservate (allocate) 6 locazioni di memoria **consecutive**, ciascuna contenente un intero. 6 è la **lunghezza** del vettore.
- ▶ La **lunghezza di un vettore deve essere costante** (nota a tempo di compilazione).
- ▶ Ogni elemento del vettore è una **variabile** identificata dal **nome** del vettore e da un **indice**

Sintassi: elemento di array `nome-array[espressione];`

Attenzione: `espressione` deve essere di tipo intero ed il suo valore deve essere compreso tra 0 a **lunghezza-1**.

▶ Esempio:

indice	elemento	variabile
0	?	vet[0]
1	?	vet[1]
2	?	vet[2]
3	?	vet[3]
4	?	vet[4]
5	?	vet[5]

- ▶ `vet[i]` è l'**elemento** del vettore `vet` di **indice** `i`.
Ogni elemento del vettore è una **variabile**.

```
int vet[6], a;
vet[0] = 15;
a = vet[0];
vet[1] = vet[0] + a;
printf("%d", vet[0] + vet[1]);
```

- ▶ `vet[0]`, `vet[1]`, ecc. sono variabili intere come tutte le altre e dunque possono stare a sinistra dell'assegnamento (es. `vet[0] = 15`), così come all'interno di espressioni (es. `vet[0] + a`).
- ▶ Come detto, l'indice del vettore è un'espressione.

```
index = 2;
vet[index+1] = 23;
```

Manipolazione di vettori

- ▶ avviene solitamente attraverso cicli **for**
- ▶ l'indice del ciclo varia in genere da **0** a **lunghezza-1**
- ▶ spesso conviene definire la lunghezza come una **costante** attraverso la direttiva **#define**

Esempio: Lettura e stampa di un vettore.

```
#include <stdio.h>
#define LUNG 5

main ()
{
int v[LUNG]; /* vettore di LUNG elementi, indicizzati da 0 a LUNG-1 */
int i;

for (i = 0; i < LUNG; i++) {
    printf("Inserisci l'elemento di indice %d: ", i);
    scanf("%d", &v[i]);
}
printf("Indice Elemento\n");
for (i = 0; i < LUNG; i++) {
    printf("%6d %8d\n", i, v[i]);
}
}
```

Inizializzazione di vettori

- ▶ Gli elementi del vettore possono essere inizializzati con **valori costanti** (valutabili a tempo di compilazione) contestualmente alla dichiarazione del vettore .

Esempio: `int n[4] = {11, 22, 33, 44};`

- ▶ l'inizializzazione deve essere contestuale alla dichiarazione

Esempio: `int n[4];
n = {11, 22, 33, 44};` \implies **errore!**

- ▶ se i valori iniziali sono meno degli elementi, i rimanenti vengono posti a 0

`int n[10] = {3};` azzera i rimanenti 9 elementi del vettore
`float af[5] = {0.0};` pone a 0.0 i 5 elementi
`int x[5] = {};` **errore!**

- ▶ se ci sono più inizializzatori di elementi, si ha un errore a tempo di compilazione

Esempio: `int v[2] = {1, 2, 3};` **errore!**

- ▶ se si mette una sequenza di valori iniziali, si può omettere la lunghezza (viene presa la lunghezza della sequenza)

Esempio: `int n[] = {1, 2, 3};` equivale a
`int n[3] = {1, 2, 3};`

- ▶ In C l'unica operazione possibile sugli array è l'**accesso** ai singoli elementi.
- ▶ Ad esempio, non si possono effettuare direttamente delle assegnazioni tra vettori.

Esempio:

```
int a[3] = {11, 22, 33};
```

```
int b[3];
```

```
b = a;
```

errore!

Esempi

- ▶ Calcolo della somma degli elementi di un vettore.

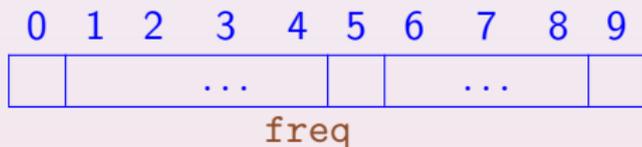
```
int a[10], i, somma = 0;
...
for (i = 0; i < 10; i++)
    somma += a[i];
printf("%d", somma);
```

- ▶ Leggere **N** interi e stampare i valori maggiori di un valore intero **y** letto in input.

```
#include <stdio.h>
#define N 4
main()  {
int ris[N];
int y, i;
printf("Inserire i %d valori:\n", N);
for (i = 0; i < N; i++) {
    printf("Inserire valore n.  %d:  ", i+1);
    scanf("%d", &ris[i]);    }
printf("Inserire il valore y:\n");
scanf("%d", &y);

printf("Stampa i valori maggiori di %d:\n", y);
for (i = 0; i < N; i++)
    if (ris[i] > y)
        printf("L'elemento %d:  %d e' maggiore di %d\n",
                i+1, ris[i], y);
}
```

- ▶ Leggere una sequenza di caratteri terminata dal carattere `\n` di fine linea e stampare le frequenze delle cifre da `'0'` a `'9'`.
- ▶ utilizziamo un vettore `freq` di 10 elementi nel quale memorizziamo le frequenze dei caratteri da `'0'` a `'9'`



`freq[0]` conta il numero di occorrenze di `'0'`

...

`freq[9]` conta il numero di occorrenze di `'9'`

- ▶ utilizziamo un ciclo per l'acquisizione dei caratteri in cui aggiorniamo una delle posizioni dell'array tutte le volte che il carattere letto è una cifra

```
int i; char ch;
int freq[10] = {0};
do {
    ch = getchar();
    switch (ch) {
        case '0': freq[0]++; break;
        case '1': freq[1]++; break;
        case '2': freq[2]++; break;
        case '3': freq[3]++; break;
        case '4': freq[4]++; break;
        case '5': freq[5]++; break;
        case '6': freq[6]++; break;
        case '7': freq[7]++; break;
        case '8': freq[8]++; break;
        case '9': freq[9]++; break;
    }
} while (ch != '\n');
printf("Le frequenze sono:\n");
for (i = 0; i < 10; i++)

    printf("Freq. di %d: %d\n", i, freq[i]);
```

- ▶ Nel ciclo **do-while**, il comando **switch** può essere rimpiazzato da un **if** come segue

```
if (ch >= '0' && ch <= '9')  
    freq[ch - '0']++;
```

Infatti:

- ▶ i codici dei caratteri da '0' a '9' sono consecutivi
- ▶ dato un carattere **ch**, l'espressione intera **ch - '0'** è la **distanza** del codice di **ch** dal codice del carattere '0'. In particolare:
 - ▶ '0' - '0' = 0
 - ▶ '1' - '0' = 1
 - ▶ ...
 - ▶ '9' - '0' = 9

- ▶ Leggere da tastiera i risultati (double) di 20 esperimenti. Stampare il numero d'ordine ed il valore degli esperimenti per i quali il risultato è minore del 50% della media.

```
#include <stdio.h>
#define DIM 20
main() {
    double ris[DIM], media;
    int i;
    /* inserimento dei valori */
    printf("Inserire i %d risultati dell'esperimento:\n", DIM);
    for (i = 0; i < DIM; i++) {
        printf("Inserire risultato n. %d: ", i);
        scanf("%g", &ris[i]); }
    /* calcolo della media */
    media = 0.0;
    for (i = 0; i < DIM; i++)
        media = media + ris[i];
    media = media/DIM;
    printf("Valore medio: %g\n", media);
    /* stampa dei valori minori di media*0.5 */
    printf("Stampa dei valori minori di media*0.5:\n");
    for (i = 0; i < DIM; i++)
        if (ris[i] < media * 0.5)
            printf("Risultato n. %d: %g\n", i, ris[i]); }
}
```

Array multidimensionali

Sintassi: dichiarazione

tipo-elementi nome-array [lung₁] [lung₂] ... [lung_n];

Esempio: `int mat[3][4];` \implies matrice 3 \times 4

- Per ogni dimensione i l'indice va da 0 a $lung_i-1$.

		colonne			
		0	1	2	3
righe	0	?	?	?	?
	1	?	?	?	?
	2	?	?	?	?

Esempio: `int marketing[10][5][12]`

(indici potrebbero rappresentare: prodotti, venditori, mesi dell'anno)

Accesso agli elementi di una matrice

```
int i, mat[3][4];
```

```
...
```

```
i = mat[0][0];      elemento di riga 0 e colonna 0 (primo elemento)
```

```
mat[2][3] = 28;     elemento di riga 2 e colonna 3 (ultimo elemento)
```

```
mat[2][1] = mat[0][0] * mat[1][3];
```

- ▶ Come per i vettori, l'unica operazione possibile sulle matrici è l'accesso agli elementi tramite l'operatore `[]`.

Esempio: Lettura e stampa di una matrice.

```
#include <stdio.h>
#define RIG 2
#define COL 3
main()
{
int mat[RIG][COL];
int i, j;
/* lettura matrice */
printf("Lettura matrice %d x %d;\n", RIG, COL);
for (i = 0; i < RIG; i++)
    for (j = 0; j < COL; j++)
        scanf("%d", &mat[i][j]);
/* stampa matrice */
printf("La matrice e':\n");
for (i = 0; i < RIG; i++) {
    for (j = 0; j < COL; j++)
        printf("%6d ", mat[i][j]);
    printf("\n");        } /* a capo dopo ogni riga */
}
```

Esempio: Programma che legge due matrici $M \times N$ (ad esempio 4×3) e calcola la matrice somma.

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        c[i][j] = a[i][j] + b[i][j];
```

Inizializzazione di matrici

```
int mat[2][3] = {{1,2,3}, {4,5,6}};
```

1	2	3
4	5	6

```
int mat[2][3] = {1,2,3,4,5,6};
```

```
int mat[2][3] = {{1,2,3}};
```

1	2	3
0	0	0

```
int mat[2][3] = {1,2,3};
```

```
int mat[2][3] = {{1}, {2,3}};
```

1	0	0
2	3	0

Esercizio

Programma che legge una matrice A ($M \times P$) ed una matrice B ($P \times N$) e calcola la matrice C prodotto di A e B

- ▶ La matrice C è di dimensione $M \times N$.
- ▶ Il generico elemento C_{ij} di C è dato da:

$$C_{ij} = \sum_{k=0}^{P-1} A_{ik} \cdot B_{kj}$$

Soluzione

```
#define M 3
#define P 4
#define N 2
int a[M][P], b[P][N], c[M][N];
...
/* calcolo prodotto */
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++) {
        c[i][j] = 0;
        for (k = 0; k < P; k++)
            c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
```

- ▶ Tutti gli elementi di **c** possono essere inizializzati a **0** al momento della dichiarazione:

```
int a[M][P], b[P][N], c[M][N] = {0};
...
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < P; k++)
            c[i][j] += a[i][k] * b[k][j];
```