Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Gli Array

Andrea Piroddi - andrea.piroddi@unibo.it
Credit: Pietro Di Lena

Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration.

Stan Kelly-Bootle

Introduzione

- Gli array sono variabili di tipo **vettoriale**:
  - variabili scalari: contengono un solo valore (ex. `char`, `int`, `float`, `double`),
  - variabili vettoriali: contengono più valori dello stesso tipo (elementi).
- Gli **array**, così come le strutture, sono un tipo di dato **aggregato**:
  - strutture: permettono di *collezionare* un certo numero di dati di **tipo differente**.
  - array: permettono di *collezionare* un certo numero di dati dello **stesso tipo**.
- Gli array sono estremamente comodi nei casi in cui sia necessario avere a disposizione un alto numero di variabili dello stesso tipo.
  - Esempio: vogliamo leggere (da file o tastiera) 100 caratteri per poterli utilizzare successivamente in qualche modo. Potremmo dichiarare 100 variabili *distinte* di tipo `char` per contenere i caratteri letti (molto poco pratico), oppure potremmo utilizzare un *unico* array di 100 `char` (soluzione ideale).
- Tutti gli elementi di un array condividono lo stesso nome e sono contraddistinti unicamente da un *indice*, che ne indica la posizione all'interno dell'array.
- E' possibile accedere direttamente ad ogni valore all'interno dell'array utilizzando il relativo indice.
- Nel linguaggio C, l'organizzazione dei dati all'interno di un array rispecchia direttamente l'organizzazione *fisica* di tali dati in memoria centrale.

Dichiarazione di array

- La dichiarazione di un array è molto simile ad una dichiarazione di variabile. E' necessario specificare:
  1. **tipo** degli elementi nell'array,
  2. **nome** (identificatore) dell'array,
  3. **lunghezza** (espressione costante) dell'array, racchiusa tra parentesi quadre.

  `<Tipo> <Nome> [ <EsprCost> ];`
- La lunghezza dell'array **deve** essere un'espressione costante positiva.

```
1 #define N 10
2 struct S {int x; int y;};
3 typedef enum {T,F} bool;
4
5 char      A1[1];   // A1 var di tipo array di char, con 1 elemento
6 int       A2[5];   // A2 var di tipo array di int, con 5  elementi
7 bool      A3[N];   // A3 var di tipo array di bool, con 10 elementi
8 struct S  A4[2*N];// A4 var di tipo array di struct S, con 20 elementi
9 double    A5[F];   // A5 var di tipo array di double, con 1 elemento
```

- Lo standard ISO C89 **proibisce** lunghezze che non siano costanti positive.

```
1 const int C = 10;
2 double A6[C];  // Errore: C non    un'espressione costante
3 double A7[-1]; // Errore: costante negativa
4 double A8[0];  // Non espressamente proibito.
```

Lunghezza di un array

- La dichiarazione di un array è *formalmente* una definizione: alloca spazio in memoria.
- Un array di tipo `type` e lunghezza `len` richiede spazio di memoria in byte pari a

  `len * sizeof(type)`

- Possiamo ottenere il numero totale di byte usati dall'array con l'operatore `sizeof` applicato al nome dell'array.

```
1 int A[10];
2
3 printf("Occupazione di A = %lu byte\n",sizeof(A));
4 printf("Occupazione di A = %lu byte\n",10*sizeof(int));
```

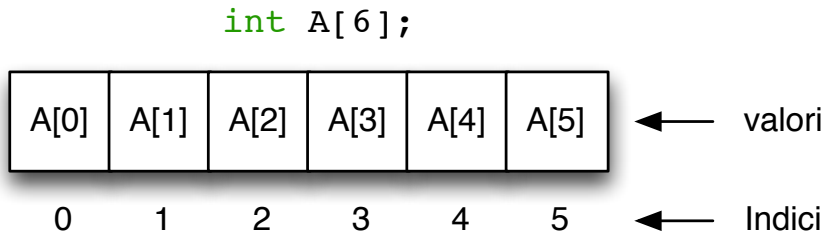
- E' preferibile definire la lunghezza di un array utilizzando una macro: dichiariamo nel codice un nome (identificatore) che rappresenti la lunghezza dell'array.
  - Una costante numerica ripetuta in diversi punti del codice viene generalmente chiamata **magic number**.
  - I magic number sono visti come segno di cattivo stile di programmazione.

```
1 #define N 100
2 int A1[N];
3 int A2[10]; // Lunghezza dichiarata con un magic number
```

Possiamo facilmente modificare e referenziare nel codice la lunghezza dell'array `A1` tramite la macro `N`. Dobbiamo esplicitamente indicare la costante 10 se vogliamo referenziare la lunghezza dell'array `A2`.

Accesso agli elementi di un array

- Gli elementi di un array sono disposti in locazioni di memoria successive e contigue.
- Ogni elemento di un array è **indicizzato** rispetto alla propria posizione.
- In C si inizia a contare da 0. Se l'array ha lunghezza N:
  1. il primo elemento ha indice 0,
  2. l'ultimo elemento ha indice N-1.



- Possiamo accedere agli elementi dell'array utilizzando valori di tipo intero.
- Ogni elemento è una variabile del tipo indicato nella dichiarazione dell'array.

```
1  #define N 6
2  int A[N], i;
3
4  // Inizializza i singoli elementi dell'array A
5  A[0]=0;           A[1]=1;
6  A[2]=A[0]+A[1]; A[3]=A[1]+A[2];
7  A[4]=A[2]+A[3]; A[5]; // A[5] non definito
8
9  // Stampa gli elementi dell'array A
10 for(i=0; i<N; i++) printf("Fib(%d)=%d\n",i,A[i]);
```

- Come per le variabili, il contenuto di un elemento non inizializzato è non definito.

Accesso agli elementi di un array: buffer overflow

- Cosa succede se cerchiamo di accedere ad un elemento che non appartiene all'array?

```
1 int A[2];
2
3 A[0]  = A[1]  = 0; // Indici nei limit
4 A[-1] = A[2]  = 0; // Indici fuori dai limiti
```

  Sintatticamente corretto anche se i due indici -1 e 2 *sforano* oltre il vettore `A`.
- La semantica dell'istruzione a riga 5 è ben definita (capiremo meglio quando parleremo di **aritmetica dei puntatori**), nonostante le conseguenze non siano predicibili.
  1. **Semantica**. Le istruzioni di assegnamento a riga 5 modificheranno il valore di qualche locazione di memoria che rispettivamente precede e segue la porzione di memoria riservata per l'array `A`. Termine tecnico: **buffer overflow**.
  2. **Comportamento**. Il comportamento di codice contenente istruzioni di accesso oltre i limiti di un array è non definito: l'eseguibile potrebbe andare in crash, funzionare correttamente o funzionare in modo erratico.
- Un esempio tipico (a causa di distrazione) di buffer overflow:

```
1 #define N 5
2 int i, A[N] = {1,2,3,4,5};
3
4 for(i=1; i<=N; i++) printf("%d\n",A[i]);
```

  Quando `i = N` non possiamo dire cosa stamperà la `printf()`.

Inizializzazione di un array

- La sintassi per l'inizializzazione di un array è la stessa vista per strutture e unioni.

```
1 #define N 3
2 int A[N] = {1,2,3};
3 // A[0]=1, A[1]=2, A[2]=3
```

- Se specifichiamo un numero di valori minore del numero di elementi nell'array, gli elementi *esclusi* sono inizializzati a zero.

```
1 #define N 5
2 int A[N] = {1,2,3};
3 // A[0]=1, A[1]=2, A[2]=3, A[3]=A[4]=0
```

- Se specifichiamo un numero di valori maggiore del numero di elementi nell'array, i valori in eccesso sono ignorati. Poco pulito.

```
1 #define N 2
2 int A[N] = {1,2,3};
3 // A[0]=1, A[1]=2
```

- Possiamo evitare di specificare la lunghezza dell'array se inizializziamo l'array al momento della dichiarazione. In questo caso, la lunghezza dell'array è determinata dal numero di elementi specificati tra le parentesi graffe.

```
1 int A[] = {1,2,3};  // sizeof(A)=3*sizeof(int)
2 // A[0]=1, A[1]=2, A[3]=3
```

Inizializzazione di un array: ulteriori caratteristiche

- E' ammesso non specificare la lunghezza dell'array unicamente quando la dichiarazione è provvista di una inizializzazione oppure se l'array è dichiarato `extern`.

```
1 int A1[] = {1,2,3}; // OK
2 int A2[];           // Errore di sintassi
3 extern int A3[];    // OK
```

- Come abbiamo già visto per le strutture e unioni, possiamo inizializzare un array facendo uso di una lista di valori solo al momento della dichiarazione.

```
1 #define N 3
2 int A1[N] = {1,2,3}; // OK
3 int A2[N];
4 A2 = {1,2,3};        // Errore di sintassi
```

- Come possiamo *calcolare* il numero di elementi in un array dichiarato senza specificare la lunghezza? Ricordiamo che l'operatore `sizeof` ci permette di ottenere la lunghezza in byte dell'array.

```
1 #define NUM_ELEM(X) (sizeof(X)/sizeof(X[0]))
2
3 int A[] = {1,2,3};
4
5 printf("Occupazione totale di A          : %lu byte\n",sizeof(A));
6 printf("Occupazione di un elemento in A : %lu byte\n",sizeof(A[0]));
7 printf("Numero di elementi in A          : %lu\n",NUM_ELEM(A));
```

Esempi di utilizzo di array 1/2

- Un array ci pemette di collezionare una serie di valori dello stesso tipo. Possiamo semplificare la struttura di un programma utilizzando un array invece di definire un numero elevato di variabili dello stesso tipo.

```
1  // Legge una sequenza di massimo N caratteri e la stampa invertita
2  #include <stdio.h>
3
4  #define N 100
5
6  int main() {
7   char c, buf[N];
8   int  i;
9
10  // Legge massimo N caratteri e li salva in buf
11  for(i=0; i<N && (c=getchar())!='\n'; i++)
12   buf[i]=c;
13
14  // Stampa i caratteri letti in ordine inverso
15  for(i = i-1; i >= 0; i--)
16   printf("%c",buf[i]);
17  printf("\n");
18  return 0;
19 }
```

Da notare l'inizializzazione dell'indice `i` a riga 15. A termine del ciclo a riga 11:

1. se `i` = `N` $\Rightarrow$ `A[i]` è una posizione al di fuori dell'array.
2. se `i` = numero di caratteri letti + 1 $\Rightarrow$ `A[i]` contiene un valore non definito.

Esempi di utilizzo di array 2/2

- Possiamo interpretare gli indici in vari modi.

```
1  // Stampa quali e quanti caratteri ci sono in un testo letto da
2  // tastiera
3  #include <stdio.h>
4
5  #define N 128 // Numero di caratteri ascii a 7bit
6  #define S 32  // Consideriamo solo caratteri stampabili
7
8  int main() {
9   int c,code[N]={0}; // Inizializziamo l'array a 0
10
11  // Contiamo quante volte un carattere viene letto
12  while((c=getchar())!=EOF)
13   if(c>=0 && c<N) code[c]++;
14
15  // Stampiamo quante volte abbiamo letto un carattere
16  for(c=S; c<N; c++)
17   if(code[c]>0) // Ignoriamo i caratteri non letti
18    printf("Code: %3d Symbol: %c Times: %d\n",c,c,code[c]);
19  return 0;
20 }
```

- Nell'esempio, abbiamo un array che contiene un elemento per ogni carattere ascii.
- Nell'array un carattere è indicizzato dal proprio codice ascii.
- La costante `EOF` è una macro che indica la fine del testo.
- Il valore della costante `EOF` dipende dall'implementazione (tipicamente -1).

Array multidimensionali

- Un array può essere definito con un qualsiasi numero di dimensioni.

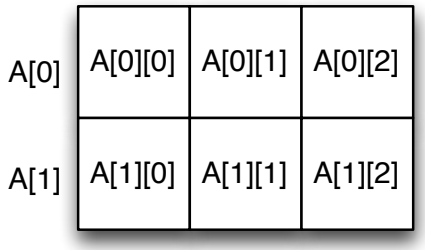
  `<Tipo> <Nome> [ <EsprCost1> ] [ <EsprCost2> ] .. [ <EsprCostN> ];`

- Possiamo definire molto facilmente matrici, cubi e, in generale, array multidimensionali.

```
1 int matrix[2][3];        // Matrice con due righe e 3 colonne
2 int cube[3][3][3];       // Cubo di lato 3
3 int array[2][3][4][5]; // Array multidimensionale
```

- Un array bidimensionale è in realtà un array di array. Allo stesso modo, un array tridimensionale è un array di array bidimensionali e così via.

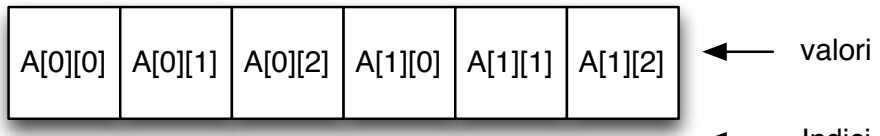
`int A[2][3];`



Gli elementi `A[0]` e `A[1]` di `A` sono array (monodimensionali) di lunghezza 3.

- Anche gli array multidimensionali sono rappresentati sequenzialmente in memoria: gli elementi nella prima riga sono seguiti dagli elementi nella seconda riga, e così via.

`int A[2][3];`



Inizializzazione di array multidimensionali

- L'inizializzazione di array multidimensionali segue all'incirca le stesse regole già viste per l'inizializzazione di array monodimensionali strutture e unioni.

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

- Possiamo omettere le parentesi graffe più interne. In tal caso, gli elementi sono inizializzati in ordine di memorizzazione. Meglio evitare.

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

- Valgono le stesse regole per l'inizializzazione di elementi non specificati. Possiamo non specificare elementi da inizializzare per singola riga, non per colonna.

```
1 int mat1[2][3] = {1,2};
2 // 1 2 0
3 // 0 0 0
4 int mat2[2][3] = {{1},{2}};
5 // 1 0 0
6 // 2 0 0
7 int mat3[2][3] = {0};
8 // 0 0 0
9 // 0 0 0
```

Inizializzazione di array multidimensionali: ulteriori caratteristiche

- Possiamo omettere la lunghezza della prima dimensione se inizializziamo l'array multidimensionale al momento della dichiarazione.

```
1 int mat1[][3];                              // Errore
2 int mat2[][3] = {{1,2,3},{4,5,6}}; // OK
3 // 1 2 3
4 // 4 5 6
```

- Possiamo unicamente omettere la prima lunghezza.

```
1  // 1 2   5 6
2  // 3 4   7 8
3  int mat1[ ][ ][2] = {{{1,2},{3,4}},{{5,6},{7,8}}}; // Errore
4  int mat2[2][ ][2] = {{{1,2},{3,4}},{{5,6},{7,8}}}; // Errore
5  int mat3[2][2][ ] = {{{1,2},{3,4}},{{5,6},{7,8}}}; // Errore
6  int mat4[ ][2][2] = {{{1,2},{3,4}},{{5,6},{7,8}}}; // OK
7  int mat5[2][2][2] = {{{1,2},{3,4}},{{5,6},{7,8}}}; // OK
8  // 1 2 0    5 6 0
9  // 3 4 0    7 8 0
10 // 0 0 0    0 0 0
11 int mat6[ ][3][3] = {{{1,2},{3,4}},{{5,6},{7,8}}}; // OK
```

- E' permesso omettere l'inizializzazione se la dichiarazione è di tipo `extern`.

```
1 extern int mat[][3]; // OK
```

La dichiarazione specifica l'esistenza di una variabile di tipo array di `int`, la cui effettiva dimensione di memoria è specificate in qualche altro posto.

Indicizzazione di array multidimensionali

- Ogni elemento in un array *n*-dimensionale è indicizzato da *n* indici distinti.
- Abbiamo bisogno, ad esempio, di due indici distinti e di due cicli annidati per stampare il contenuto di un array bidimensionale.

```
1  #define M 2
2  #define N 3
3
4  int mat[M][N] = {{1,2,3},{4,5,6}};
5  int i,j;
6
7  for(i=0; i<M; i++) {
8    for(j=0; j<N; j++)
9      printf("%d ",mat[i][j]);
10   printf("\n");
11 }
```

- Gli indici devono essere *protetti* da parentesi quadre.

```
1  int mat[2][3] = {{1,2,3},{4,5,6}};
2
3  printf("%d\n",mat[0][1]); // OK
4  printf("%d\n",mat[0, 1]); // Equivale a mat[1]
```

L'espressione `mat[0,1]` fa uso dell'operatore virgola ed è equivalente a `mat[1]`, che indica il secondo array monodimensionale di `mat`. L'istruzione a riga 4 è sintatticamente valida ma il risultato della stampa è non definito.

Occupazione di memoria di array multidimensionali

- Così come abbiamo visto per gli array monodimensionali, l'occupazione di memoria in byte di un array multidimensionale può essere ottenuta utilizzando l'operatore `sizeof`.

```c
#define M 2
#define N 3

int mat[M][N];

printf("Occupazione di mat    = %lu byte\n",sizeof(mat));
printf("Occupazione di mat    = %lu byte\n",M*N*sizeof(int));
printf("Occupazione di mat[0] = %lu byte\n",sizeof(mat[0]));
printf("Occupazione di mat[1] = %lu byte\n",sizeof(mat[1]));
```

- Come possiamo *calcolare* il numero di righe, colonne ed elementi di un array bidimensionale dichiarato in modo *incompleto*?

```c
#define NUM_ELEM(X) (sizeof(X)/sizeof(X[0][0]))
#define NUM_ROWS(X) (sizeof(X)/sizeof(X[0]))
#define NUM_COLS(X) (sizeof(X)/(sizeof(X[0][0])*NUM_ROWS(X)))

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

printf("Occupazione totale di A          : %lu byte\n",sizeof(A));
printf("Occupazione di un elemento in A : %lu byte\n",sizeof(A[0][0]));
printf("Numero di elementi in A          : %lu\n",NUM_ELEM(A));
printf("Numero di righe di A             : %lu\n",NUM_ROWS(A));
printf("Numero di colonne di A           : %lu\n",NUM_COLS(A));
```

Esempi di utilizzo di array multidimensionali

- Gli array multidimensionali sono estremamente indicati per implementare facilmente operazioni di calcolo matematico, come ad esempio la moltiplicazione di matrici.

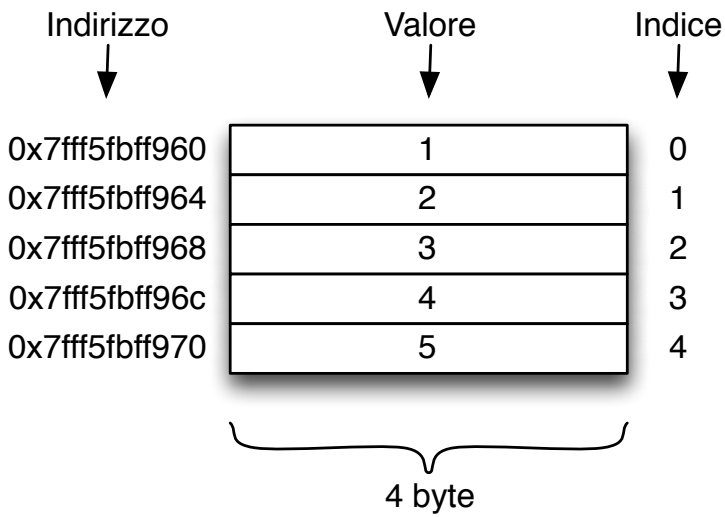
```c
#include <stdio.h>

#define M 2
#define N 2
#define K 2

int main() {
 int M1[M][K]={{1,-2},{-3,4}};
 int M2[K][N]={{-1,2},{3,-4}};
 int M3[M][N]={{0}};
 int row,col,k;

 // Prodotto riga per colonna di M1 e M2
 for(row=0; row<M; row++)
  for(col=0; col<N; col++)
   for(k=0; k<K; k++)
    M3[row][col] += M1[row][k]*M2[k][col];
 // Stampa del prodotto
 for(row=0; row<M; row++) {
  for(col=0; col<N; col++)
   printf("%d ",M3[row][col]);
  printf("\n");
 }
 return 0;
}
```

Rappresentazione in memoria di array

- Per poter comprendere meglio come utilizzare gli array dobbiamo entrare nei dettagli di rappresentazione di un array in memoria.
- Ricordiamo che:
  - utilizziamo l'operatore `&` per ottenere l'indirizzo di memoria di una variabile;
  - stampiamo un indirizzo di memoria utilizzando lo specificatore di formato `%p`.

```
1 #define N 5
2 int i, A[N]={1,2,3,4,5};
3
4 for(i=0; i<N; i++)
5   printf("Indirizzo: %p Valore: %d Indice: %d\n",&A[i],A[i],i);
```

Se `sizeof(int)` = 4 byte, il vettore `A` nell'esempio potrebbe avere la seguente rappresentazione.



Rappresentazione in memoria di array: il nome dell'array

- In C il nome di un array è utilizzato come sinonimo dell'**indirizzo di memoria** a partire dal quale l'array è memorizzato
- Tale indirizzo coincide con l'**indirizzo di memoria del primo elemento** dell'array.

```
1 int  A[]={1,2,3,4,5};
2 printf("Indirizzo di A    : %p\n",A);     // Indirizzo di A
3 printf("Indirizzo di A[0] : %p\n",&A[0]);// Indirizzo primo elemento di A
```

- Differenze tra nome di array e nome di variabile scalare:
  1. Un riferimento nel codice al nome di un array rappresenta un riferimento non al suo contenuto ma all'indirizzo di memoria del primo elemento dell'array.
  2. Il contenuto di una variabile di tipo array (il proprio indirizzo di memoria) non può essere modificato.
- Possiamo quindi vedere una variabile di tipo array come una variabile di tipo `const` (a sola lettura) contenente un indirizzo di memoria.

```
1 int  A1[] = {1,2,3,4,5};
2 int  A2[] = {5,4,3,2,1};
3
4 A1 = 0;  // Errore: non possiamo assegnare un valore
5
6 A1 = A2; // Errore: non possiamo copiare il contenuto
7          // di un array con un singolo assegnamento
```

Rappresentazione in memoria di array multidimensionali

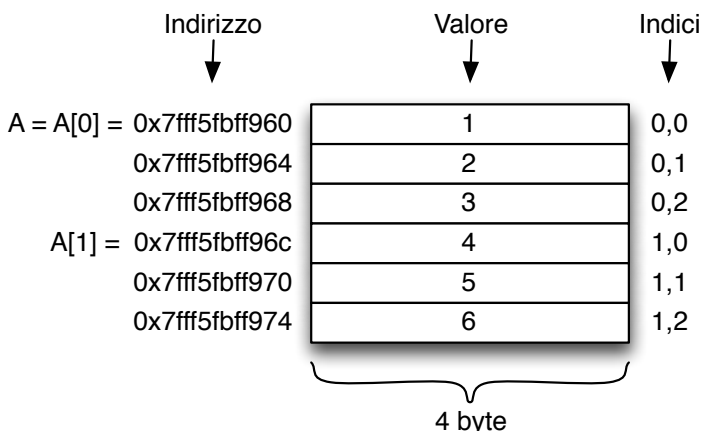
- Possiamo adesso a capire meglio come sono rappresentati gli array multidimensionali.
- Un riferimento ad un valore nella prima dimensione dell'array è un riferimento all'indirizzo di memoria a partire dal quale è memorizzato il sotto-array.

```
1 int  A[2][3] = {{1,2,3},{4,5,6}};
2 printf("Ind. di A       : %p\n",A);       // Ind. di A
3 printf("Ind. di A[0]    : %p\n",A[0]);    // Ind. primo sotto-array di A
4 printf("Ind. di A[0][0] : %p\n",&A[0][0]);// Ind. primo elemento di A
```

Le tre `printf()` stamperanno lo stesso indirizzo di memoria.

- Il seguente codice stampa *in ordine* gli indirizzi di memoria di un array bidimensionale.

```
1 int  i, j, A[2][3] = {{1,2,3},{4,5,6}};
2 for(i=0; i<2; i++)
3   for(j=0; j<3; j++)
4     printf("Ind: %p Val: %d Indici: %d,%d\n",&A[i][j],A[i][j],i,j);
```



Operatori con il tipo di dato array (cenni introduttivi)

- Esattamente come abbiamo visto per le strutture ed unioni, non tutti gli operatori possono essere utilizzati su un tipo di dato array: le operazioni ammissibili dipendono dal **tipo** dell'array.
- Rimarchiamo la seguente distinzione:

```
int A[2][3];
```

  - `A` è un'espressione di **tipo array di array di** `int` (contiene 2 array di `int`).
  - `A[0]` è un'espressione di **tipo array di** `int` (contiene 3 `int`).
  - `A[0][0]` è un'espressione di **tipo** `int` (è un `int`).
- Nonostante le espressioni `A`, `A[0]` e `&A[0][0]` valgano lo stesso indirizzo di memoria, il tipo a loro associato è profondamente differente.
- Le operazioni che possiamo effettuare sui tipi di dato array coincidono con le operazioni che il linguaggio C permette su tipi di dato che contengono indirizzi di memoria.
- Vedremo questi aspetti quando parleremo di puntatori (**aritmetica dei puntatori**).
- Diversamente, le operazioni che possiamo effettuare sui singoli elementi di un array dipendono unicamente dal tipo di tali elementi.

Array di strutture

- Possiamo definire array per tutti i tipi di dato visti finora.
- Gli array di strutture sono tipi di dato particolarmente comodi per poter organizzare facilmente grosse quantità di dati.

```
1  #define N 100
2  // Struttura contenente 2 coordinate cartesiane
3  struct point { int x; int y; };
4
5  // Array contenente 100 coordinate cartesiane 2D
6  struct point coord[N];
7
8  // Inizializza l'array con 100 punti del segmento
9  // [0,100) sull'asse delle ascisse: (0,0), (1,0), ..
10 int i;
11 for(i=0; i<N; i++) {  coord[i].x = i; coord[i].y = 0; }
```

- Possiamo creare un nuovo nome per un tipo di dato array di strutture la `typedef`.

```
1  #define N 100
2
3  // points: tipo di dato array contentente 100 strutture
4  typedef struct point {int x; int y;} points[N];
5
6  // coord: variabile di tipo points
7  points coord;
```

Il tipo della variabile `coord` non cambia nei due esempi.

Strutture contenenti array

- Possiamo definire strutture contenenti array. Vediamo cosa cambia se riscriviamo l'esempio nella slide precedente sfruttando questa nuova possibilità.

```
1  #define N 100
2  // Struttura contenente 2 array con 100 elementi di tipo int
3  struct points { int x[N]; int y[N]; };
4
5  // Variabile di tipo struct points
6  struct points coord;
7
8  // Inizializza la variabile con 100 punti del segmento
9  // [0,100) sull'asse delle ascisse: (0,0), (1,0), ..
10 int i;
11 for(i=0; i<N; i++) {  coord.x[i] = i; coord.y[i] = 0; }
```

- Questo esempio è molto simile all'esempio nella slide precedente: la nuova organizzazione dei dati ci permette di memorizzare lo stesso tipo di informazione.
- Cambiano completamente le modalità di accesso ai dati: confrontare riga 11 in questo esempio con riga 11 nella slide precedente.
- C'è una seconda differenza estremamente rilevante tra i due esempi:
  1. il riferimento al nome della variabile `coord` in questa slide è un riferimento all'intero **contenuto** della struttura `struct points`;
  2. Il riferimento al nome della variabile `coord` nella slide precedente è un riferimento ad un **indirizzo** di memoria.

Strutture contenenti array: caratteristica principale

- Un array definito in una struttura può essere ricopiato con un solo assegnamento.

```c
#define N 100

// Tipo di dato struttura contenente un array con 100 elementi int
typedef struct { int vect[N];  } vect;

vect V1 = {0}; // Dichiarazione con inizializzazione
vect V2;       // Dichiarazione senza inizializzazione

V2 = V1; // Legale: ricopiamo interamente l'array .vect[]
```

- Non possiamo fare lo stesso se lavoriamo direttamente su tipi di dato array.

```c
#define N 100

// Tipo di dato array con 100 elementi int
typedef int vect[N];

vect V1 = {0}; // Dichiarazione con inizializzazione
vect V2;       // Dichiarazione senza inizializzazione

V2 = V1; // Errore: non possiamo assegnare un valore a V2
```

- Nonostante questa caratteristica appaia molto comoda, in realtà è fortemente sconsigliata se vogliamo utilizzare tipi di dato array con le funzioni.

Array e funzioni

- Le funzioni non possono restituire un tipo di dato array

```
1 int [ ] func1(void); // Errore
2 int [2] func2(void); // Errore
```

- Le funzioni possono prendere come argomenti tipi di dato array.

```
1 void func1(int vect[ ]); // OK
2 void func2(int vect[2]); // OK
```

- Come viene trattato dalla funzione un parametro formale di tipo array? Che cosa stampa la seguente funzione, se assumiamo che `sizeof(int)` $= 4$?

```
1  #include <stdio.h>
2
3  void func(int v[]) {
4    printf("Indirizzo: %p\n",v);
5    printf("Dimensione: %lu\n",sizeof(v));
6  }
7
8  int main() {
9    int vect[100];
10   printf("Indirizzo: %p\n",vect);
11   printf("Dimensione: %lu\n",sizeof(vect));
12   func(vect);
13   return 0;
14 }
```

Array come parametri formali di funzioni 1/4

- Ricordiamo che il C adotta il passaggio dei parametri per valore.
- Ricordiamo che Il nome di un array è sinonimo dell'indirizzo di memoria del primo elemento dell'array.
- Dal punto di vista di una funzione, un parametro formale di tipo array è una variabile locale contenente un indirizzo di memoria (**decadimento** da array a *puntatore*).
- Quindi, la `sizeof()` applicata ad un parametro formale di tipo array ritorna la dimensione in byte di una variabile contenente un indirizzo di memoria (puntatore).
- Il contenuto del parametro formale può essere modificato all'interno della funzione.

```
1 void func(int v[]) {
2   v = 0; // Legale
3 }
```

Con l'assegnamento a riga 2, la funzione `func()` *perde* l'indirizzo di memoria del primo elemento dell'array passato in input.

- Conseguenza: una funzione **non conosce** la lunghezza di un tipo di dato array passato come parametro attuale ma solo l'indirizzo di memoria del suo primo elemento.

Array come parametri formali di funzioni 2/4

- Per poter definire correttamente funzioni che abbiano array come parametri è necessario fornire alla funzione la lunghezza dell'array, oltre al suo indirizzo di memoria.
- Non è sufficiente indicare la lunghezza dell'array nella parte dichiarativa della funzione.

```c
#include <stdio.h>

// Funziona correttamente solo con array di lunghezza
// esattamente 100
void func(int v[100]) {
  int i;
  for(i=0; i<100; i++)
    printf("%d\n",v[i]);
}

int main() {
  int vect[] = {1,2};

  // Sintatticamente corretto ma causa buffer overflow
  func(vect);
  return 0;
}
```

Tramite il parametro formale `v`, la funzione `func` accede alle locazioni di memoria sullo stack che seguono (indirizzi alti) la posizione in cui `v` è memorizzato. Non è predicibile quali valori saranno stampati dalla `printf()`.

Array come parametri formali di funzioni 3/4

- La parte dichiarativa della funzione deve contenere un ulteriore parametro che specifichi la lunghezza del vettore.

```
1  #include <stdio.h>
2
3
4  // Funziona correttamente, se invocata correttamente
5  void func(int v[], int n) {
6    int i;
7    for(i=0; i<n; i++)
8      printf("%d\n",v[i]);
9  }
10
11 int main() {
12   int vect[] = {1,2};
13
14   func(vect, 2);    // OK
15   func(vect, 100); // Buffer overflow
16   return 0;
17 }
```

Questa implementazione gestisce correttamente array di qualsiasi lunghezza. Spetta al chiamate invocare la funzione con i parametri appropriati.

Array come parametri formali di funzioni 4/4

- A differenza di quanto visto finora, le funzioni con array come parametri formali sono un primo esempio di come sia possibile *simulare* il **passaggio per riferimento** in C.
- Definizioni:
  - **passaggio per valore**: la funzione riceve una copia del parametro.
  - **passaggio per riferimento**: la funzione riceve un riferimento al parametro.
- All'interno di una funzione possiamo accedere direttamente alla locazione di memoria in cui risiede l'array, esterno alla funzione.
- Possiamo quindi modificare il contenuto di locazioni di memoria che non risiedono nel record di attivazione della funzione chiamata.

```
1  // Setta ad init i valori degli elementi nell'array
2  void initialize(int vect[], int n, int init) {
3    int i;
4    for(i=0; i<n; i++) vect[i]=init;
5  }
6
7  int main() {
8    int vect[100];
9    // Setta ad 1 tutti gli elementi dell'array locale vect
10   initialize(vect, 100, 1);
11
12   return 0;
13 }
```

Array multidimensionali come parametri formali di funzioni

- Il passaggio di array multidimensionali a funzione introduce un ulteriore livello di complicazione.
- Innanzitutto, abbiamo un'ulteriore regola sintattica:
  - la lunghezza della prima dimensione è opzionale,
  - le lunghezze delle dimensioni successive alla seconda devono essere indicate.

```
1 void func1(int mat[][ ], int n); // Errore di sintassi
2 void func2(int mat[][3], int n); // OK
```

- La lunghezza delle dimensioni successive alla prima è necessaria per permettere al compilatore di calcolare correttamente gli indirizzi di memoria dei vari elementi nell'array multidimensionale.
- La funzione `func2()` nell'esempio è *definita* per qualsiasi tipo di array bidimensionale che abbia esattamente tre colonne.
- In C **non è possibile** in nessun modo definire **correttamente** una funzione che prenda in input un array multidimensionale di dimensioni *non specificate*!
- Come di consueto, il C non è troppo rigido: il compilatore permette di invocare la funzione `func2()` anche con array di dimensione non compatibile. Il compilatore ci avvisa della discrepanza con un warning.

Array multidimensionali come parametri formali di funzioni: esempio 1/2

- La lunghezza delle dimensioni superiori alla prima è necessaria per poter calcolare correttamente l'offset delle righe dell'array multidimensionale.

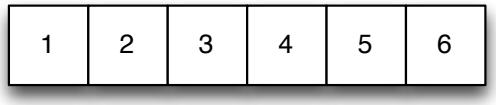
```c
#include <stdio.h>

void func1(int A[][3], int n) {
  int i;
  for(i=0; i<n; i++)
    printf("A[%d][0] = %d\n",i,A[i][0]);
}

void func2(int A[][2], int n) {
  int i;
  for(i=0; i<n; i++)
    printf("A[%d][0] = %d\n",i,A[i][0]);
}

int main() {
  int A[3][2] = {{1,2},{3,4},{5,6}};

  func1(A,3); // Warning
  printf("---\n");
  func2(A,3); // OK

  return 0;
}
```

`func1()` e `func2()` differiscono solo nella parte dichiarativa: cosa stampano?

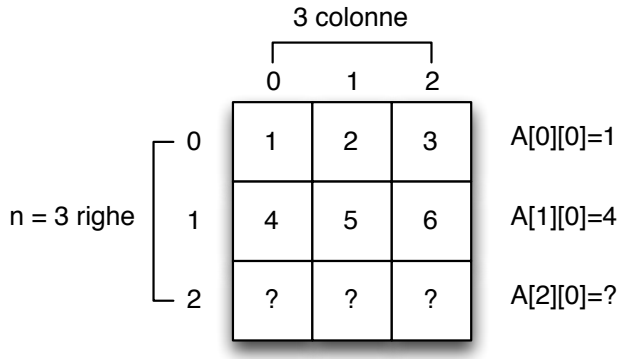
Array multidimensionali come parametri formali di funzioni: esempio 2/2

- L'array multidimensionale `A` è rappresentato sequenzialmente in memoria.

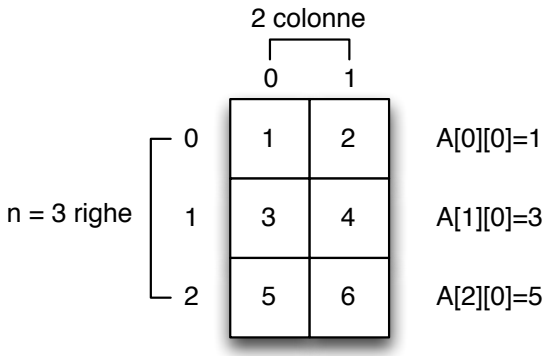
```
int A[3][2] = {{1,2},{3,4},{5,6}};
```



- La funzione `func1()` si aspetta in input array multidimensionali che abbiano esattamente 3 colonne. Il numero di righe è indicato dal parametro formale `n`.



- La funzione `func2()` si aspetta in input array multidimensionali che abbiano esattamente 2 colonne. Il numero di righe è indicato dal parametro formale `n`.



Array e funzioni: considerazioni finali

1. Il passaggio del *tipo di dato array* tramite indirizzo è estremamente **conveniente**:
   - una funzione può operare su array di grandi dimensioni senza dover pagare l'over-head necessario ad effettuare una copia dell'array nel record di attivazione;
   - se si adottasse la strategia del passaggio per valore, non sarebbe possibile gestire *efficientemente* array di grandi dimensioni come parametri di funzioni;
   - è bene evitare di definire funzioni che prendano come parametri strutture contenenti array *ingombranti*: il passaggio della struttura avviene per valore.
2. Le funzioni che hanno *tipi di dato array* come parametri formali o variabili locali sono potenzialmente **insicure**, in quanto suscettibili a problemi di buffer overflow:
   - il linguaggio C non fornisce protezioni relativamente alle istruzioni che causano buffer overflow;
   - le più comuni e *classiche* tecniche di attacco (hacking) sfruttano debolezze legate a problemi di buffer overflow nel codice;
   - è compito del programmatore evitare di scrivere codice potenzialmente insicuro.
3. La gestione del *tipo di dato array multidimensionale* come parametro di funzione è estremamente **scomoda** e **limitata**:
   - non c'è modo di definire funzioni che prendano in input array multidimensionali *generici*, le cui dimensioni non siano quindi fissate *a-priori*;
   - la memoria *dinamica* ci viene in soccorso (vedremo come).

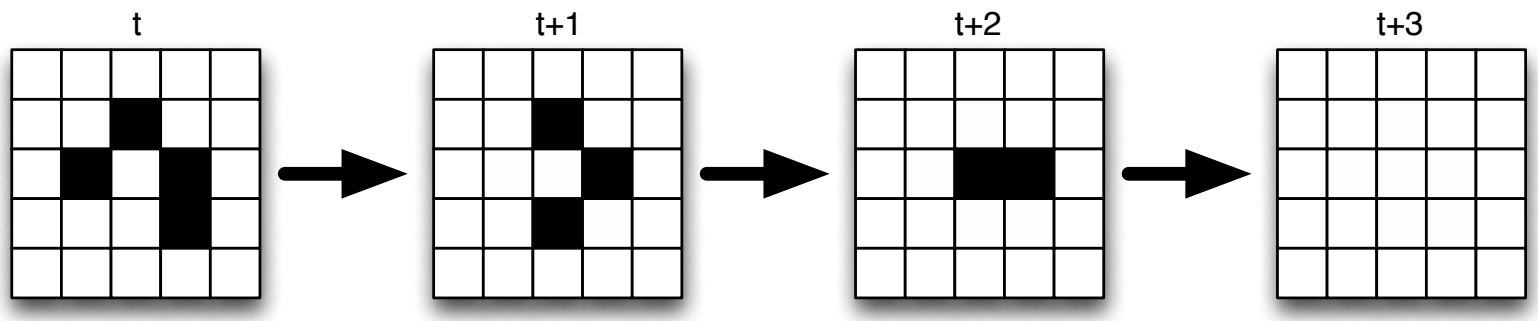
The Game of Life

- Il Game of Life è un **automa cellulare** sviluppato e studiato dal matematico John Conway (1970).
  - Gli automi cellulari sono un formalismo matematico, spesso utilizzato per modellare **sistemi complessi** (sistemi il cui comportamento *globale* emerge come conseguenza di di semplici interazioni locali).
  - Lo scopo dell'automa cellulare Game of Life è quelli di mostrare come comportamenti di tipo *biologico* possano emergere da semplici regole di interazione.
- E' stato dimostrato che Game of Life è un modello Turing-completo:
  - Ad esempio, è possibile implementare in Game of Life un qualsiasi algoritmo implementabile in C.
  - Il fatto che sia possibile implementare in Game of Life un qualsiasi algoritmo non implica che sia semplice farlo!
- Il Game of Life non è un vero e proprio gioco: non ci sono giocatori.
- Il sistema parte da una configurazione iniziale ed *evolve* verso configurazioni complesse oppure banali.
- Alcuni pattern speciali, chiamati **glider**, sono in grado di *viaggiare* attraverso l'intero *mondo*.

The Game of Life: descrizione

- Il *gioco* si svolge su una matrice bidimensionale (universo del gioco).
- In ogni istante di tempo ogni cella della matrice può essere in uno tra due stati possibili: **vivo** o **morto**.
- Le condizioni che determinano se una cella sarà viva o morta al tempo $t + 1$ dipendono unicamente dallo stato delle 8 celle immediatamente adiacenti al tempo t.

1. Ogni cella viva con meno di due vicini vivi muore (sotto-popolazione)
2. Ogni cella viva con due o tre vicini vivi sopravvive alla generazione successiva (equilibrio)
3. Ogni cella viva con più di tre vicini vivi muore (sovrappopolazione)
4. Ogni cella morta con esattamente tre vicini diventa viva nella generazione successiva (nuove nascite).



- Si parte da una configurazione iniziale in cui alcune celle sono vive e altre morte.
- Le regole di aggiornamento dello stato vengono applicate iterativamente fintanto che non si raggiunge una situazione di stallo (configurazione periodica).

The Game of Life: implementazione

- Possiamo implementare facilmente un *simulatore* del Game of Life facendo uso di matrici bidimensionali.
- E' sufficiente implementare:
  - una funzione per l'aggiornamento della configurazione corrente `update()`,
  - una funzione per la stampa della configurazione corrente `print()`,
  - una funzione per inizializzare una configurazione di partenza `initialize()`.
- La matrice che rappresenta l'*universo di gioco* è nascosta nel file di libreria.
- Vediamo unicamente come è implementabile la funzione di aggiornamento.

```
1  // gol.h
2
3  // Aggiorna la configurazione corrente
4  void update(void);
5
6  // Stampa la configurazione corrente
7  void print(void);
8
9  // Inizializza la matrice di partenza,
10 // caricando la configurazione da file
11 void initialize(char filename[]);
```

The Game of Life: la funzione di aggiornamento 1/2

- La funzione di aggiornamento necessita di due matrici di uguale dimensione: una matrice contiene la configurazione attuale, la seconda matrice viene utilizzata come spazio temporaneo per calcolare la configurazione all'istante successivo.
- Potremmo utilizzare una sola matrice, al costo di complicare l'implementazione

```
1  // gol.c
2  #define M 100 // Righe
3  #define N 100 // Colonne
4
5  static unsigned short int W[M][N]={{0}}; // Mondo del gioco
6  static unsigned short int T[M][N]={{0}}; // Spazio temporaneo
7
8  static short int cell_status(int i, int j);
9
10 void update(void) {
11   int i,j;
12   // Aggiorna le celle e salva nella matrice T
13   for(i=0; i<M; i++)
14     for(j=0; j<N; j++)
15       T[i][j] = cell_status(i,j);  // Determina se i,j vive o muore
16
17   // Ricopia la matrice T in W
18   for(i=0; i<M; i++)
19     for(j=0; j<N; j++)
20       W[i][j] = T[i][j];
21 }
```

The Game of Life: la funzione di aggiornamento 2/2

- Ai bordi, il mondo può essere interpretato come
  - **toroidale** (soluzione scelta): una cella ai bordi è adiacente al lato opposto
  - **quiescente**: le celle ai bordi si assumono tutte morte

```
1  // gol.c
2
3  // Aggiornamento con mondo toroidale:
4  static int cell_status(int i, int j) {
5    int a,b, x=0;
6    /*
7      Sia k un intero tale che -1<=k<=N. L'espressione (k+N)%N
8      restituisce:
9      1) k    se 0<=k<N
10     2) 0    se k=N
11     3) N-1  se k=-1
12   */
13   for(a=-1; a<=1; a++)
14     for(b=-1; b<=1; b++)
15       if((a != 0) || (b != 0))
16         x += W[(i+a+M)%M][(j+b+N)%N];
17
18   if(W[i][j]==1 && x<2)              return 0; // Caso 1
19   if(W[i][j]==1 && (x==2 || x==3)) return 1; // Caso 2
20   if(W[i][j]==1 && x>3)              return 0; // Caso 3
21   if(W[i][j]==0 && x==3)             return 1; // Caso 4
22   return 0;
23 }
```