Università degli Studi di Milano

Corso di Laurea in
Sicurezza dei Sistemi e delle Reti Informatiche

Lezione 7 – Tipi di dato built-in

FABIO SCOTTI

Laboratorio di programmazione per la sicurezza

Indice

1. RAPPRESENTAZIONE DELL'INFORMAZIONE IN C 3

2. I TIPI DI DATO BUILT-IN 4

3. RAPPRESENTAZIONE NUMERICA 5
   3.1 Interi con segno(int) 5
   3.2 Interi senza segno (unsigned) 6
   3.3 Tipi reali o floating point (float) 6
   3.4 Un esercizio commentato 7

4. RAPPRESENTAZIONE DEI CARATTERI 10

5. VETTORI 11

6. MATRICI 16

7. STRINGHE 21

1. Rappresentazione dell'informazione in C

Nei primi esempi di programmi che abbiamo presentato, abbiamo notato come il C permetta di definire e trattare vari *tipi di dato*, ad esempio interi e caratteri. La dichiarazione del tipo di dato è infatti presente in tutte le dichiarazione della variabili. In esse al nome di una variabile viene associato il suo tipo (ad esempio `int indice;`)

Per *tipo di dato* si intende un insieme di **valori** e un insieme di **operazioni** che possono essere applicati. Ad esempio i numeri interi e le usuali operazioni aritmetiche come la somma, sottrazione, etc.

Ogni tipo di dato ha una propria rappresentazione in memoria (codifica binaria) che utilizza un certo numero di celle di memoria. Tuttavia l'uso nei nostri programmi dei tipi di dato ci consente di trattare le informazioni in maniera astratta, cioè prescindendo dalla sua rappresentazione nella macchina. In altre parole se sommiamo due interi possiamo non preoccuparci di come devono essere sommati i bit che corrispondono ai due addendi in memoria, ma semplicemente indichiamo un segno di somma nella nostra istruzione in C (ad esempio scriviamo c=a+b;).

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 applicabili**
- per ogni variabile è possibile determinare a priori la **quantità di memoria necessaria**
- è possibile **rilevare a priori** (a tempo di compilazione) **errori nell'uso delle variabili** all'interno di operazioni non lecite per il tipo corrispondente

I tipi di dato si dividono in

- tipi semplici;
- tipi strutturati.

I tipi semplici consentono di rappresentare informazioni composte da un valore come ad esempio una temperatura, una velocità, ecc.

I tipi strutturati consentono di rappresentare informazioni costituite dall'aggregazione di varie componenti come ad esempio una data, una matrice, una cartella clinica, 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. Ad esempio: un valore di tipo data è costituito da tre valori (semplici).

2. I tipi di dato built-in

Il C mette a disposizione un insieme di tipi predefiniti (tipi built-in) e dei meccanismi per definire nuovi tipi (tipi user-defined).

I tipi semplici che il C mette a disposizione sono:

- Interi
- Reali
- Caratteri

Il tipo deve essere scelto tenendo conto delle esigenze di rappresentazione che deve avere la nostra variabile che intendiamo usare. Se per esempio dobbiamo dichiarare una variabile che deve assumere solo valori interi (come ad esempio un contatore di un ciclo) allora possiamo dichiarare una variabile `cont` come intero nel seguente modo `int cont;` se invece dobbiamo impiegare una variabile che può assumere dei valori reali (come ad esempio la diagonale di un generico rettangolo) allora impiegheremo un variabile reale `diag` dichiarandola nel seguente modo `float diag;`

Per comprendere meglio di ognuno di essi le loro peculiarità andremo ad analizzare nelle lezioni successive i seguenti aspetti:

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

In C è sempre possibile ottenere la occupazione in memoria di una variabile o di un tipo attraverso la funzione `sizeof( )` che restituisce lo spazio di memoria occupato in byte.

3. Rappresentazione numerica

Iniziamo la discussione dei tipi di dato built-in del C con i tipi di dato adatti a rappresentare numeri. Le principali differenze riguarderanno se sono tipi per numeri interi oppure con virgola e il numero di bit che sono utilizzati nella loro rappresentazione nell'elaboratore.

3.1 Interi con segno(int)

In C esistono 3 tipi di interi con segno:

- **short** (dichiarabile nei seguenti modi `short int`, `signed short`, `signed short int`)
- **int** (dichiarabile nei seguenti modi `signed int`, `signed`)
- **long** (dichiarabile nei seguenti modi `long int`, `signed long`, `signed long int`)

Il loro intervallo di definizione va da -2^{n-1} a $+2^{n-1}-1$, dove n dipende dal compilatore e rappresenta il numero di bit impiegati per la loro rappresentazione.

Analizzando la loro occupazione in memoria (direttamente legata al numero di bit con il quale il dato è memorizzato e quindi all'intervallo di rappresentazione), abbiamo che valgono le seguenti relazioni:

- sizeof(short) ≤ sizeof(int) ≤ sizeof(long)
- sizeof(short) ≥ 2 (ovvero, almeno 16 bit)
- sizeof(long) ≥ 4 (ovvero, almeno 32 bit)

Ricordo che la funzione `sizeof()` in C ritorna il numero di byte occupati da una variabile o da un tipo di dato che gli viene passato.

Nel caso del compilatore gcc (vedi lezioni precedenti) si ha la seguente situazione

- 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

Queste costanti possono ad esempio servire in un programma per controllare se il dato immesso da un utente cade nell'intervallo di rappresentazione. Se non lo fosse, possiamo segnalarlo all'utente al posto che far continuare l'esecuzione e generare quindi un probabile errore.

Se vogliamo dichiarare una costante scriveremo gli interi semplicemente in decimale, ad esempio 0, 10, -10, . . .

Gli operatori utilizzabili sono: +, -, *, /, %, ==, !=, <, >, <=, >=

Per quanto riguarda le funzioni di ingresso/uscita si usano le funzioni `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)

Ad esempio per stampare un intero che era stato definito come short potremmo scrivere:
`printf("%hd", contatore);`

3.2 Interi senza segno (unsigned)

In C esistono 3 tipi di interi senza segno:

- **unsigned short** (oppure: `unsigned short int`)
- **unsigned int**
- **unsigned long** (equivalente a `unsigned long int`)

Il loro intervallo di definizione va da 0 a $+2^n-1$, dove n dipende dal compilatore e rappresenta il numero di bit impiegati per la loro rappresentazione.

Le costanti definite che indicano gli intervalli di rappresentazione sono contenuti nel file limits.h e sono:

USHRT_MAX, UINT_MAX, ULONG_MAX (si noti che il minimo è sempre 0)

Se vogliamo dichiarare una costante scriveremo

- decimale: come per interi con segno
- esadecimale: 0xA, 0x2F4B, . . .

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` e per interi long si antepone `l` (minuscola)

3.3 Tipi reali o floating point (float)

In C i tipi reali vengono rappresentati mediante virgola e ne esistono 3 tipi:

- **float** (dimensione in memoria 4 byte)
- **double** (dimensione in memoria 8 byte)
- **long double** (dimensione in memoria 12 byte)

In particolare questa notazione si chiama in **virgola mobile** e verrà presentata approfonditamente in altri corsi.

Le grandezze e gli intervalli di questi tipi di dato sono scritte nel file ***float.h*** di libreria e *dipendono dal compilatore*. Comunque valgono le seguenti relazioni:

sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)

Se vogliamo dichiarare una costante scriveremo gli interi semplicemente in decimale USANDO SEMPRE IL PUNTO oppure usando la notazione scientifica.

Esempio:

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

Vediamo ora **l'output** dei variabili float mediante printf:

- Se usiamo %f stampiamo il numero in virgola fissa
- Se usiamo %8.3f stampiamo 8 cifre complessive, di cui 3 cifre decimali
- Se usiamo %e stampiamo in forma esponenziale

Esempio:

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

Otteniamo sul nostro output

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

Esempio

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

Otteniamo sul nostro output

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

Possiamo stampare anche in forma ottimizzata usando %g (oppure %G) che sceglie la forma più compatta fra la notazione esponenziale (%e) e quella di %f

Per quanto riguarda **l'input** con scanf nel caso dei float si può usare indifferentemente %f o %e.

3.4 Un esercizio commentato

Nell'esempio che segue vediamo come definire variabili per interi e vediamo la loro occupazione in memoria. Attraverso il meccanismo solito di definizione delle variabili definiamo 4 variabili di tipo carattere, 4 di tipo int, 4 di tipo long e 4 di tipo double.

Attraverso istruzioni del tipo `printf("la variabile i1 si trova in 0x %p (%d) \n", &i1 , &i1 );` andiamo a pubblicare l'indirizzo della variabile `i1` che è ottenibile scrivendo `&i1` e lo stampiamo in due modi: in formato esadecimale (%p nella printf) e decimale (%d nella printf)

N.B. nella printf scriviamo sempre 0x davanti ad un numero che verrà stampato in modo esadecimale per esplicitarlo all'utente.

```
// Occupazione di memoria dei tipi di dato
....  include
```

```
int main()
{
    char c1;
    char c2;
    char c3;
    char c4;

    int i1;
    int i2;
    int i3;
    int i4;

    long l1;
    long l2;
    long l3;
    long l4;

    double d1;
    double d2;
    double d3;
    double d4;

    // SOLO per stavolta non inizializiamo le celle di memoria

    printf("\n*****************************************\n");
    printf("* Occupazione in memoria \n");
    printf("*****************************************\n\n");
    printf(" il char c1 si trova in 0x %p (%d) \n", &c1 , &c1);
    printf(" il char c2 si trova in 0x %p (%d) \n", &c2 , &c2 );
    printf(" il char c3 si trova in 0x %p (%d) \n", &c3 , &c3 );
    printf(" il char c4 si trova in 0x %p (%d) \n", &c4 , &c4 );

    printf(" l' int  i1 si trova in 0x %p (%d) \n", &i1 , &i1 );
    printf(" l' int  i1 si trova in 0x %p (%d) \n", &i2 , &i2 );
    printf(" l' int  i1 si trova in 0x %p (%d) \n", &i3 , &i3 );
    printf(" l' int  i1 si trova in 0x %p (%d) \n", &i4 , &i4 );

    printf(" il long l1 si trova in 0x %p (%d) \n", &l1 , &l1);
    printf(" il long l2 si trova in 0x %p (%d) \n", &l2 , &l2 );
    printf(" il long l3 si trova in 0x %p (%d) \n", &l3 , &l3 );
    printf(" il long l4 si trova in 0x %p (%d) \n", &l4 , &l4 );
```

```
    printf(" il double d1 si trova in 0x %p (%d) \n", &d1 , &d1);
    printf(" il double d2 si trova in 0x %p (%d) \n", &d2 , &d2 );
    printf(" il double d3 si trova in 0x %p (%d) \n", &d3 , &d3 );
    printf(" il double d4 si trova in 0x %p (%d) \n", &d4 , &d4 );

    getchar();

}
```

Una volta compilato ed eseguito il nostro programma stamperà un output simile a questo.

```
**********************************************
* Occupazione in memoria
**********************************************

 il char c1 si trova  0x 0022FF6F (2293615)
 il char c2 si trova  0x 0022FF6E (2293614)
 il char c3 si trova  0x 0022FF6D (2293613)
 il char c4 si trova  0x 0022FF6C (2293612)
 l' int  i1 si trova  0x 0022FF68 (2293608)
 l' int  i1 si trova  0x 0022FF64 (2293604)
 l' int  i1 si trova  0x 0022FF60 (2293600)
 l' int  i1 si trova  0x 0022FF5C (2293596)
 il long l1 si trova  0x 0022FF58 (2293592)
 il long l2 si trova  0x 0022FF54 (2293588)
 il long l3 si trova  0x 0022FF50 (2293584)
 il long l4 si trova  0x 0022FF4C (2293580)
 il double d1 si trova  0x 0022FF40 (2293568)
 il double d2 si trova  0x 0022FF38 (2293560)
 il double d3 si trova  0x 0022FF30 (2293552)
 il double d4 si trova  0x 0022FF28 (2293544)
```

Se osserviamo gli *indirizzi* delle varie variabili che abbiamo dichiarato, possiamo notare che le variabili sono state allocate rispettando l'ordine di dichiarazione (c1, c2, c3, c4, i1,…). La memoria libera è quindi verso i numeri bassi in quanto le celle di memoria sono state allocate dalla …615 verso la …544 (leggendo gli indirizzi stampati in decimale).

Partendo dal presupposto che le nostre printf stampano degli indirizzi rappresentati in byte possiamo notare che i caratteri "distano" un solo byte, gli interi 4 byte. Questo significa che sono quindi rappresentati a 4x8bit ovvero 32 bit.

Procedendo con l'analisi notiamo che i long occupano 4 byte e i double 8 byte.

N.B. Nella notazione esadecimale i numeri di una cifra possono essere 0,1,…,9,A,B,C,D,E,F essendo una base 16 e non 10 come nel caso del decimale. Attenzione quindi a contare i salti di indirizzo se leggiamo numeri in esadecimale.

4. Rappresentazione dei caratteri

I caratteri alfanumerici del codice ASCII (American Standard Code For Information Interchange) compongono praticamente tutti i nostri file di testo e moltissimi altri file. Questi caratteri vengono rappresentati in C mediante la dichiarazione di variabili di tipo `char`.

Un codice associa ad ogni carattere un intero. Ad esempio il codice ASCII associa al carattere '#' il numero decimale 35 e alle lettere '0'… '9' della nostra tastiera i numeri decimali dal 48 al 57.

Ad esempio, i caratteri 'a'… 'z' della nostra tastiera sono stati associati ai numeri decimali dal 97 al 122.

La cosa più importante da ricordare è che i caratteri in C possono essere usati come interi se ci ricordiamo la corrispondenza fra il carattere ed il suo valore nel codice (es: a→97)

In C esistono 3 tipi di caratteri:

- **char**
- **signed char** (rappresentato in complemento a 2)
- **unsigned char**

I tipi signed ed unsigned char sono esattamente rappresentati come interi.

Attenzione, invece il tipo **char** dipende dal compilatore e può essere stato definito come signed oppure unsigned.

Tipicamente vale sizeof(char) ≤ sizeof(int) dove sizeof(char) vale 8 bit.

Tutti gli operatori definiti su int (+, -, *, /, %, ==, !=, <, >, <=, >=) valgono sui caratteri

Attenzione questo è interessante perché permette di capire ordinare i caratteri. Se ad esempio x='a' e y='b' allora la proposizione (y>x) è vera. Questo è possibile perché il C vede il contenuto di x come 97 (ovvero 'a' nel codice ASCII) ed il contenuto di y come 98!

Ad esempio se una variabile carattere x rispetta la seguente proposizione ((x>96)AND(x<123)) allora è un carattere minuscolo.

Le operazioni di input e output dei caratteri si realizzano sempre con scanf e printf usando nella stringa di controllo %c.

Ecco un esempio nel quale immaginiamo di inserire due interi (35 e 97) e li visualizziamo con la printf sia come intero (%d) sia come carattere (%c).

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

Otterremmo in output alle due printf

```
35 48
#  a
```

5. Vettori

Per evitare ogni tipo di confusione è opportuno specificare la relazione che lega i termini ARRAY, VETTORE e STRINGA in C.

Un array e' una struttura di dati costituita da un insieme di variabili dello stesso tipo di dato (intero, carattere, reale ecc.), a cui e' possibile accedere tramite un nome, e referenziare uno specifico elemento attraverso un indice. Nel linguaggio C gli elementi di un array sono allocati in memoria centrale in celle adiacenti, in cui viene associato l'indirizzo più basso, al primo elemento dell' array e l'indirizzo più alto all'ultimo elemento dell'array.

Gli array possono essere definiti ad una dimensione (vettori), a due dimensioni (matrici) , ad n dimensioni (array multidimensionali). Le stringhe sono pertanto array di caratteri.

A questo punto possiamo definire il vettore: Un **vettore** e' un insieme **contiguo** monodimensionale di elementi dello **stesso tipo** (cioè che vengono allocati in memoria in un blocco unico). E' possibile accedere ad un vettore tramite un nome, e referenziarne uno specifico elemento attraverso un indice.

Come di dichiara un vettore? Nella parte dichiarativa del programma si indica il tipo di elemento del vettore (per esempio `int`), spazio, il nome della variabile (ad esempio `x`), e fra parentesi quadre si dichiara il numero di elementi che il vettore dovrà contenere. Ad esempio il codice

```
int x[100];
```

dichiara un array **`x`** costituito da 100 numeri interi.

Come si accede agli elementi del vettore? Usando il nome del vettore e scrivendo fra parentesi quadre l'indice dell'elemento a cui vogliamo accedere. Ad esempio il codice

```
x[0]= 1;
```

assegna il valore 1 al primo elemento dell'array di interi `x`.

 ***ATTENZIONE***:
il C numera gli elementi di un array a partire da 0. Pertanto nell'array x di 100 elementi interi posso assegnare 1 al primo elemento con la riga di codice

`x[0]=1;`

mentre posso assegnare all'ultimo elemento il valore 100 con la riga di codice

`x[99]=100;`

I vettori di solito vengono gestiti mediante cicli `for`. Questo è perché il for deve essere usato quando si conosce il numero di cicli da eseguire. Questo è proprio il caso degli array in quanto il numero di elementi che li compongono è fissato fin dalla dichiarazione

Ad esempio, assegniamo al vettore **`x`**, i numeri da 1 a 100. Questa operazione che deve essere ripetuta per tutti i 100 elementi dell'array può essere implementata semplicemente dalla seguente riga di codice

```
for(i=0; i<100; i++) x[i]=i+1;
```

N.B. Usare le graffe nei costrutti `if` e `for` anche in casi semplici allunga il listato, ma lo rende più leggibile, come in questo caso:

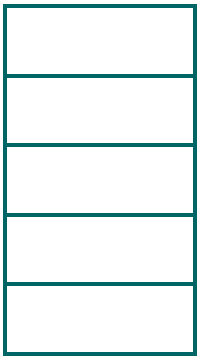
```
for(i=0; i<100; i++)
{
        x[i]=i+1;
}
```

Come è possibile immaginare la situazione in memoria quando dichiaro un array?

Supponiamo di aver dichiarato un array di interi nel seguente modo:

```
int  A[5];
```

In memoria centrale viene allocata una sequenza di locazioni intere come in figura.



Ogni locazione sarà tale da poter contenere un valore dell'array ovvero un intero (4 byte per ogni intero). Per accedere ai 5 elementi useremo queste notazioni A[0], A[1], A[2], A[3], A[4] .

Ecco alcuni esempi:

Il codice

```
A[3] = 7 ;
```

Assegna il valore intero 7 alla quarta componente dell' array A

In questo caso

```
if (A[3]==7)  printf ("%d", A[0]) ;
```

se la quarta componente dell' array A e' uguale a 7 allora viene stampata la prima componente dell'array A.

L'array può essere gestito impiegando espressioni aritmetiche :

```
j=2;
```

```
A[j+2] = …
```

coincide con A[4] ossia con il quinto elemento. Infatti l'indirizzamento di un elemento dell'array può avvenire con una espressione. Ad esempio

```
int v=2, a=1;
```

…

```
x[v+4-a] = 1;
```

corrisponde ad andare ad assegnare 1 a x[5] ovvero il sesto elemento.

Attenzione che A[-1] NON ESISTE . A[5] NON ESISTE !

 ***ATTENZIONE***:

Gli array non possono essere assegnati ad array! Se dichiaro

```
int x[5];

int y[5];

…

x=y ;     // NON E' CORRETTA !! NON COPIO UN ARRAY SULL'ALTRO
```

 ***ATTENZIONE***:

In C non si effettua nessun controllo sull'effettiva esistenza della componente indirizzata. Ad esempio il codice:

```
x[12] = 9;
```

passa senza errori la compilazione. Il numero 9 verrà memorizzato nella tredicesima locazione intera a partire dall'inizio dell'array, anche se questa cella NON appartiene all'array. Probabilmente stiamo andando a scrivere su un'altra variabile (o peggio ancora su una sua porzione se la variabile è memorizzata su più byte come i nel caso dei `float`) creando degli errori difficilissimi da debuggare.

Occorre quindi porre molta attenzione quando si usano espressioni complesse per indicizzare gli elementi.

Che indirizzo hanno le celle di un array?

```
main()
{
     int i, x[100];
    for(i=0; i<100; i++) x[i]=i+1;
    printf("Il valore di x e' %d\n", x);
    printf("Il valore di x[0] e' %d\n", x[0]);
    printf("L'indirizzo di x[0] e' %d\n", &x[0]);
    printf("Il valore di x[99] e' %d\n", x[99]);
    printf("L'indirizzo di x[99] e' %d\n", &x[99]);
    printf("La memoria occupata dal vettore e' %d\n",
        (void *)&x[99]+sizeof(int)-(void *)&x[0]);
}
```

Questo codice calcola l'occupazione di memoria del vettore x[100], tenendo in considerazione la quantità di celle di memoria (bytes) usate per quel tipo di variabile (int, cioè 4 bytes per elemento). L'operatore (void *) fa in modo che il calcolo venga effettuato tra locazioni di memoria e non semplicemente tra due numeri che indicano la locazione finale (x[99]) e quella iniziale (x[0]). Il calcolo è effettuato nel seguente modo. Immaginiamo di avere un vettore A di 4 interi A[0],A[1],A[2],A[3]. Ogni intero occupa 4 byte partendo, per esempio, dall'indirizzo 1001:

1001 byte A[0]
1002 byte
1003 byte
1004 byte
1005 byte A[1]
1006 byte
1007 byte
1008 byte
1009 byte A[2]
1010 byte
1011 byte
1012 byte
1013 byte A[3]
1014 byte
1015 byte
1016 byte

Se si calcola l'indirizzo di A[3] (ovvero &A[3] = 1013) meno l'indirizzo di A[0] (ovvero &A[0] = 1001) ovviamente rimangono fuori gli ultimi 4 byte. Quindi nel calcolo `(void *)&x[3]+ sizeof(int)-(void *)&x[0]`
abbiamo 1013 + 4 - 1001 = 16 byte come ci aspettavamo.

6. Matrici

Ricordiamo che un array e' una struttura di dati costituita da un insieme di variabili dello stesso tipo di dato (intero, carattere, reale ecc.), a cui e' possibile accedere tramite un nome, e referenziare uno specifico elemento attraverso un indice. Nel linguaggio C gli elementi di un array sono allocati in memoria centrale in celle adiacenti, in cui viene associato l'indirizzo più basso, al primo elemento dell' array e l'indirizzo più alto all'ultimo elemento dell'array.

Gli array bidimensionali in C sono definite **matrici**.

Come di dichiara una matrice? Nella parte dichiarativa del programma si indica il tipo di elemento della matrice (per esempio `int`), spazio, il nome della variabile (ad esempio `M`), e fra doppie parentesi quadre si dichiara il numero di elementi che la matrice dovrà contenere. Ad esempio il codice

```
int M[10][10];
```

dichiara un matrice **`M`** costituita da 100 numeri interi.

Come si accede agli elementi della matrice? Usando il nome della matrice e scrivendo fra le doppie parentesi quadre gli indici dell'elemento a cui vogliamo accedere. Ad esempio il codice

```
M[0][2]= 1;
```

assegna il valore 1 all'elemento sulla prima riga, terza colonna della matrice di interi `M`. Ricordiamo che in C tutti gli array (quindi vettori, matrici e stringhe) gli elementi vengono indirizzati a partire dal numero 0 e non 1.

Come gestire gli elementi di una matrice?
Le matrici vengono indicizzate di solito mediante due cicli `for` annidati (come tutti i tipi di array). Esaminiamo questo esempio di codice

```
int  matrix[6][4];

int i,j;

for  (i=0; i<6; i++)

{

     for  (j=0; j<4;   j++)

     {

          matrix[i][j]= 0 ;

     }

}
```

In esso viene definita una matrice di 6 righe e 4 colonne. Vengono definiti inoltre due indici. Essi sono necessari per indirizzare gli elementi della matrice. L'indice i servirà per scandire le righe ed un indice j per scandire le colonne. Il codice inizializza a zero la matrice.

Come distinguere le righe dalle colonne?
Per chiarire come indirizzare gli elementi di una matrice senza confondere gli indici delle righe e delle colonne si studi lo schema seguente:

	Secondo indice (j)			
Primo indice (i)	0 , 0	0 , 1	0 , 2	0 , 3
	1 , 0	1 , 1	1 , 2	1 , 3
	2 , 0	2 ,1	2 , 2	2 , 3
	3 , 0	3 , 1	3 ,2	3 , 3
	4 , 0	4 , 1	4 ,2	4 , 3
	5 , 0	5 , 1	5 ,2	5 , 3

Lo schema rappresenta dal punto di vista logico la matrice. Attenzione, la matrice è memorizzata diversamente in memoria: gli elementi della matrice vengono memorizzati sequenzialmente. Dal punto di vista dell' allocamento si può immaginare che la matrice venga srotolata e disposta in memoria in memoria formando un vettore lungo quanto tutti gli elementi della matrice.

Come inizializzare gli elementi di una matrice?

L'inizializzazione di una matrice ad esempio dichiarata nel seguente modo
```
int matrix[3][3]
```
con le prime nove potenze di due può essere scritta cosi:
```
int  matrix[3][3]={{1,2,4},{8,16,32},{64,128,256}};
```

 ***ATTENZIONE***:

Le matrici non possono essere assegnate ad altre matrici! Se dichiaro

```
int M1[5][5];

int M2[5][5];

…

M1 = M2 ;  // NON E' CORRETTA !! NON COPIO UN ARRAY SULL'ALTRO
```

 ***ATTENZIONE***:

In C non si effettua nessun controllo sull'effettiva esistenza dell'elemento indirizzato. Ad esempio il codice:

```
 x[12][3] = 9;
```

passa senza errori la compilazione anche se la matrice ha solo 10 righe!

Il valore 9 verrà scritto in memoria in una posizione errata, probabilmente alterando il valore di una altra nostra variabile. Occorre quindi porre molta attenzione quando si usano gli indici o espressioni complesse per indicizzare gli elementi.

Come stampare gli elementi di una matrice?

Rivediamo in un semplice programma tutte le operazione riguardanti le matrici

```
#include<stdio.h>
int main()
{
int  matrice[5][4];
/* dichiarazione della matrice di nome matrice*/
int  i , j ;
/* dichiarazione indice di riga ed indice di colonna*/
for (i=0; i<5; i++)
{
      for (j=0; j<4; j++)
      {
            matrice [i][j] = i * j ; // come esempio
      }
}
for (i=0; i<5; i++)
{
      for (j=0; j<4; j++)
      {
            printf ("%3d ", matrice [i][j]) ;
      }
      printf ("\n") ; /* a capo, ad ogni riga della matrice*/
}
return 0;
}// end main
```

In questo codice i primi due cicli for annidati semplicemente riempiono gli elementi della matrice con il prodotto degli indici. I successivi due cicli for annidati stampano gli elementi della matrice. La riga seguente

```
            printf ("%3d ", matrice [i][j]) ;
```

permette di stampare gli interi con solo 3 cifre ed uno spazio di separazione. Terminata la stampa di una riga, l'istruzione

```
      printf ("\n") ; /* a capo, ad ogni riga della matrice*/
```

permette di andare a capo, iniziando a stampare la prossima riga della matrice effettivamente a capo. Togliendo questa riga tutta la matrice verrebbe stampata sulla stessa riga del nostro terminale.

7. Stringhe

Le stringhe, esattamente come i vettori e le matrici sono degli array, ovvero una struttura di dati costituita da un insieme di variabili dello stesso tipo di dato (intero, carattere, reale ecc.), a cui e' possibile accedere tramite un nome, e referenziare uno specifico elemento attraverso un indice.

Gli array monodimensionali di caratteri in C sono definite **stringhe**.

Il terminatore

Occorre però aggiungere che una stringa correttamente allocata in memoria termina SEMPRE con il carattere terminatore di stringa: `'\0'` (una cella tutta a zero). Non si considera il terminatore come facente parte della stringa (ad esempio non lo si conteggia fra i caratteri) ma DEVE essere presente.

Come di dichiara una stringa? Nella parte dichiarativa del programma si indica il tipo di elemento della stringa che NECESSARIAMENTE deve essere un carattere (quindi `char`), spazio, il nome della variabile (ad esempio `s`), e fra doppie parentesi quadre si dichiara il numero di elementi che il vettore di caratteri (ovvero la stringa) dovrà contenere. Ad esempio il codice

```
char s[10];
```

dichiara un stringa **`s`** costituita da 10 caratteri. In effetti una stringa non è null'altro che un array di caratteri. Valgono quindi tutte le considerazioni sull'uso degli array viste nelle lezioni precedenti

Costanti stringa

Se dobbiamo assegnare stringhe più lunghe esiste un metodo più pratico. Immaginiamo di volere inizializzare una stringa s con la valore della seguente stringa di caratteri `"Laboratorio di programmazione per la sicurezza"`. Il codice per farlo è il seguente:

```
char s[]="Laboratorio di programmazione per la sicurezza";
```

E' un esempio di array non dimensionato a cui si assegna una costante stringa. In C una costante stringa si racchiude tra virgolette doppie, come **`"Crema"`**. Il terminatore di stringa viene aggiunto ***automaticamente*** dal compilatore.

ATTENZIONE:

NON CONFONDERE LA DICHIARAZIONE DI UN CARATTERE (esempio ‘m’) scrivendola invece come stringa (esempio “m”). Questo errore produce malfunzionamenti difficili da debuggare in programmi complessi!

La costante stringa "M" è diversa dal carattere 'M' in quanto "M" e' composta da **due** caratteri: 'M'+'\0' ovvero il carattere M ed il terminatore. Quindi NON sono la stessa cosa!

Come si accede agli elementi della stringa?

Immaginiamo di dichiarare la stringa `s` ed di inizializzarla al valore costante stringa `"mamma"` mediante il codice

```
char s[]="mamma";
```

La situazione è la seguente:

- Dal punto di vista del **programma** la stringa conta **cinque** caratteri ('m'+'a'+'m'+'m'+'a');
- Dal punto di vista della **memoria** sono stati allocate **sei** celle per poter fare spazio anche al terminatore ('m'+'a'+'m'+'m'+'a'+ '\0'), ma nei nostri programmi di questo fatto non ce ne accorgiamo;
- Ogni carattere allocato per la stringa occupa un **byte** in memoria.

Essendo la stringa un array di caratteri accediamo ad essa come in un array , quindi possiamo usare porzioni di codice come le seguenti:

```
int  v[5];
char s[]="mamma";
…

v[0]= 1  ; // assegno l’intero      1  alla prima cella
s[0]= 'x'; // assegno il carattere 'x' alla prima cella. Att! non uso “x”
```

L’input/output con le stringhe

Esattamente come l’input da tastiera di un intero è realizzato in C mediante la chiamata ad una scanf del tipo mediante il segnaposto “%d” una stringa viene acquisita mediante il segnaposto “%s”. Si consideri il seguente esempio di codice:

```
int  conto;
char s[128];

printf(“Immetti un intero e batti enter”);
scanf ( "%d" , &conto );
...
printf(“Immetti una stringa senza spazi e batti enter”);
scanf ( "%s" , s );
```

che mostra come sia possibile acquisire una stringa. Attenzione la prima scanf riceve un SOLO intero, mentre la seconda scanf riceve un ARRAY di caratteri e li deposita tutti in memoria a partire dall'indirizzo `s`. La funzione scanf provvede a memorizzare correttamente anche il terminatore di stringa.

 ***ATTENZIONE***:

Proprio perché non è noto a priori il numero di caratteri che l'utente immetterà da tastiera occorre dichiarare una stringa abbastanza grande per soddisfare le nostre esigenze.
Se ad esempio compiliamo la porzione di codice seguente

```
char s[5];   // attenzione, allocate spazio solo per 5 caratteri
...
printf(“Immetti una stringa senza spazi e batti enter”);
scanf ( "%s" , s );
```

e l'utente immette da tastiera la stringa “mamma” la situazione in memoria è la seguente

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]
‘m’	‘a’	‘m’	‘m’	‘a’	‘\0’

dove ogni cella occupa un byte.

Se l'utente immette “zio” la situazione in memoria è la seguente

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]
‘z’	‘i’	‘o’	‘\0’	??	??

Le ultime due celle rimarranno allocate ma non vi sarà nessun valore corretto memorizzato. Più precisamente, dei numeri vi sono sicuramente memorizzati ma probabilmente rimasti da precedenti programmi/dati che sono stati caricati in quelle celle della memoria centrale. Naturalmente non hanno alcun senso per il nostro programma.

Se l'utente immette "unastringalunghissima" la situazione in memoria è la seguente:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]
'u'	'n'	'a'	's'	't'	'r'

La stringa viene memorizzata nello spazio che abbiamo riservato e poi scrive su posizioni di memoria adiacenti!

Se l'utente immette "uso gli spazi" la situazione in memoria è la seguente:

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]
'u'	's'	'o'	\0'	??	??

La memorizzazione dei caratteri immessi da tastiera si ferma se si incontra uno spazio. Dove rimangono i caratteri non memorizzati? Nel buffer di tastiera. Questo argomento verrà ulteriormente dettagliato nella lezione riguardante i flussi di dati in C e la gestione dell'input/output.

Conteggio dei caratteri e stampa delle stringhe

In una situazione di codice come quella descritta nella quale non sappiamo a priori quanti caratteri immetterà da tastiera l'utente, come possiamo contare i caratteri immessi?
Ecco una possibile soluzione.

```
char s[128];
int lunghezza;
...

printf("Immetti una stringa senza spazi e batti enter");
scanf ( "%s" , s );
lunghezza = strlen(s);
```

La funzione strlen() prende in ingresso un indirizzo al quale inizia una stringa e ritorna il numero di caratteri. Molto semplicemente la funzione scorre la stringa contando i caratteri prima del terminatore

Allo stesso modo la chiamata alla funzione `printf`

```
printf("Ecco la stringa immessa: %s "  , s );
```

stampa `"Ecco la stringa immessa: "` ed al posto del segnaposto %s, stampa carattere per carattere la stringa s sino a quando viene incontrato il terminatore.

Nulla ci vieta di scrivere la porzione di codice che calcola la lunghezza di una stringa senza usare la funzione strlen(). Ad esempio il seguente codice:

```
#include <stdio.h>
#include <string.h>
int main()
{

char s[]="esercitazione";
int lunghezza, i;


// lunghezza  = strlen(s);

lunghezza = 0;
i=0;
while (s[i]!= '\0')
{
 lunghezza = lunghezza + 1;
 i = i + 1;
 }
printf( "\n Stringa:%s  lunga %d caratteri \n ", s , lunghezza);

getchar();

}
```

Produce a monitor

```
 Stringa:esercitazione  lunga 13 caratteri
```

Il codice è basato sul ciclo `while` (necessario perché non sappiamo quanti cicli dovranno essere scanditi) che continua a contare i caratteri finché il carattere contenuto nella stringa `s` alla posizione `i`-esima è il terminatore (`s[i]!= '\0'`).