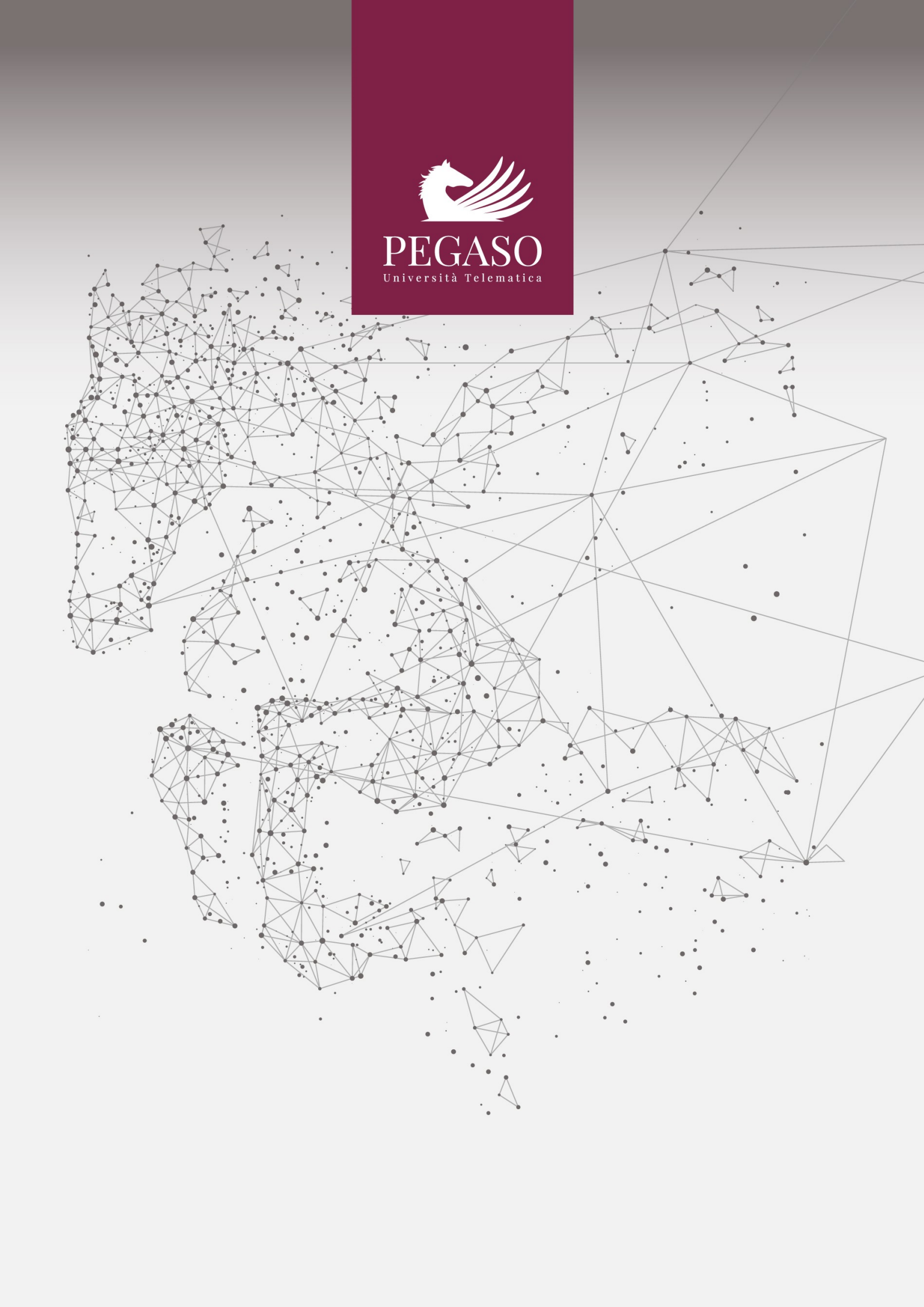

PEGASO
Università Telematica

# Indice

# 1. Array locali statici e array locali automatici

Ricordiamo che una variabile locale `static` esiste per la durata del programma, ma è visibile soltanto nel corpo di una funzione.

Possiamo applicare lo specificatore `static` alla definizione di un array locale in modo che l'array non debba essere creato e inizializzato ogni volta che la funzione è chiamata e che non venga distrutto ogni volta che si esce dalla funzione nel programma.

Questo riduce il tempo di esecuzione del programma, soprattutto per i programmi con funzioni chiamate frequentemente e che contengono array grandi.

☺ *Prestazioni*

*In funzioni che contengono array automatici e che vengono attivate e terminate frequentemente, rendete gli array* `static`

*In questo modo, non dovranno essere creati a ogni chiamata della funzione.*

Gli array che sono `static` sono inizializzati una volta per tutte all'avvio del programma. Se non inizializzate esplicitamente un array `static`, gli elementi di quell'array vengono automaticamente inizializzati a zero.

Il seguente programma illustra la funzione `staticArrayInit` (righe 21–39) con un array locale `static` (riga 24) e la funzione `automaticArrayInit` (righe 42–60) con un array locale automatico (riga 45).

La funzione `staticArrayInit` è chiamata due volte (righe 12 e 16).

L'array locale `static` nella funzione è inizializzato a zero prima dell'avvio del programma (riga 24).

La funzione stampa l'array, aggiunge 5 a ogni elemento e stampa di nuovo l'array.

La seconda volta che la funzione è chiamata, gli elementi dell'array hanno mantenuto il valore 5.

```
1  // Gli array statici
2  //  sono automaticamente inizializzati a zero
3  #include <stdio.h>
4
5  void staticArrayInit(void); // prototipo di funzione
6  void automaticArrayInit(void); // prototipo di funzione
7
8  // la funzione main inizia l'esecuzione del programma
9  int main(void)
10 {
11    puts("First call to each function:");
12    staticArrayInit();
13    automaticArrayInit();
14
15    puts("\n\nSecond call to each function:");
16    staticArrayInit();
17    automaticArrayInit();
18 }
19
20 // funzione usata per illustrare un array locale statico
21 void staticArrayInit(void)
22 {
23    //  inizializza gli a 0 prima che la funzione sia chiamata
24    static int array1[3];
25
26    puts("\nValues on entering staticArrayInit:");
27
28    // invia in uscita i contenuti di array1
29    for (size_t i = 0; i <= 2; ++i) {
```

```
30          printf("array1[%u] = %d  ", i, array1[i]);
31       }
32
33       puts("\nValues on exiting staticArrayInit:");
34
35       // modifica e invia in uscita i contenuti di array1
36       for (size_t i = 0; i <= 2; ++i) {
37          printf("array1[%u] = %d  ", i, array1[i] += 5);
38       }
39    }
40
41    // funzione usata per illustrare un array locale automatico
42    void automaticArrayInit(void)
43    {
44       // inizializza gli elementi ogni volta
45       int array2[3] = {1, 2, 3};
46
47       puts("\n\nValues on entering automaticArrayInit:");
48
49       // invia in uscita i contenuti di array2
50       for (size_t i = 0; i <= 2; ++i) {
51          printf("array2[%u] = %d  ", i, array2[i]);
52       }
53
54       puts("\nValues on exiting automaticArrayInit:");
55
56       // modifica e invia in uscita i contenuti di array2
57       for (size_t i = 0; i <= 2; ++i) {
58          printf("array2[%u] = %d  ", i, array2[i] += 5);
59       }
60    }
```

```
First call to each function:
Values on entering staticArrayInit:
array1[0] = 0  array1[1] = 0  array1[2] = 0
Values on exiting staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5
Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
Second call to each function:
Values on entering staticArrayInit:
array1[0] = 5  array1[1] = 5  array1[2] = 5 — valori preservati dall'ultima chiamata
Values on exiting staticArrayInit:
array1[0] = 10  array1[1] = 10  array1[2] = 10
Values on entering automaticArrayInit:
array2[0] = 1  array2[1] = 2  array2[2] = 3  —  valori reinizializzati dopo l'ultima
chiamata
Values on exiting automaticArrayInit:
array2[0] = 6  array2[1] = 7  array2[2] = 8
```

**Passare gli array alle funzioni**

Per passare come argomento un array a una funzione dovete specificare il nome dell'array senza alcuna parentesi.

Ad esempio, se l'array `hourlyTemperatures` è stato definito come

```
int hourlyTemperatures[HOURS_IN_A_DAY];
```

la chiamata di funzione

```
modifyArray(hourlyTemperatures, HOURS_IN_A_DAY)
```

passa l'array `hourlyTemperatures` e la sua dimensione alla funzione `modifyArray`.

Il C passa automaticamente gli array alle funzioni per riferimento.

Le funzioni chiamate possono modificare i valori degli elementi negli array originali della funzione chiamante.

Il nome dell'array ha come valore l'indirizzo del primo elemento dell'array.

Poiché viene passato l'indirizzo di partenza dell'array, la funzione chiamata sa precisamente dove l'array è memorizzato.

Di conseguenza, quando la funzione chiamata modifica nel suo corpo gli elementi dell'array, essa modifica gli effettivi elementi dell'array nelle loro originarie locazioni di memoria.

Il programma seguente fa vedere che "il valore del nome di un array" è realmente l'indirizzo del primo elemento dell'array attraverso la stampa di `array`, `&array[0]` e `&array` e usando lo specificatore di conversione `%p` per stampare indirizzi.

Lo specificatore di conversione `%p` normalmente stampa gli indirizzi come numeri esadecimali, ma ciò dipende dal compilatore.

I numeri esadecimali (in base 16) consistono nelle cifre da 0 a 9 e nelle lettere da A a F (queste lettere sono gli equivalenti esadecimali dei numeri decimali 10–15).

L'output mostra che `array`, `&array` e `&array[0]` hanno lo stesso valore, e cioè `0031F930`.

L'output di questo programma è dipendente dal sistema, ma gli indirizzi sono sempre identici per una specifica esecuzione di questo programma su un particolare computer.

Prestazioni

Il passaggio degli array per riferimento ha senso per ragioni di prestazioni. Se gli array fossero passati per valore, verrebbe passata una copia di ogni elemento.

Per array grandi, passati frequentemente, ciò consumerebbe significative quantità di tempo e memoria per copiare gli array.

```
1  // Il nome di un array
2  //  coincide con l'indirizzo del suo primo elemento.
3  #include <stdio.h>
4
5  // la funzione main inizia l'esecuzione del programma
6  int main(void)
7  {
8     char array[5]; // definisci un array di dimensione 5
9
10    printf("    array = %p\n&array[0] = %p\n   &array = %p\n",
11           array, &array[0], &array);
12 }
```

```
        array = 0031F930
&array[0] = 0031F930
        &array = 0031F930
```

# 2. Ordinamento e ricerca

Consideriamo un programma che ordini i valori negli elementi di un array a di 10 elementi in ordine crescente.

La tecnica che usiamo è chiamata *bubble sort* (letteralmente "ordinamento a bolle") o *sinking sort* (letteralmente "ordinamento per affondamento"), perché i valori più piccoli salgono verso la cima dell'array a poco a poco "come bolle", come le bolle d'aria che si formano nell'acqua, mentre i valori più grandi scendono verso il fondo dell'array.

La tecnica consiste nell'effettuare diverse passate lungo l'array.

A ogni passata vengono confrontate le successive coppie di elementi (l'elemento 0 e l'elemento 1, poi l'elemento 1 e l'elemento 2, ecc.).

Se una coppia è in ordine crescente (o se i valori sono identici), si lasciano i valori come sono.

Se una coppia è in ordine decrescente, i valori vengono scambiati nell'array.

```
1  // Programma bubble sort
2  // Ordina i valori di un array in ordine crescente.
3  #include <stdio.h>
4  #define SIZE 10
5
6  // la funzione main inizia l'esecuzione del programma
7  int main(void)
8  {
9     // inizializza a
```

```
10      int a[SIZE] = {2, 6, 4, 8, 10, 12, 89, 68, 45, 37};
11
12      puts("Data items in original order");
13
14      // stampa l'array originario
15      for (size_t i = 0; i < SIZE; ++i) {
16         printf("%4d", a[i]);
17      }
18
19      // bubble sort
20      // ciclo per il numero di passate
21      for (unsigned int pass = 1; pass < SIZE; ++pass) {
22
23         // ciclo per il numero di confronti a ogni passata
24         for (size_t i = 0; i < SIZE - 1; ++i) {
25
26            // confronta due elementi adiacenti e scambiali se
27            // il primo elemento e' maggiore del secondo
28            if (a[i] > a[i + 1]) {
29               int hold = a[i];
30               a[i] = a[i + 1];
31               a[i + 1] = hold;
32            }
33         }
```

```
34      }
35
36      puts("\nData items in ascending order");
37
38      // stampa l'array ordinato
39      for (size_t i = 0; i < SIZE; ++i) {
40         printf("%4d", a[i]);
41      }
42
43      puts("");
44
```

```
Data items in original order
 2  6  4  8 10 12 89 68 45 37
Data items in ascending order
 2  4  6  8 10 12 37 45 68 89
```

Dapprima il programma confronta a[0] e a[1], poi a[1] e a[2], quindi a[2] e a[3], e così via, finché completa la passata confrontando a[8] e a[9].

Benché vi siano 10 elementi, sono eseguiti solo nove confronti.

Per via del modo in cui sono fatti i successivi confronti, un valore grande si può muovere in giù lungo l'array di molte posizioni in una singola passata, ma un valore piccolo si può muovere in su solo di una posizione.

Alla prima passata è garantito che il valore più grande scenda giù fino all'elemento che sta al fondo dell'array, a[9].

Alla seconda passata è garantito che il secondo valore più grande scenda giù fino ad a[8].

Alla nona passata il nono valore più grande scende fino ad a[1].

Questo lascia il valore più piccolo in a[0], così, pur essendovi dieci elementi, sono necessari solo nove passate per ordinare l'array.

L'ordinamento è eseguito dai cicli annidati `for`

Se è necessario uno scambio, questo è eseguito con le tre assegnazioni nel corpo dell'`if`.

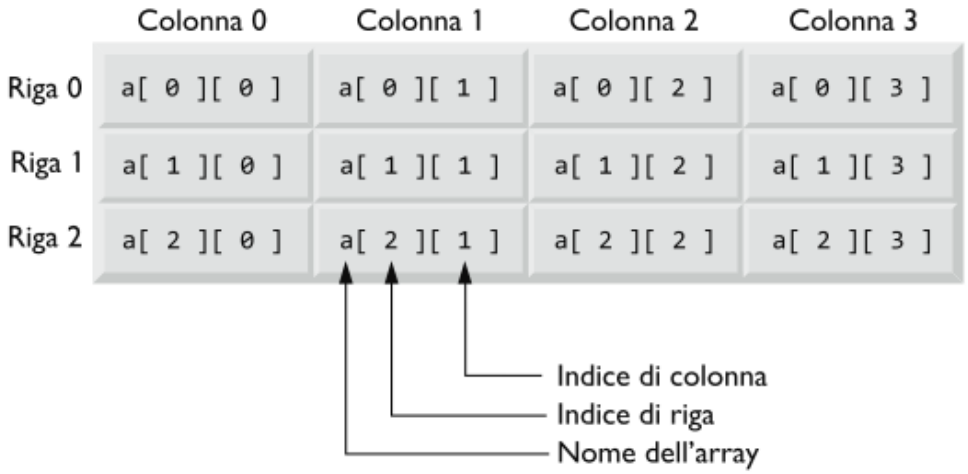
La variabile extra hold memorizza temporaneamente uno dei due valori da scambiare.

# 3. Array multidimensionali

Si possono usare array con più indici.

Vengono chiamati array multidimensionali.

Per esempio, array con due indici (array bidimensionali) rappresentano tabelle di valori disposte in righe e colonne.




Ad esempio, un array bidimensionale

```
int b[2][2]
```

si può definire e inizializzare con

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

Esempio di programma:

```
#include <stdio.h>


void printArray(int a[][3]); // prototipo
```

```
int main(void)
{
   int array1[2][3] = {{1, 2, 3}, {4, 5, 6}};
   puts("Values in array1 by row are:");
   printArray(array1);

   int array2[2][3] = {1, 2, 3, 4, 5};
   puts("Values in array2 by row are:");
   printArray(array2);

   int array3[2][3] = {{1, 2}, {4}};
   puts("Values in array3 by row are:");
   printArray(array3);
}

// funzione per stampare un array
// con due righe e tre colonne
void printArray(int a[][3])
{
   for (size_t i = 0; i <= 1; ++i) {
      for (size_t j = 0; j <= 2; ++j) {
         printf("%d ", a[i][j]);
      }
      printf("\n"); // inizia una nuova riga
   }
}
```

```
Values in array1 by row are:

1 2 3

4 5 6

Values in array2 by row are:

1 2 3

4 5 0

Values in array3 by row are:

1 2 0

4 0 0
```

# Riferimenti bibliografici


- Paul Deitel, Harvey Deitel, "Il linguaggio C – Fondamenti e tecniche di programmazione", Libro edito da Pearson Italia. Include anche utili esercizi di autovalutazione.