http://www.elet.polimi.it/upload/martucci/index.html

- Lucidi del Corso di Informatica C Aerospaziali
- AA 2002-03
- 4° Modulo
- Tipi Strutturati
- -->Array ... Vettore
- -->Stringhe ..

Fondamenti di Informatica C - AA 2002-03 - Microsoft Internet Explorer

File Modifica Visualizza Preferiti Strumenti ?

Indietro Cerca Preferiti Multimedia Indirizzo Vai

Politecnico di Milano
Informatica C

Facoltà di Ingegneria - Milano Bovisa
Corso di Laurea Ingegneria Aerospaziale [E-O]
Anno Accademico 2002-03 - secondo semestre

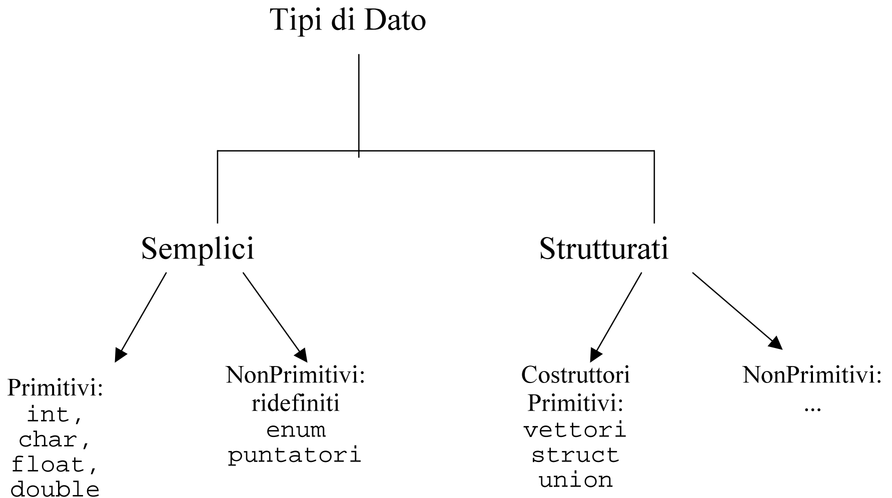
Proff. R. Martucci - L. Mazzei - M. Mussini

ultimo aggiornamento domenica 23 febbraio 2003

| | |
|---|---|
| Orario delle lezioni ed aule | sarà allocata una seconda aula di laboratorio e si provvederà alla divisione in 4 squadre (due per aula) |
| Programma del Corso | la articolazione nelle unità didattiche sarà contenuta nel piano delle lezioni definitivo |
| Piano delle Lezioni | Attenzione al calendario di lezioni, esercitazioni e laboratorio (giorni di calendario e N. di ore) - nel caso di ore perse per qualunque motivo, i recuperi verranno comunicati sul sito |

Internet

# Tipi di Dato Strutturati




### Tipi di dato strutturati:

I dati strutturati (o strutture di dati) sono ottenuti mediante composizione di altri dati (di tipo semplice, oppure strutturato).

### Tipi strutturati in C:

- **vettori** (o array)
- **record** (struct)
- **record varianti** (union)

# Vettori

Un vettore e` un insieme **ordinato** di elementi **tutti dello stesso tipo**.

### Caratteristiche del vettore:

- **omogeneita`**
- **ordinamento** ottenuto mediante dei valori interi (***indici***) che consentono di accedere ad ogni elemento della struttura.

| | |
|---|---|
| 0 | X |
| 1 | Y |
| 2 | Z |
| .. | .. |
| | |
| n-1 | |

## TIPI STRUTTURATI DEL C - COSTRUTTORE ARRAY - 1

Consente di dichiarare variabili che tramite un unico identificatore rappresentano un aggregato di informazioni omogenee.

**Costruttore** array **[ ]**
- un array è definito come una **sequenza** di lunghezza **fissata** di **elementi omogenei** (dello stesso tipo)
- ogni elemento è individuato tramite un **indice** che rappresenta la sua posizione nell'array
- un array rappresenta in modo implicito un ordinamento tra gli elementi, legato alla loro posizione
- se DIM è la dimensione dell'array (DIM costante intera), l'**indice** degli elementi assume valori da 0 a DIM -1

**Dichiarazione di una variabile array**: sintassi C

```
tipo_elem  nome_array [DIM]
```

**`DIM`**
: è un'espressione **costante** di tipo intero che definisce la dimensione dell'array

**`tipo_elem`**
: è il tipo degli elementi (built-in o user-defined, semplice o strutturato)

**`nome_array`**
: è l'identificatore (nome) di una variabile (di tipo) array costituita da DIM elementi di tipo tipo_elem

## COSTRUTTORE ARRAY - 2

**Accesso ai singoli elementi**: tramite **indice** (operazione legata al costruttore di tipo)

**`nome_array[i]`**: con questa notazione si individua il valore dell'elemento i-mo

**`i`** è una variabile (che rappresenta l'indice dell'array) di tipo *integral* i cui valori significativi per accedere agli elementi dell'array sono da 0 a DIM -1

**Operatori** utilizzabili per i singoli elementi: operatori «leciti» per il tipo degli elementi

**Spazio di memoria** allocato per una variabile di tipo array (numero di byte)

DIM * sizeof(tipo_elem)

lo spazio di memoria viene allocato ad indirizzi fisici contigui

**Rappresentazione in memoria** di un array**: `int vett[4]`**

| | |
|---|---|
| vett[0] | valore di vett[0] |
| vett[1] | valore di vett[1] |
| vett[2] | valore di vett[2] |
| vett[3] | valore di vett[3] |

## COSTRUTTORE ARRAY - 3

La dichiarazione di una variabile di tipo array

- alloca lo spazio di memoria sufficiente a contenere la variabile
- assegna al nome della variabile (`nome_array`) l'indirizzo del primo byte che contiene il primo elemento dell'array
- il valore associato a nome_array è quindi un **indirizzo** ed è costante
- a `nome_array`, oltre all'indirizzo è associata anche l'informazione **sul tipo** degli elementi che lo compongono e, in particolare, sulla quantità di memoria allocata per quel tipo. Questa «associazione» viene fatta dal compilatore in fase di traduzione del programma, tramite opportuna modalità di indirizzamento in linguaggio macchina.

```
int vett[30];
```

**Aritmetica degli indirizzi** per gli array:

Le notazioni

```
vett        e   &vett[0]
vett + 1    e   &vett[1]
vett + i    e   &vett[i]
```

definiscono lo stesso indirizzo di memoria, e

- l'espressione `(vett + i) - (vett + j)`
  da come valore un intero pari al numero di elementi tra j e i

# Vettori

## Operatori:

In C **non esistono operatori specifici per i vettori**; per operare sui vettori è necessario operare singolarmente sugli elementi componenti (coerentemente con il tipo ad essi associato).

☞ **non e` possibile** l'assegnamento diretto tra vettori:

```
int V[10], W[10];
…
V=W;   /* e` scorretto! */
```

☞ Non e` possibile leggere (o scrivere) un intero vettore (a parte, come vedremo, le ***stringhe***); occorre leggere/scrivere le sue componenti; ad esempio, per leggere un vettore:

```
int V[100];
int  i;

for(i=0; i<100; i++)
    {   printf("valore %d-simo? ", i);
        scanf("%d", &V[i]);
    }
```



## Gestione degli elementi di un vettore:

Le singole componenti di un vettore possono essere manipolate coerentemente con il tipo ad esse associato.

**Ad esempio:** int A[100];

☞ agli elementi di A e` possibile applicare tutti gli operatori definiti per gli interi.

**Quindi:**

```
    A[i] =n%i;
    A[10]=A[0]+7;
    scanf("%d", &A[i]);
…
```

# Vettori

☞ L'indice deve essere di tipo *integral type* (int, char o enum). Ad esempio:

```
#include <stdio.h>
typedef enum{blu, giallo, rosso} indice;

main()
{indice k=blu;
 int A[3];

for(k=blu; k<=rosso; k++)
   {
   printf("Dammi elemento %d: ", k);
   scanf("%d", &A[k]);
   }
printf("Valore %d:%d\n",blu,A[blu]);
printf("Valore%d:%d\n",giallo,A[giallo]);
printf("Valore %d:%d\n",rosso,A[rosso]);
}
```



# Vettore come costruttore di tipo

In C e` possibile utilizzare il costruttore [ ] per introdurre tipi di dato non primitivi che rappresentano particolari vettori.

### Sintassi della dichiarazione:

typedef **<tipo-componente> <tipo-vettore>** [**<dim>**]

dove:

- <tipo-componente> e` l'identificatore di tipo di ogni singola componente
- <tipo-vettore> e` l'identificatore che si attribuisce al nuovo tipo
- <dim> e` il numero di elementi che costituiscono il vettore (deve essere una costante).

### Ad esempio:

```
typedef int Vettori[30];

Vettori  V1,V2;
```

☞ V1 e V2 sono variabili di tipo Vettori; ognuno rappresenta un vettore di 30 elementi interi.

☞ V1 e V2 possono essere utilizzati come vettori di interi.

# Vettori

### Riassumendo:

### Variabili di tipo vettore:

**<tipo-componente> <nome>[<dim>];**

### Vettore come costruttore di tipo:

**typedef <tipo-componente> <tipo-vettore> [<dim>];**

### Vincoli:

- <dim> e` una **costante intera**.
- <tipo-componente> e' un **qualsiasi** tipo, semplice o strutturato.

### Uso:

- il vettore e' una sequenza di dimensione fissata <dim> di componenti dello stesso tipo <tipo_componente>.
- la singola componente i-esima di un vettore V e' individuata dall'indice i-esimo, secondo la notazione V[i].

☞ L'intervallo di variazione degli indici e`

[0,..<dim>-1]

- sui singoli elementi e` possibile operare secondo le modalita` previste dal tipo <tipo_componente>.



# Inizializzazione di un vettore

### Mediante un ciclo:

Per attribuire un valore iniziale agli elementi di un vettore, si puo` attuare con una sequenza di assegnamenti alle N componenti del vettore.

### Ad Esempio:

```
#define N 30
typedef int vettore [N];
vettore v;
int i;
...
for(i=0; i<N;i++)
  v[i]=0;
```

☞ La #define rende il programma piu` facilmente modificabile.

### Inizializzazione in fase di definizione:

In alternativa, e` possibile inizializzare un vettore in fase di definizione.

### Esempio:

```
int v[10] = {1,2,3,4,5,6,7,8,9,10};

/*  v[0] = 1; v[1]=2;...v[9]=10; */
```

**Addirittura e` possibile fare:**

```
int v[]={1,2,3,4,5,6,7,8,9,10};
```

☞ La dimensione e` determinata sulla base dell'inizializzazione.



**Esempio:**

**Somma di due vettori:** si realizzi un programma che, dati da standard input gli elementi di due vettori A e B, entrambi di 10 interi, calcoli e stampi gli elementi del vettore C (ancora di 10 interi), ottenuto come la somma di A e B.

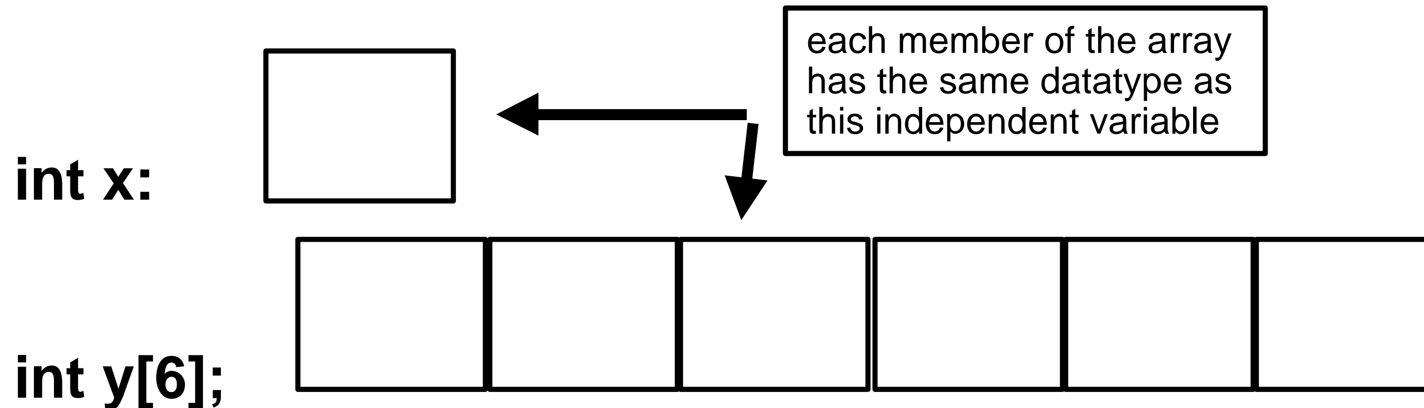
```
#include <stdio.h>
typedef int vettint[10];

main()
{
vettint A,B,C;
int i;

/* lettura dei dati */
for(i=0; i<10; i++)
{ printf("valore di A[%d] ?, i);
  scanf("%d", &A[i]);
}
for(i=0; i<10; i++)
{ printf("valore di B[%d] ?, i);
  scanf("%d", &B[i]);
}

/* calcolo del risultato */
for(i=0; i<10; i++)
  C[i]=A[i]+B[i];

/* stampa del risultato */
for(i=0; i<10; i++)
  printf("C[%d]=%d\n",i, C[i]);
}
```

# Basic Properties

- **Aggregate of data items of same type**
- **Elements are individual data items**

  each has the same type, size, and name (distinguished by an index)

  elements are stored in contiguous memory



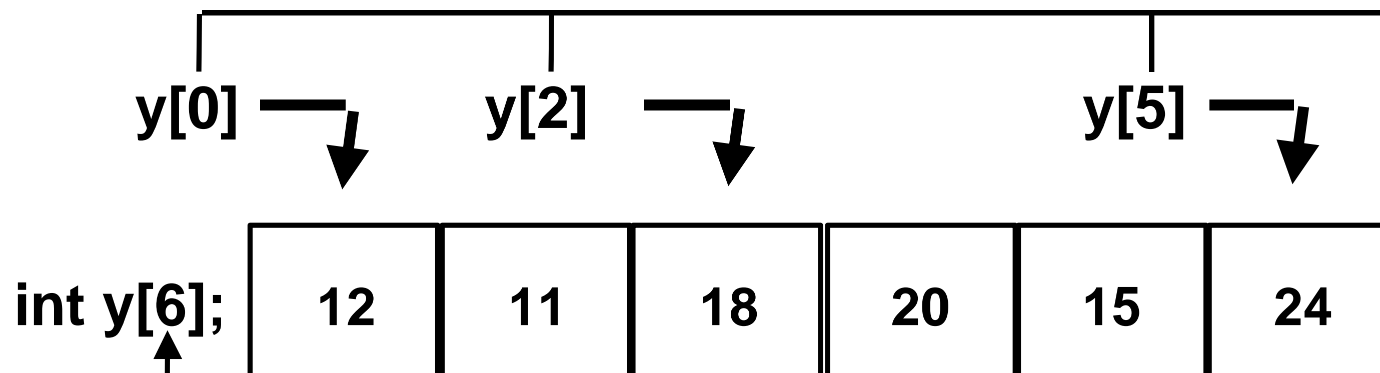
# Basic Properties

- **One name, many parts**

  an array is an indexed variable

  elements are referenced by index (subscript)

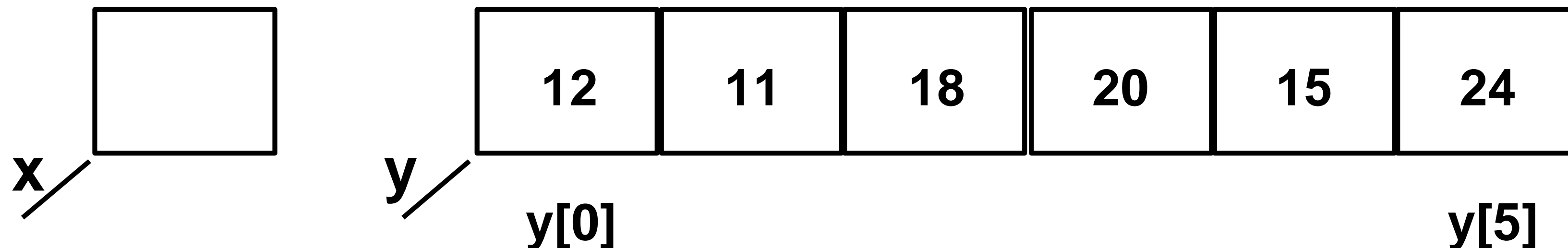
  begins with zero; ends with 1 less than # of elements



# Basic Properties

- **Array name is a constant**

  represents address of the 1st element; which is all you need to get to the other elements!




**y is equivalent to &y[0]**

**x is a variable**
**y is a constant**
**y is an address**
**&y[0] is an address**
**&y[0] is a constant**
**y[0] is a variable**

```
x = 5      /* allowed; variable, not address*/
y[0] = 5   /* allowed; variable, not address*/
y = 18     /* address constant; not allowed */
```

# Declaring Arrays

- **This is when you specify # of elements**

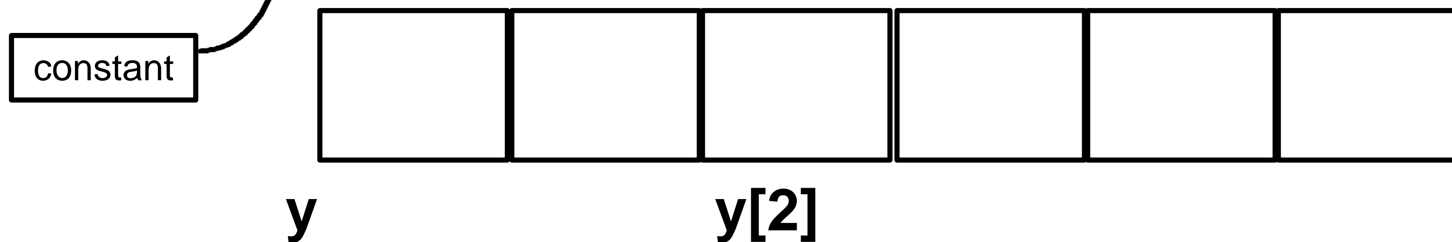
  can be a constant **OR** expression (with constant value)

- **MUST be known at compile time**

  given **ONCE** in [ ] - never again!

expression, but constant value

```
#define MAX_SIZE 5

int y[6];                    int y[MAX_SIZE + 1];
```



# Declaring Arrays

- **Initializing at declaration/definition**
  - can initialize every element allocated

    int y[6] = {12, 11, 18, 20, 15, 24};
  - can initialize fewer than all elements allocated

    int y[6] = {12}; /* remaining elements are set to 0 */
  - cannot initialize more elements thanallocated

    int x[2] = {5, 8, 9};
- **Using a default number of elements**

  let compiler determine number of elements to allocate

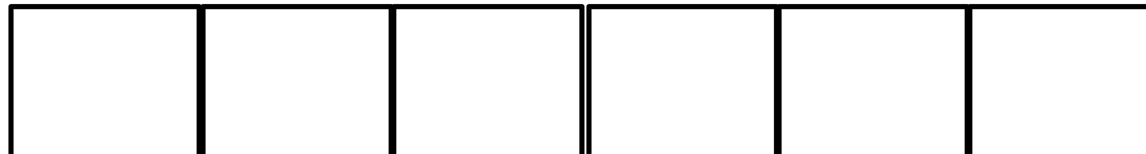
    int y[ ] = {12, 11, 18, 20, 15};

# Examples (definitions)

```
#define MAX 5

int w[MAX + 1];

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

int y[6] = {1, 2, 3};

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

How many elemetns get allocated in each case?

What are their values?



# Referencing Arrays

- **Using an index**

  - represents **OFFSET** from 1st element; this is the reason elements must be of the same type (type indicated amount of storage allocated per element)

  - Index may be a constant, integer variable, or integer expression

  y[0] = 3;   y[ctr] = 5;   y[2 + 3 * x] = 10;

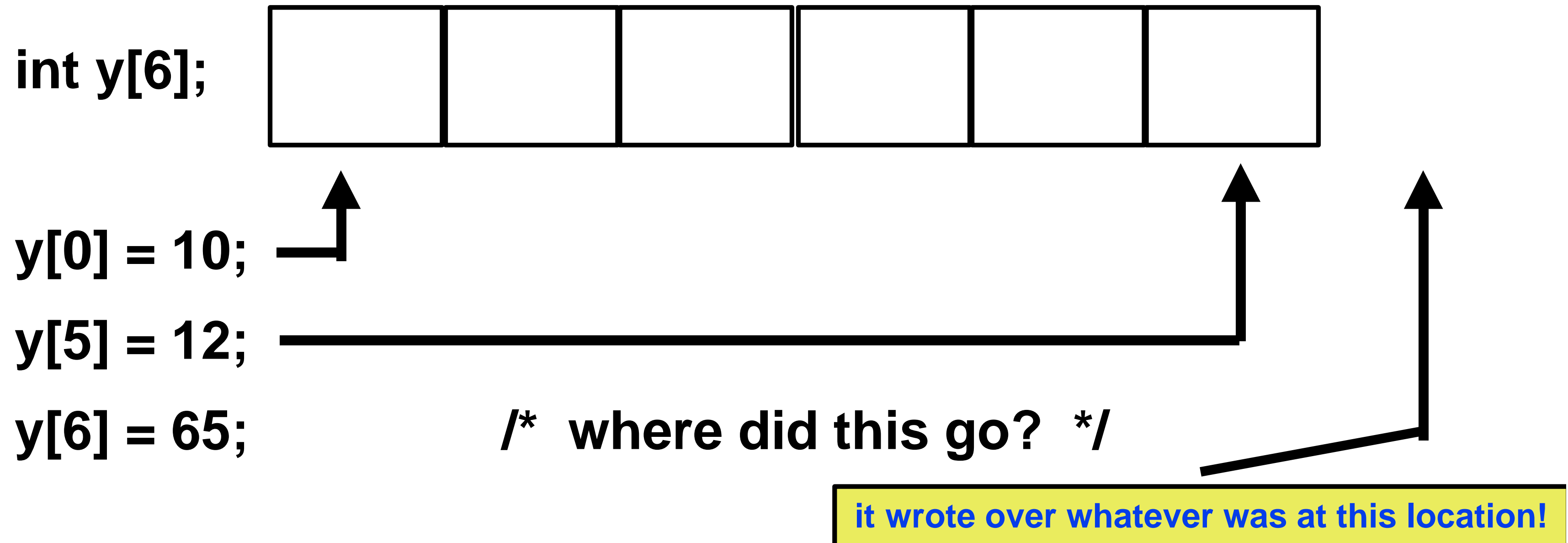
- **C does NO bounds checking**

  result of exceeding bounds is system dependent!

  You typically will not get a runtime error - it will just allow you to overwrite unprotected areas of memory!

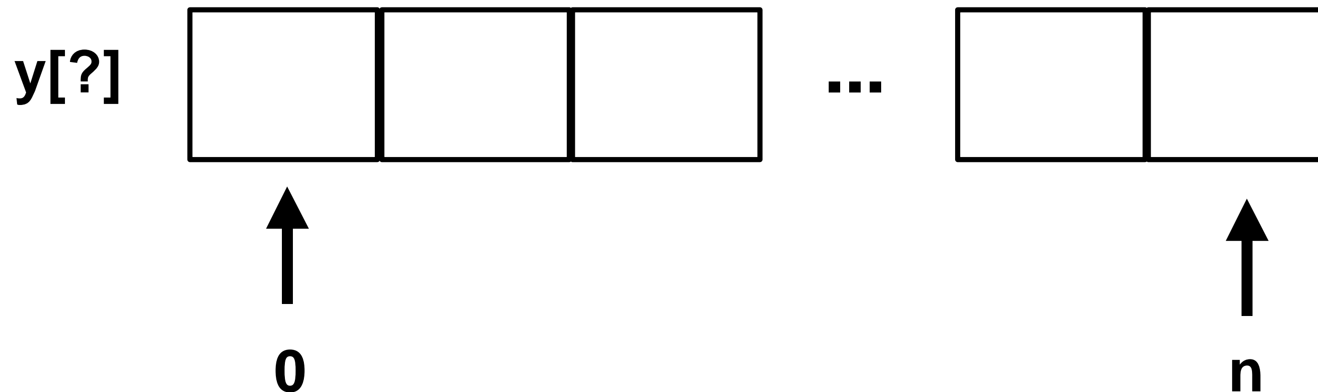
  'Mainframes' (e.g., AXP) will typically give you a runtime error only after you try to exceed bounds of your allotted code space, not before you corrupt your program/data!

# Referencing Arrays




Exceeding array bounds is a common problem. Be careful to only reference existing elements in your arrays!

# Usage with sizeof()




**sizeof( ) returns number of BYTES in ENTIRE array**

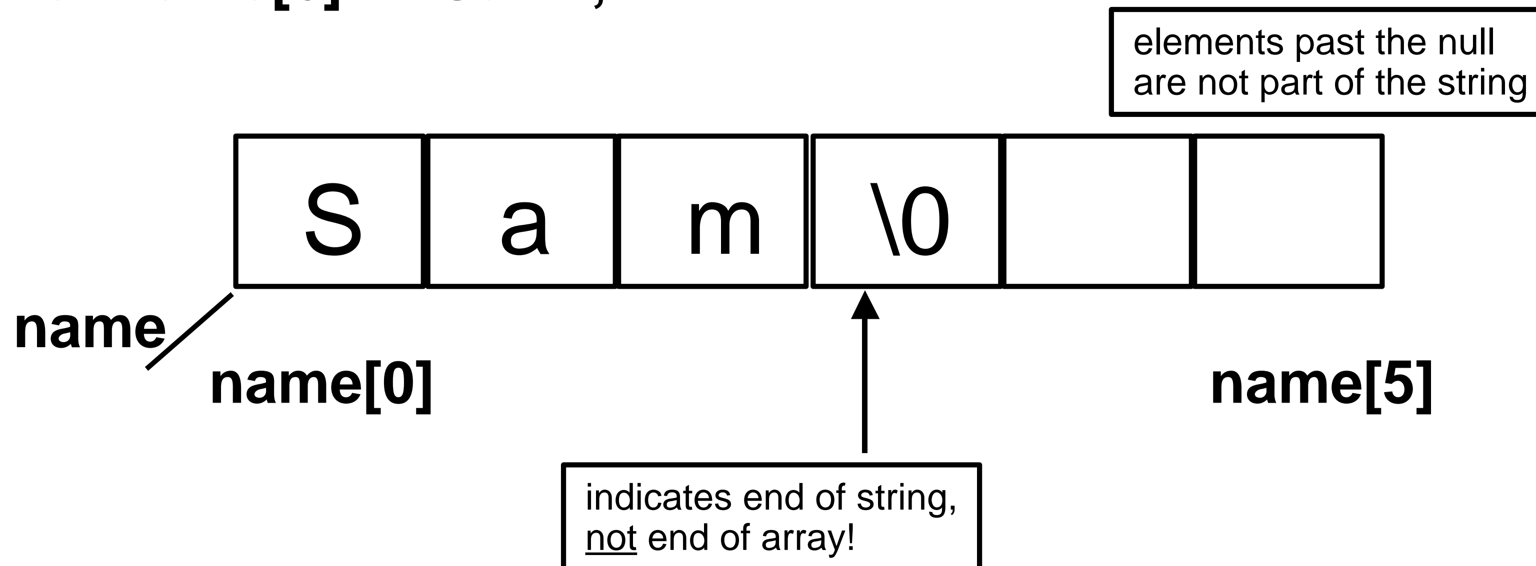
because array is a data structure

| | |
|---|---|
| **sizeof(array)/sizeof (char)** | gives # of elements in array, if array of char (equal to sizeof(array), which is # of bytes) |
| **sizeof(array)/sizeof(int)** | gives # of elements in array, if array of int |
| **sizeof(array)/sizeof(array[0])** | gives # of elements in array, regardless of type! |

# Arrays as Strings

- **No C data type for strings**

  uses null-terminated arrays of char; double quotes denotes a string (e.g., null-terminated array of char)



# Arrays as Strings

- **Initialize**

  can do each char separately or as a string constant

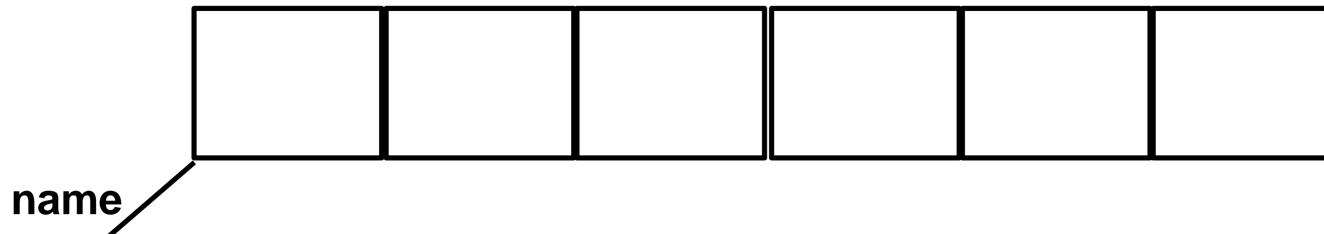
char name[6] = {'C', 'h', 'u', 'c', 'k', '\0'};

initialization list of individual chars
Note null character at end - required for a string!

char name[6] = "Sam";

double quotes implies a null character at the end

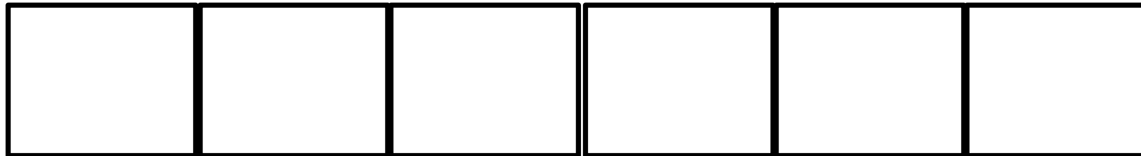
**What would be the result from each? How do they differ?**



# Arrays as Strings

- **Load from keyboard**

one char at a time or as a string



name

```
for(ctr = 0; name[ctr] != '\n'; ++ctr)
    name[ctr] = getchar( );
```

reads each character until the \n is encountered; you must insert the null at the end!

```
scanf("%s", name);
```

reads string from keyboard; converts the \n to a null, but will stop at first 'whitespace' character!

```
gets(name);
```

reads string from keyboard, including any whitespace; converts \n to null

# Multidimensional Arrays

- **Declared & initialized**

  similar to one-dimensional

  nested braces, commas

```
char name[3][8] =  {  {'S', 'a', 'm', '\0'},
                      {'C', 'h', 'u', 'c', 'k', '\0'},
                      {'M', 'a', 'r', 'k', '\0'}  };
   --- or ---
char name[3][8] = {"Sam", "Chuck", "Mark"};
```

the two above initializations are equivalent, since both use strings; if the \0 (null) characters were not in the first example, it would not be the same!

## Libreria standard sulle stringhe

Il C fornisce una libreria standard di funzioni per la gestione di stringhe.
Per poterla utilizzare e` necessario includere il file header <string.h>:

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

Tra tutte, le funzioni piu` comunemente utilizzate sono:

- **strlen**
- **strcmp**
- **strcat**
- **strcpy**



## Libreria standard sulle stringhe

- **Lunghezza di una stringa:**

**int strlen(char str[ ]);**

restituisce la lunghezza (cioe`, il numero di caratteri significativi) della stringa str specificata come argomento.

**Ad esempio:**

```
char S[10]="bologna";
int k;
…
k=strlen(S); /* k vale 7*/
```

- **Confronto tra stringhe:**

**int strcmp(char str1[ ], char str2[ ])**

esegue il confronto tra le due stringhe date str1 e str2. Restituisce:

- **0** se le due stringhe sono identiche;
- un **valore negativo**(ad esempio, -1), se str1 precede str2 (in ordine lessicografico);
- un **valore positivo** (ad esempio, +1), se str2 precede str1 (in ordine lessicografico).

**Ad esempio:**

```
char S1[10]="bologna";
char S2[10]="napoli";
int k;
…
k=strcmp(S1,S2); /* k < 0 */
k=strcmp(S1,S1); /* k=0*/
k=strcmp(S2,S1); /* k>0 */
```

- **Concatenamento di stringhe:**

**strcat( char str1[ ], char str2[ ])**

concatena le 2 stringhe date str1 e str2. Il risultato del concatenamento e` in str1.

**Ad esempio:**

```
char S1[15]="reggio";
char S2[15]="emilia";
strcat(S1, S2); /*S1="reggioemilia"*/
```

- **Copia di stringhe:**

**strcpy(char str1[ ], char str2[ ])**;

copia la stringa str2 in str1.

**Ad esempio:**

```
char S1[10]="Giuseppe";
char S2[10];
…
strcpy(S1,S2); /* S1="Giuseppe"*/
```

## COSTRUTTORE ARRAY: RIDEFINIZIONE DI TIPO

La dichiarazione di tipo (ridefinizione)

```
typedef tipo_elem  tipo_array [DIM];
```

definisce un nuovo **tipo** di nome `tipo_array`: il tipo definito è un array di DIM elementi di tipo `tipo_elem`

La dichiarazione di variabile:

```
typedef int vett[DIM];
vett   vettore_di_ingresso;
```

dichiara una variabile di nome `vettore_di_ingresso` e di tipo `vett`.

## ARRAY DI ARRAY - 1

L'elemento di un array può essere a sua volta un array.

### La dichiarazione

```
int     matrice[Dim2][Dim1];
```

- definisce la variabile **matrice** come un array di Dim2 elementi, dove ciascun elemento è un array di Dim1 elementi interi.
- alloca spazio in memoria (in locazioni fisiche contigue) adatto a contenere Dim1 x Dim2 interi.
- il nome dell'array (`matrice`) equivale a `&matrice[0]` (e a `matrice[0]`) e rappresenta l'indirizzo del primo byte di un array di Dim1 interi.
- `&matrice[0][0]` rappresenta l'indirizzo del primo byte allocato per la variabile matrice

La dichiarazione

```
typedef     int vett[Dim1];
vett        matrice[Dim2];
```

è equivalente a quella sopra vista

## ARRAY DI ARRAY - 2

**Rappresentazione in memoria** di un array a 2 dimensioni: mappa di memorizzazione

```
int  matrice [2][3];
```

| | |
|---|---|
| matrice[0] | valore di matrice[0][0] |
| | valore di matrice[0][1] |
| | valore di matrice[0][2] |
| matrice[1] | valore di matrice[1][0] |
| | valore di matrice[1][1] |
| | valore di matrice[1][2] |

**Accesso ai singoli elementi**

`matrice[i][j]` denota il valore del j-mo elemento dell'i-mo array di interi
Simili a righe (i) e colonne (j) e quindi nella mappa di memorizzazione varia più velocemente l'indice più a destra

Le notazioni

`&matrice[i][j]` e
`&matrice[0][0] + Dim1*i + j`

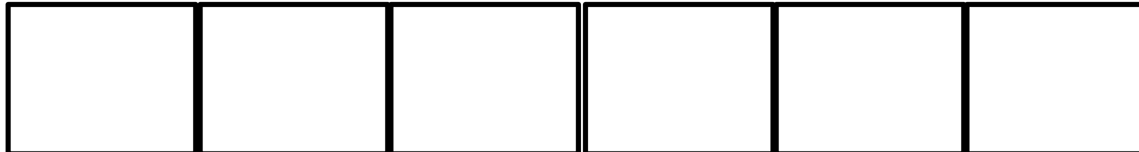
definiscono entrambe l'indirizzo del primo byte dell'elemento (i,j)

# Dimensions

**int num[6] = { 1, 2, 3, 4, 5, 6 };**

single-dimensioned array with initializer list

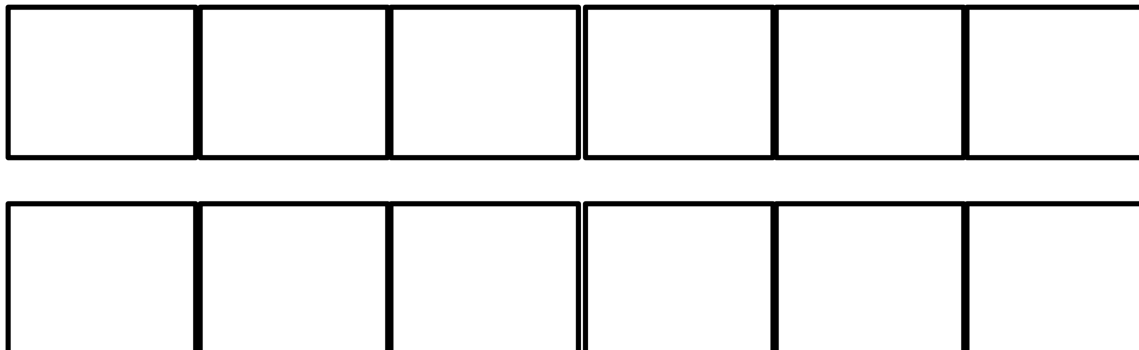
What is result?



**int num[2][6] = { { 1, 2, 3, 4, 5, 6 }, { 7, 8 } };**

multiple-dimensioned array (2 dimensions) with partial initializer list

What is result?

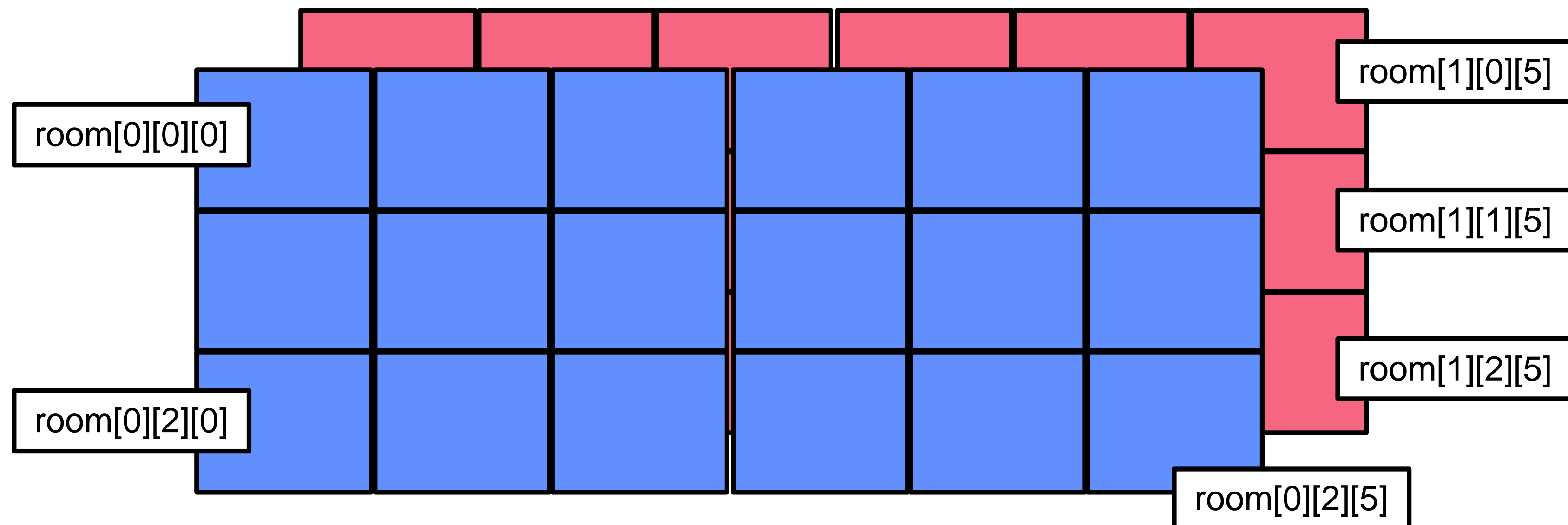


# Multidimensional Arrays

- **Subscripts match dimensions**

  int room[2][3][6];

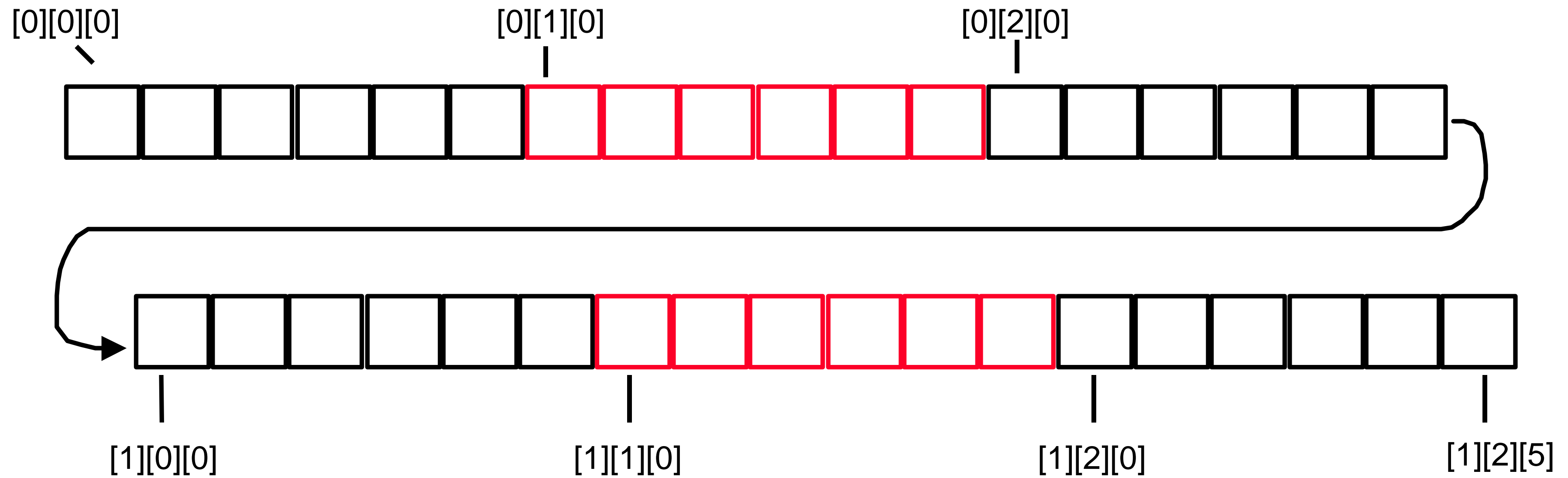
  can think of as 2 “sets” of 3 “rows” and 6 “columns”



# Multidimensional Arrays

- **Really stored in contiguous fashion**

  int room[2][3][6];




just convenient to think of as 2 “sets” of 3 “rows” and 6 “columns”!

**ESEMPIO3 -**
**CALCOLO PRODOTTO MATRICE PER VETTORE (mat*vett=vris)**

```
....................
int     vett[M], mat[N][M], vris[N];
int     i,j;
............................

    for(i=0;i<=N-1;i++)
    {
        vris[i]=0;
        for (j=0; j<=M-1;j++)
         vris[i]=vris[i]+mat[i][j]*vet[j];
    }
```

**ESEMPIO4 -**
**CALCOLO PRODOTTO MATRICE PER MATRICE (mat1*mat2=ris)**

```
....................
int     mat1[N][M], mat2[M][K], ris[N][K];
int     i,j,k;
............................

for(i=0;i<=N-1;i++)
 {
  for (k=0;k<=K,k++)
   {
    ris[i][k]=0;
    for (j=0; j<=M-1;j++)
     ris[i][k]=ris[i][k]+mat1[i][j]*mat2[j][k];
    }
 }
```