UNIVERSITÀ
DEGLI STUDI DI BARI
ALDO MORO

Corso di Laurea in Informatica e Tecnologie per la Produzione del Software *(Track B)* - A.A. 2017/2018

Laboratorio di Informatica

Puntatori

(Parte 2)

docente: Cataldo Musto

cataldo.musto@uniba.it

Slides ispirate ai contenuti proposti
dal prof. Corrado Mencar. Grazie.

Aritmetica dei Puntatori

- I puntatori sono inoltre particolarmente utili per manipolare e accedere agli elementi di un array, attraverso un meccanismo che prende il nome di **aritmetica dei puntatori**
- **Concetti che già conosciamo**

  Il nome dell'array è un puntatore al primo elemento dell'array

Aritmetica dei Puntatori

- I puntatori sono inoltre particolarmente utili per manipolare e accedere agli elementi di un array, attraverso un meccanismo che prende il nome di **aritmetica dei puntatori**
- **Concetti che già conosciamo**

  Il nome dell'array è un puntatore al primo elemento dell'array

```
int array[10] // dichiaro un array di interi
int* p_array // dichiaro un puntatore a un intero

p_array = &array[0];
p_array = array; // scritture equivalenti
```

Aritmetica dei Puntatori

```c
#include <stdio.h>

int main() {
  int array[10]; // dichiaro un array di interi
  int* p_array; // dichiaro un puntatore a un intero

  p_array = &array[0];

  printf("Nome dell'array: \t\t%X \n", array);
  printf("Indirizzo del primo elemento: \t%X \n", &array[0]);
  printf("Puntatore all'array: \t\t%X \n", p_array);
}

```

Cosa stampa?

Aritmetica dei Puntatori

```c
#include <stdio.h>

int main() {
  int array[10]; // dichiaro un array di interi
  int* p_array; // dichiaro un puntatore a un intero

  p_array = &array[0];

  printf("Nome dell'array: \t\t%X \n", array);
  printf("Indirizzo del primo elemento: \t%X \n", &array[0]);
  printf("Puntatore all'array: \t\t%X \n", p_array);
}

```

Cosa stampa?

```
gcc version 4.6.3
Nome dell'array:                A9FBE990
Indirizzo del primo elemento:   A9FBE990
Puntatore all'array:            A9FBE990
```

Aritmetica dei Puntatori

```c
#include <stdio.h>

int main() {
  int array[10]; // dichiaro un array di interi
  int* p_array; // dichiaro un puntatore a un intero

  p_array = &array[0];

  printf("Nome dell'array: \t\t%X \n", array);
  printf("Indirizzo del primo elemento: \t%X \n", &array[0]);
  printf("Puntatore all'array: \t\t%X \n", p_array);
}
```

Cosa stampa?

```
gcc version 4.6.3
Nome dell'array:                A9FBE990
Indirizzo del primo elemento:   A9FBE990
Puntatore all'array:            A9FBE990
```

Le tre scritture sono equivalenti. Il nome dell'array è un indirizzo di memoria, che corrisponde all'indirizzo di memoria del primo elemento dell'array, che a **sua volta può tranquillamente essere assegnato a un puntatore**

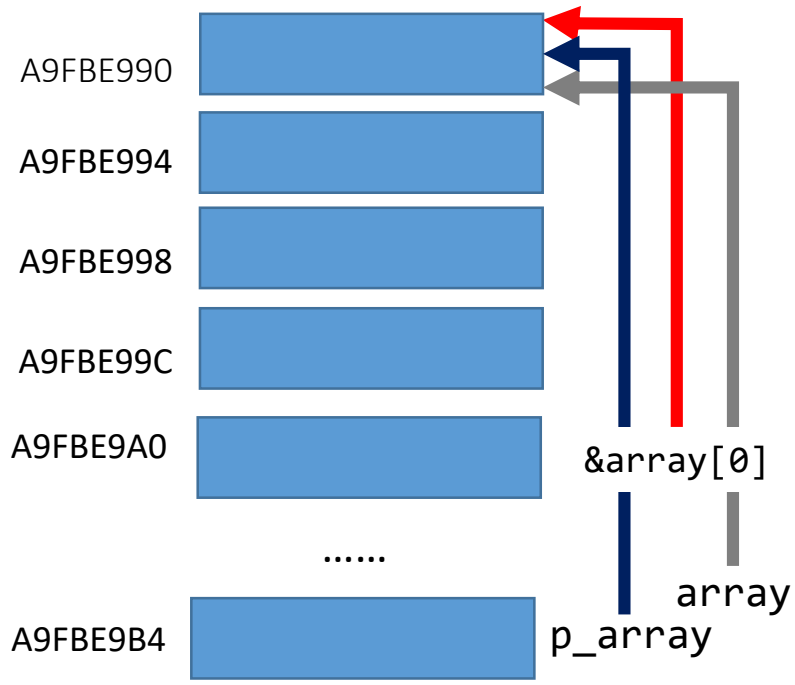
Aritmetica dei Puntatori

```c
#include <stdio.h>

int main() {
  int array[10]; // dichiaro un array di interi
  int* p_array; // dichiaro un puntatore a un intero

  p_array = &array[0];

  printf("Nome dell'array: \t\t%X \n", array);
  printf("Indirizzo del primo elemento: \t%X \n", &array[0]);
  printf("Puntatore all'array: \t\t%X \n", p_array);
}
```



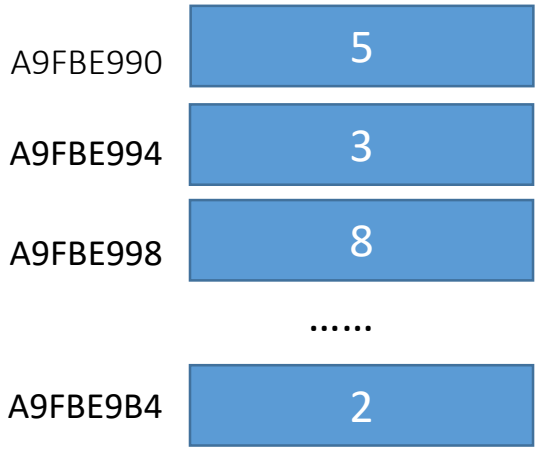
Le tre scritture sono equivalenti. Il nome dell'array è un indirizzo di memoria, che corrisponde all'indirizzo di memoria del primo elemento dell'array, che a **sua volta può tranquillamente essere assegnato a un puntatore**

Aritmetica dei Puntatori

- Array e puntatori sono in stretta relazione
- **Attraverso l'aritmetica dei puntatori possiamo accedere e manipolare gli elementi di un array in modo «alternativo»**

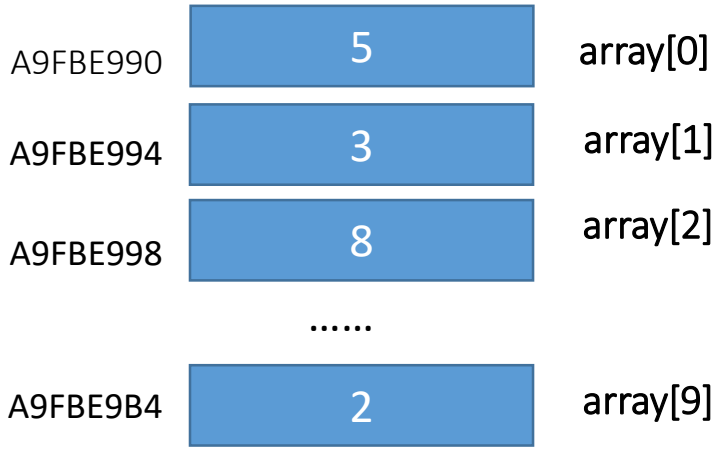
Aritmetica dei Puntatori

- Array e puntatori sono in stretta relazione
- **Attraverso l'aritmetica dei puntatori possiamo accedere e manipolare gli elementi di un array in modo «alternativo»**
- Come accediamo normalmente agli elementi di un array?



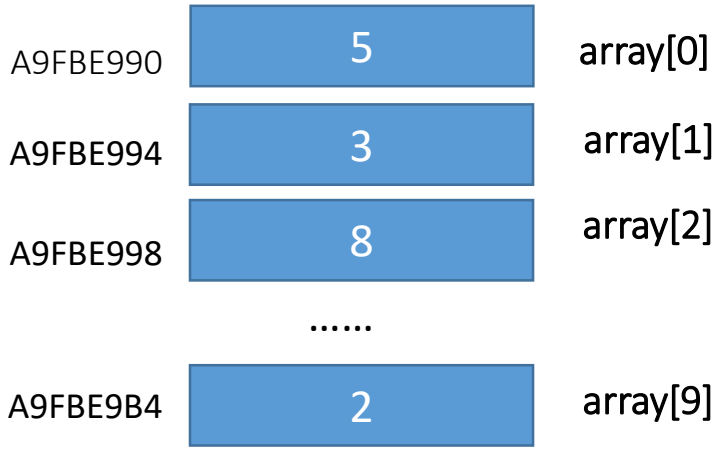
Aritmetica dei Puntatori

- Array e puntatori sono in stretta relazione
- **Attraverso l'aritmetica dei puntatori possiamo accedere e manipolare gli elementi di un array in modo «alternativo»**
- Come accediamo normalmente agli elementi di un array?
  - Con la notazione indice (es. **`array[i]`** → i+1 esimo elemento del vettore)



Aritmetica dei Puntatori

- Array e puntatori sono in stretta relazione
- **Attraverso l'aritmetica dei puntatori possiamo accedere e manipolare gli elementi di un array in modo «alternativo»**
- Come accediamo normalmente agli elementi di un array?
  - Con la notazione indice (es. **`array[i]`** → i+1 esimo elemento del vettore)



Attraverso l'aritmetica dei puntatori **possiamo utilizzare i puntatori in espressioni aritmetiche.**

Aritmetica dei Puntatori - Esempio

- Attraverso l'aritmetica dei puntatori **possiamo utilizzare i puntatori in espressioni aritmetiche.**
  - `int* arrayPtr = &array[0];` `// punta al primo elemento`

Aritmetica dei Puntatori - Esempio

- Attraverso l'aritmetica dei puntatori **possiamo utilizzare i puntatori in espressioni aritmetiche.**
  - `int* arrayPtr = &array[0];` `// punta al primo elemento`
  - `arrayPtr= arrayPtr+4;` `// che cosa succede?`

Aritmetica dei Puntatori - Esempio

- Attraverso l'aritmetica dei puntatori **possiamo utilizzare i puntatori in espressioni aritmetiche.**
  - `int* arrayPtr = &array[0];` `// punta al primo elemento`
  - `arrayPtr= arrayPtr+4;` `// che cosa succede?`

Attraverso quella espressione aritmetica **`arrayPtr`** punterà al quinto elemento dell'array.

Aritmetica dei Puntatori - Esempio

- Attraverso l'aritmetica dei puntatori **possiamo utilizzare i puntatori in espressioni aritmetiche.**
  - `int* arrayPtr = &array[0];` `// punta al primo elemento`
  - `arrayPtr= arrayPtr+1;` `// che cosa succede?`

Attraverso quella espressione aritmetica **`arrayPtr`** punterà all'elemento successivo dell'array.

Aritmetica dei puntatori

- E se volessimo usare l'aritmetica dei puntatori per accedere al valore **dei singoli elementi di un array?**

Aritmetica dei puntatori

- E se volessimo usare l'aritmetica dei puntatori per accedere al valore **dei singoli elementi di un array?**
- **Bisogna usare l'operatore di indirezione,** per accedere al contenuto dei puntatori

Aritmetica dei puntatori

- E se volessimo usare l'aritmetica dei puntatori per accedere al valore **dei singoli elementi di un array?**
- **Bisogna usare l'operatore di indirezione,** per accedere al contenuto dei puntatori

Dato un array a di elementi di tipo t (t a[])

- `a` **è il puntatore al primo elemento dell'array, pertanto:**
  - `a[0]` accede al primo elemento dell'array
  - `*a` accede al primo elemento dell'array

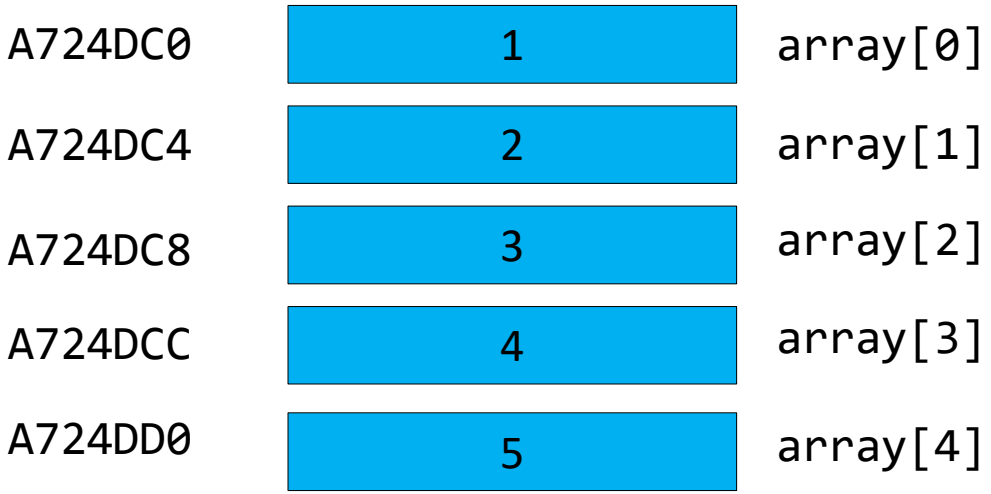
Aritmetica dei puntatori

- E se volessimo usare l'aritmetica dei puntatori per accedere al valore **dei singoli elementi di un array?**
- **Bisogna usare l'operatore di indirezione,** per accedere al contenuto dei puntatori

Dato un array a di elementi di tipo t (t a[])

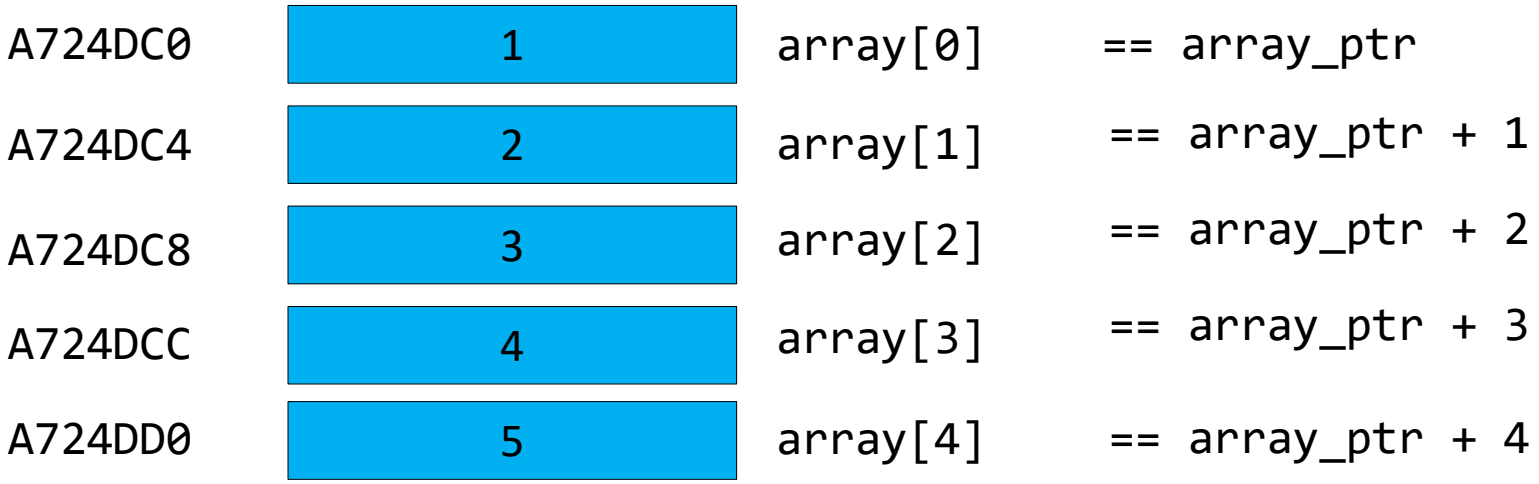
- **`a` è il puntatore al primo elemento dell'array, pertanto:**
  - `a[0]` accede al primo elemento dell'array
  - `*a` accede al primo elemento dell'array
- **`a+i` punta all'i-esimo elemento dell'array, pertanto**
  - `a[i]` accede allo i-esimo elemento dell'array
  - `*(a+i)` accede allo i-esimo elemento dell'array

Aritmetica dei puntatori – Recap Generale



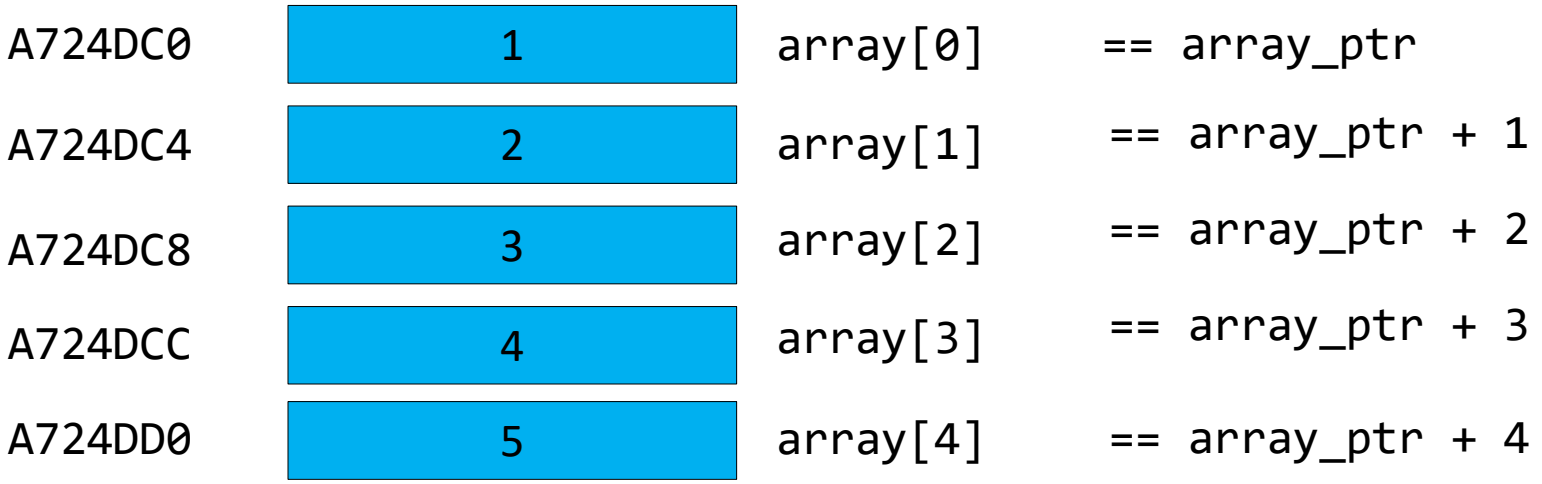

```
int array[5] = {1, 2, 3, 4, 5}
```

Aritmetica dei puntatori – Recap Generale




```
int array[5] = {1, 2, 3, 4, 5}
int* array_ptr = &array[0]
```

Aritmetica dei puntatori – Recap Generale



La prima si chiama **«notazione indice»**

La seconda si chiama **«notazione puntatore + offset»**

```
int array[5] = {1, 2, 3, 4, 5}
int* array_ptr = &array[0]

array[3] == *(array_ptr+3) // le notazioni sono equivalenti!
```

Utilizzo dei puntatori - Passaggio degli Array

- **Concetti che già conosciamo**

  Quando sono utilizzati come parametri nelle funzioni, gli array vengono passati automaticamente per riferimento.

  Ora riusciamo a capire meglio il perchè

Utilizzo dei puntatori - Passaggio degli Array

- **Concetti che già conosciamo**

  Quando sono utilizzati come parametri nelle funzioni, gli array vengono passati automaticamente per riferimento.

  Ora riusciamo a capire meglio il perché

  Perché il nome dell'array è un puntatore al primo elemento, quindi passandolo ad una funzione stiamo implicitamente passando l'indirizzo (dunque realizziamo senza saperlo un passaggio dei parametri per riferimento)

Utilizzo dei puntatori - Passaggio degli Array

- **Concetti che già conosciamo**

  Quando sono utilizzati come parametri nelle funzioni, gli array vengono passati automaticamente per riferimento.

  Ora riusciamo a capire meglio il perché

  Perché il nome dell'array è un puntatore al primo elemento, quindi passandolo ad una funzione stiamo implicitamente passando l'indirizzo (dunque realizziamo senza saperlo un passaggio dei parametri per riferimento)

Esempio

- Dato l'array **`int vector[3]`** e data una funzione **`double f(int v[]){...}`**
- Quando si invoca **`f(vector)`**, poiché vector è un puntatore al primo elemento dell'array, stiamo passando un puntatore e non l'intero array

Utilizzo dei puntatori – Passaggio degli Array

```
int sum(int v[], int n)
```

- indica espressamente che v è un array
- **Più indicato usare la sintassi degli array**

```
int sum(int* v, int n)
```

- Non è chiaro che v è un array
- Più indicato se si usa l'aritmetica dei puntatori

L'array può essere passato alla funzione in entrambi i modi. **La scelta è personale.** La prima è probabilmente più leggibile e più comprensibile. La seconda è più utile se si utilizza l'aritmetica dei puntatori dentro la funzione.

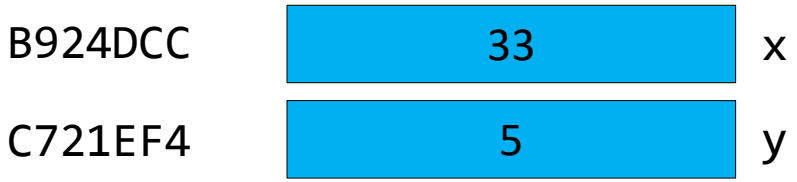
Array di Puntatori

- I puntatori sono delle variabili a tutti gli effetti
  - Solo che possono contenere degli indirizzi

Array di Puntatori

- I puntatori sono delle variabili a tutti gli effetti
  - Solo che possono contenere degli indirizzi
- I puntatori possono anche essere **inseriti in un array** o in una struct
- **Esempio**

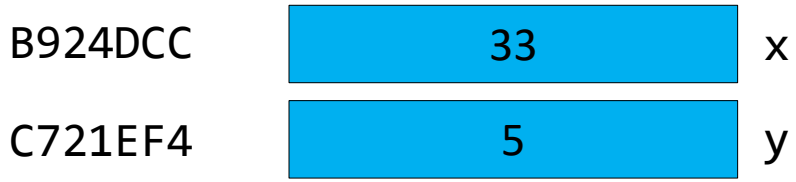
```
int x = 33; // dichiaro i valori int
y = 5;
```



Array di Puntatori

- I puntatori sono delle variabili a tutti gli effetti
  - Solo che possono contenere degli indirizzi
- I puntatori possono anche essere **inseriti in un array** o in una struct
- **Esempio**

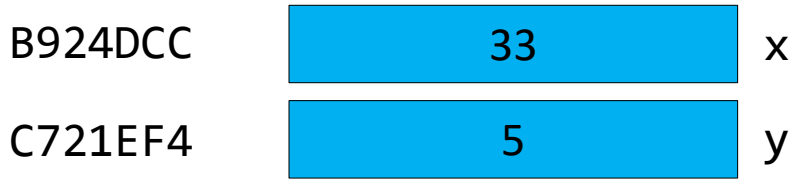
```
int x = 33; // dichiaro i valori int
y = 5;
// array di puntatori ad interi
int* array[2] = // come lo inizializzo?
```



Array di Puntatori

- I puntatori sono delle variabili a tutti gli effetti
  - Solo che possono contenere degli indirizzi
- I puntatori possono anche essere **inseriti in un array** o in una struct
- **Esempio**

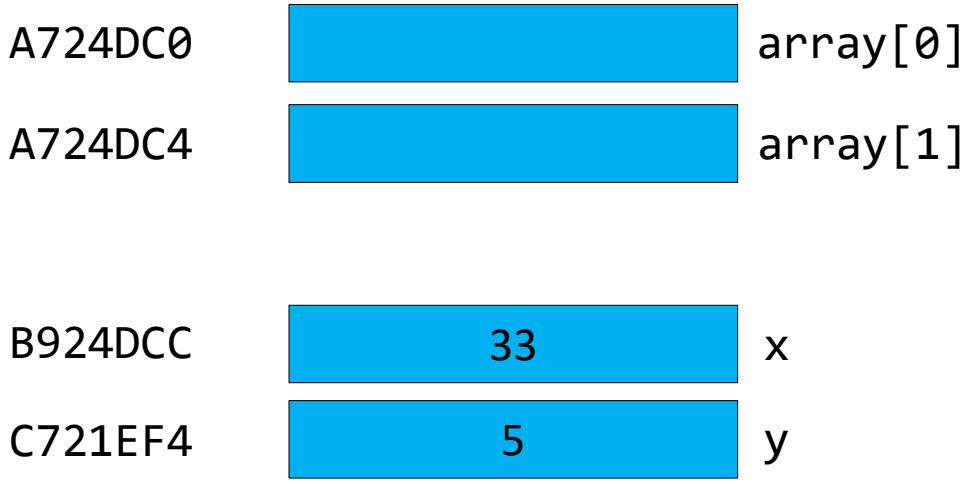
```
int x = 33; // dichiaro i valori int
y = 5;
// array di puntatori ad interi
int* array[2] = {&x, &y};
```



Array di Puntatori

- I puntatori sono delle variabili a tutti gli effetti
  - Solo che possono contenere degli indirizzi
- I puntatori possono anche essere **inseriti in un array** o in una struct
- **Esempio**

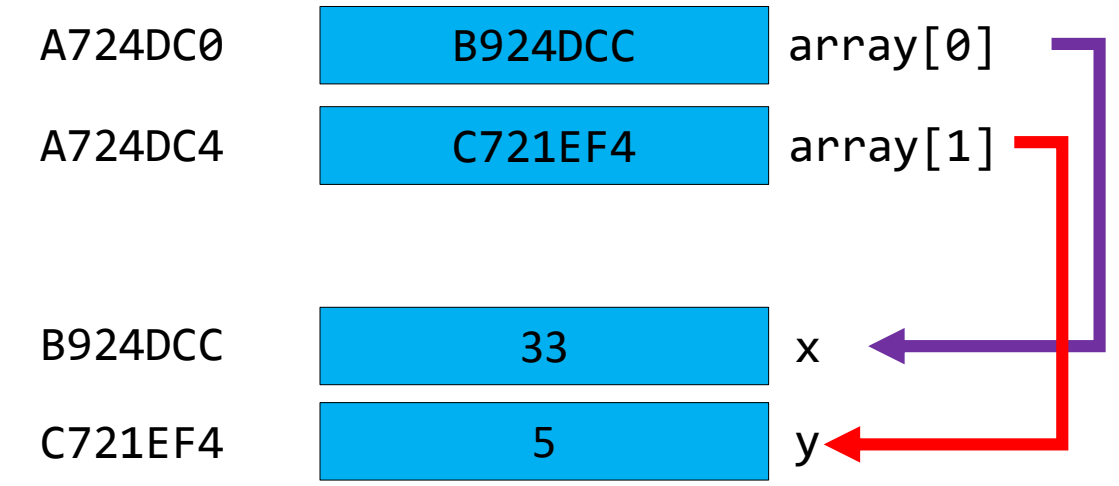
```
int x = 33; // dichiaro i valori int
y = 5;
// array di puntatori ad interi
int* array[2] = {&x, &y};
```



Array di Puntatori

- I puntatori sono delle variabili a tutti gli effetti
  - Solo che possono contenere degli indirizzi
- I puntatori possono anche essere **inseriti in un array** o in una struct
- **Esempio**

```
int x = 33; // dichiaro i valori int
y = 5;
// array di puntatori ad interi
int* array[2] = {&x, &y};
```



Dentro l'array non abbiamo valori, **ma indirizzi!**

Problema

- Riscriviamo la funzione **`scambia()`** facendo in modo che prenda in input **un array di puntatori**, **invece che due puntatori**

```
void scambia(int* a, int* b) {
   int t; // variabile locale di appoggio
   t = *a; // scambio dei valori
   *a = *b;
   *b = t;
}

int main() {
   int x = 33, y = 5;
   scambia(&x, &y);
   printf("x = %d, y = %d\n", x, y);
}
```

Problema

- Riscriviamo la funzione **`scambia()`** facendo in modo che prenda in input **un array di puntatori**, **invece che due puntatori**

```
void scambia(int* a, int* b) {
   int t; // variabile locale di appoggio
   t = *a; // scambio dei valori
   *a = *b;
   *b = t;
}

int main() {
   int x = 33, y = 5;
   scambia(&x, &y);
   printf("x = %d, y = %d\n", x, y);
}
```

```
void scambia(int* a[]) {




                    }


int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

Problema

- Riscriviamo la funzione **`scambia()`** facendo in modo che prenda in input **un array di puntatori**, **invece che due puntatori**

```
void scambia(int* a, int* b) {
   int t; // variabile locale di appoggio
   t = *a; // scambio dei valori
   *a = *b;
   *b = t;
}

int main() {
   int x = 33, y = 5;
   scambia(&x, &y);
   printf("x = %d, y = %d\n", x, y);
}
```

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

Problema

- Riscriviamo la funzione **`scambia()`** facendo in modo che prenda in input **un array di puntatori**, **invece che due puntatori**

```
void scambia(int* a, int* b) {
   int t; // variabile locale di appoggio
   t = *a; // scambio dei valori
   *a = *b;
   *b = t;
}

int main() {
   int x = 33, y = 5;
   scambia(&x, &y);
   printf("x = %d, y = %d\n", x, y);
}
```

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

Dichiaro il vettore di puntatori

Problema

- Riscriviamo la funzione **`scambia()`** facendo in modo che prenda in input **un array di puntatori**, **invece che due puntatori**

```
void scambia(int* a, int* b) {
   int t; // variabile locale di appoggio
   t = *a; // scambio dei valori
   *a = *b;
   *b = t;
}

int main() {
   int x = 33, y = 5;
   scambia(&x, &y);
   printf("x = %d, y = %d\n", x, y);
}
```

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

Cambiamo l'input della funzione

Problema

- Riscriviamo la funzione **`scambia()`** facendo in modo che prenda in input **un array di puntatori**, **invece che due puntatori**

```
void scambia(int* a, int* b) {
   int t; // variabile locale di appoggio
   t = *a; // scambio dei valori
   *a = *b;
   *b = t;
}

int main() {
   int x = 33, y = 5;
   scambia(&x, &y);
   printf("x = %d, y = %d\n", x, y);
}
```

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

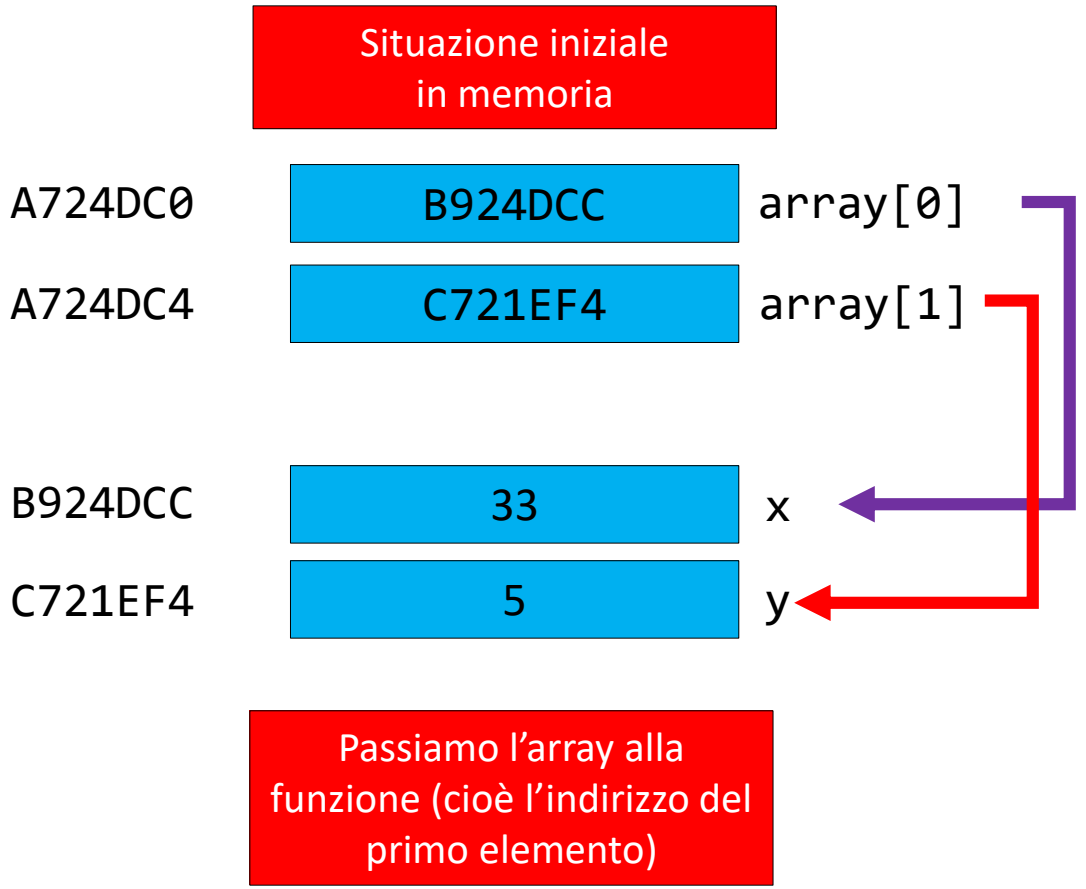
Modifichiamo la procedura di scambio

Array di Puntatori - Esempio

- Studiamo meglio la procedura di scambio

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

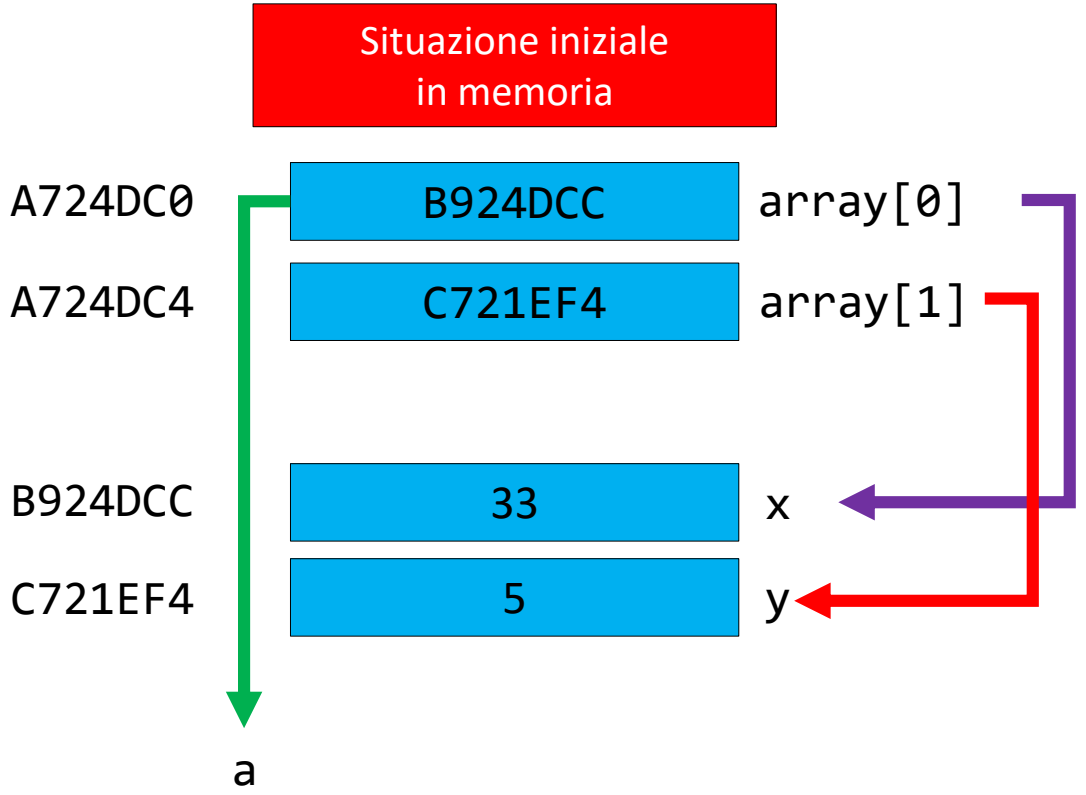


Array di Puntatori - Esempio

- Studiamo meglio la procedura di scambio

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

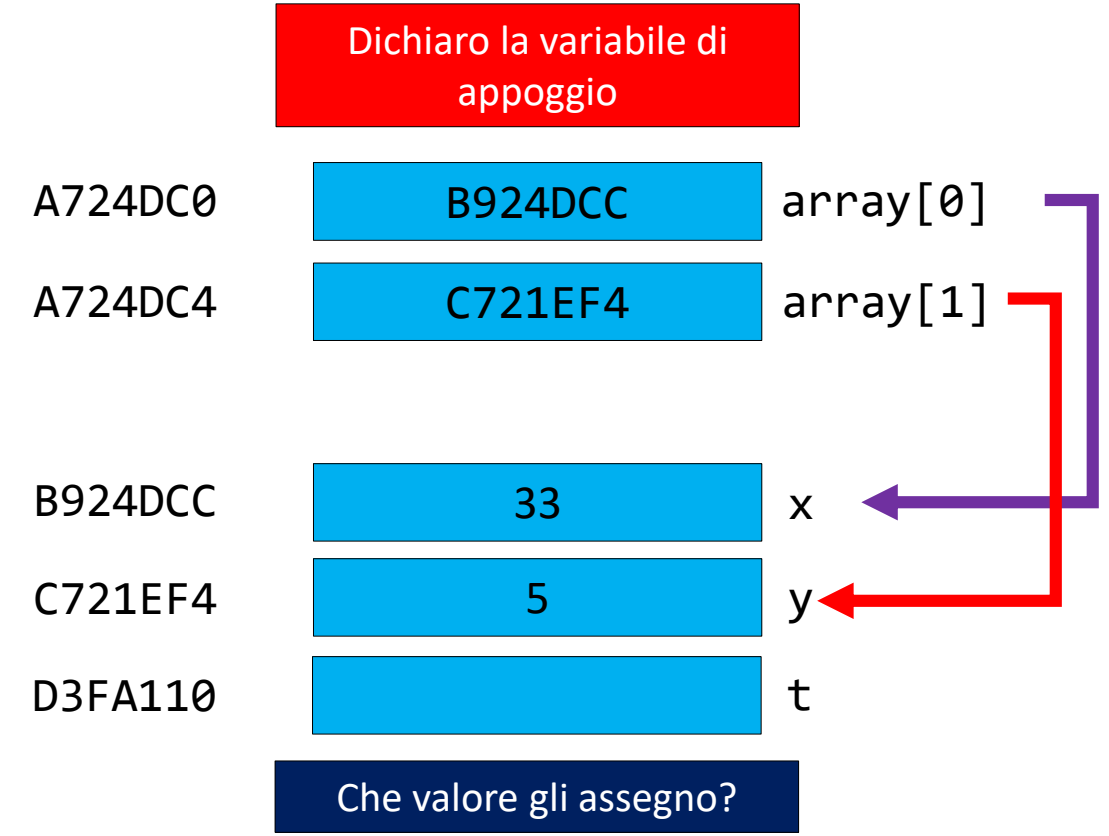


Array di Puntatori - Esempio

- Studiamo meglio la procedura di scambio

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

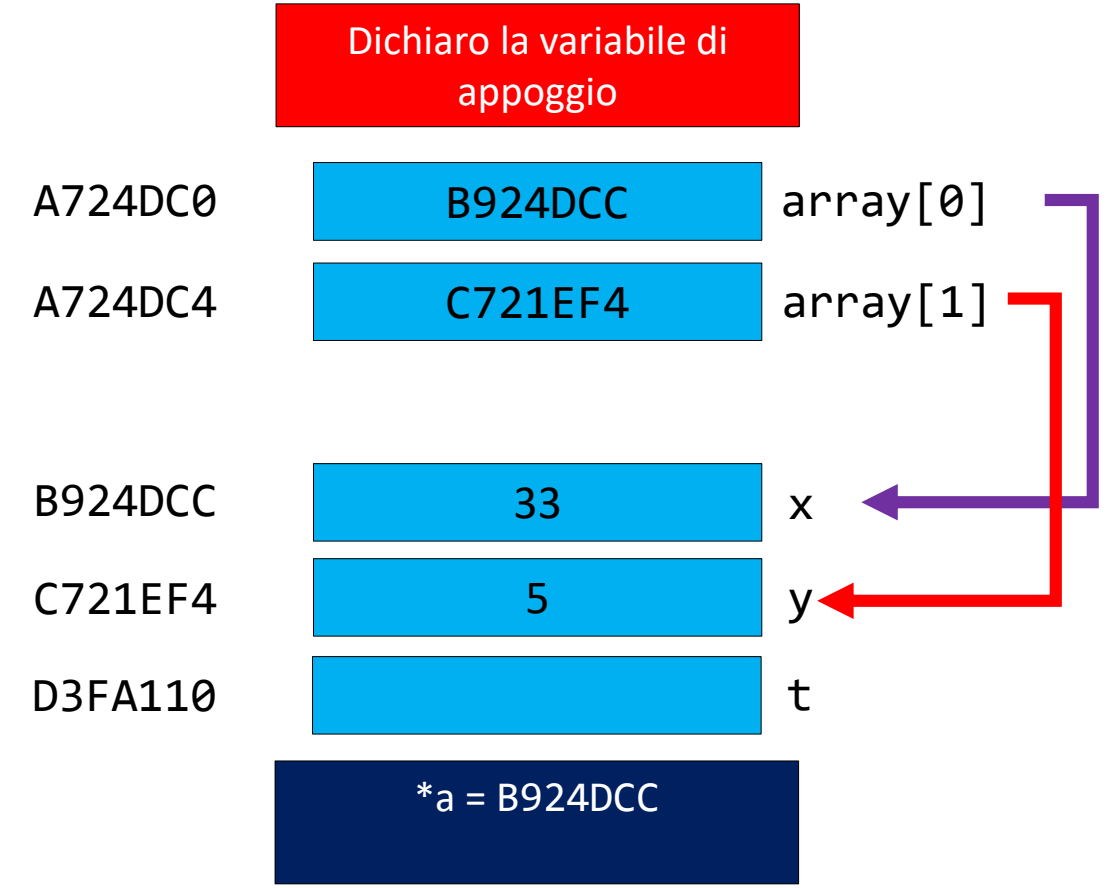


Array di Puntatori - Esempio

- Studiamo meglio la procedura di scambio

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

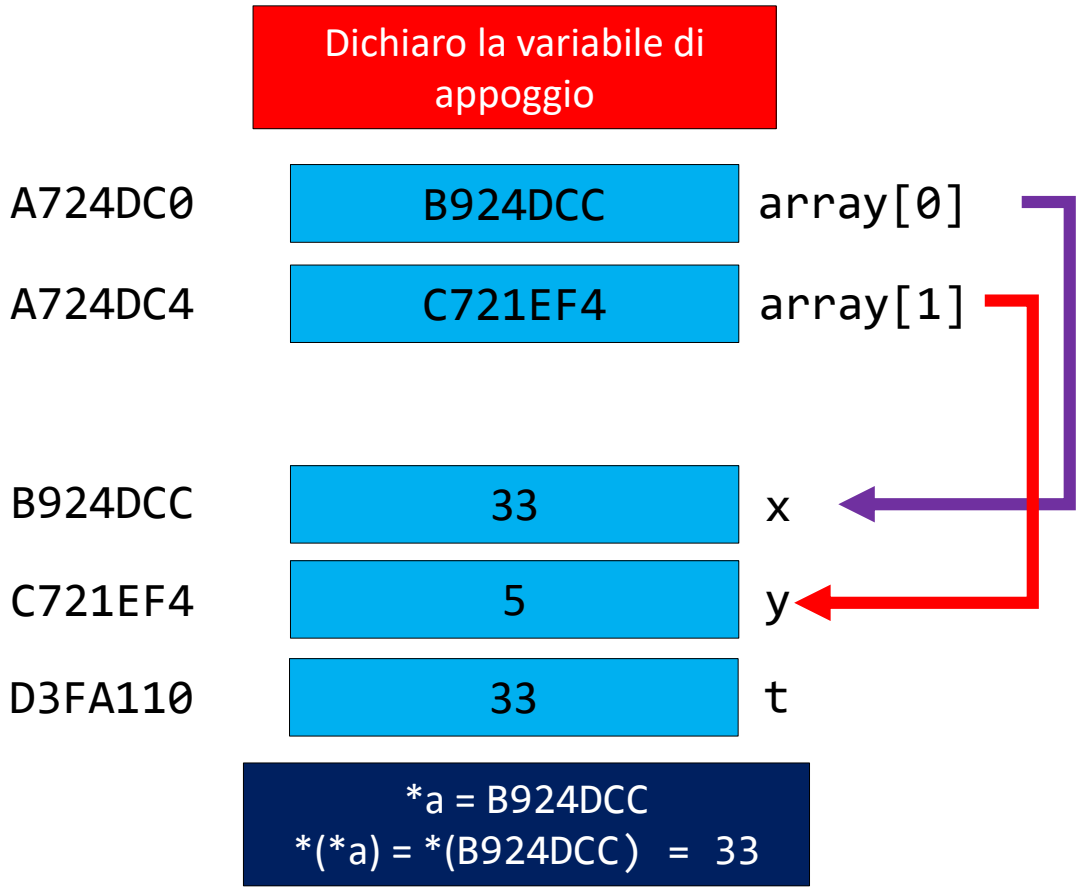


Array di Puntatori - Esempio

- Studiamo meglio la procedura di scambio

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

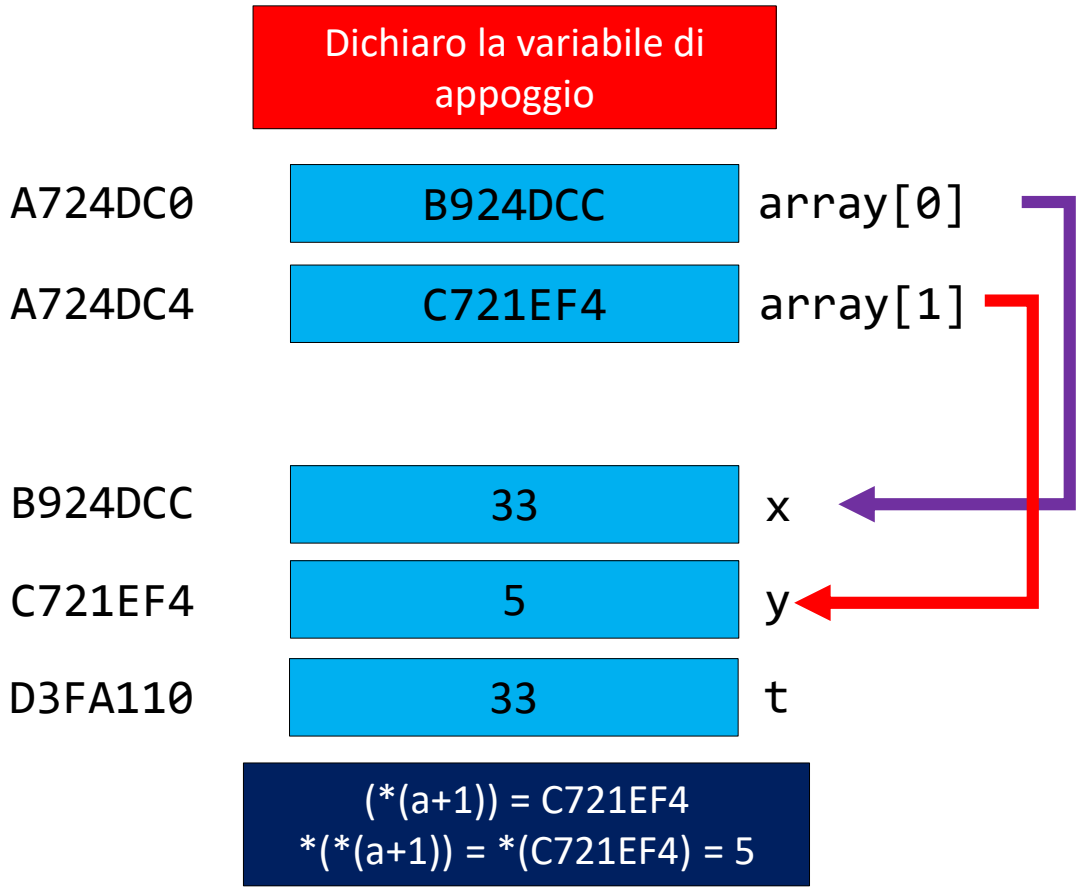


Array di Puntatori - Esempio

- Studiamo meglio la procedura di scambio

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

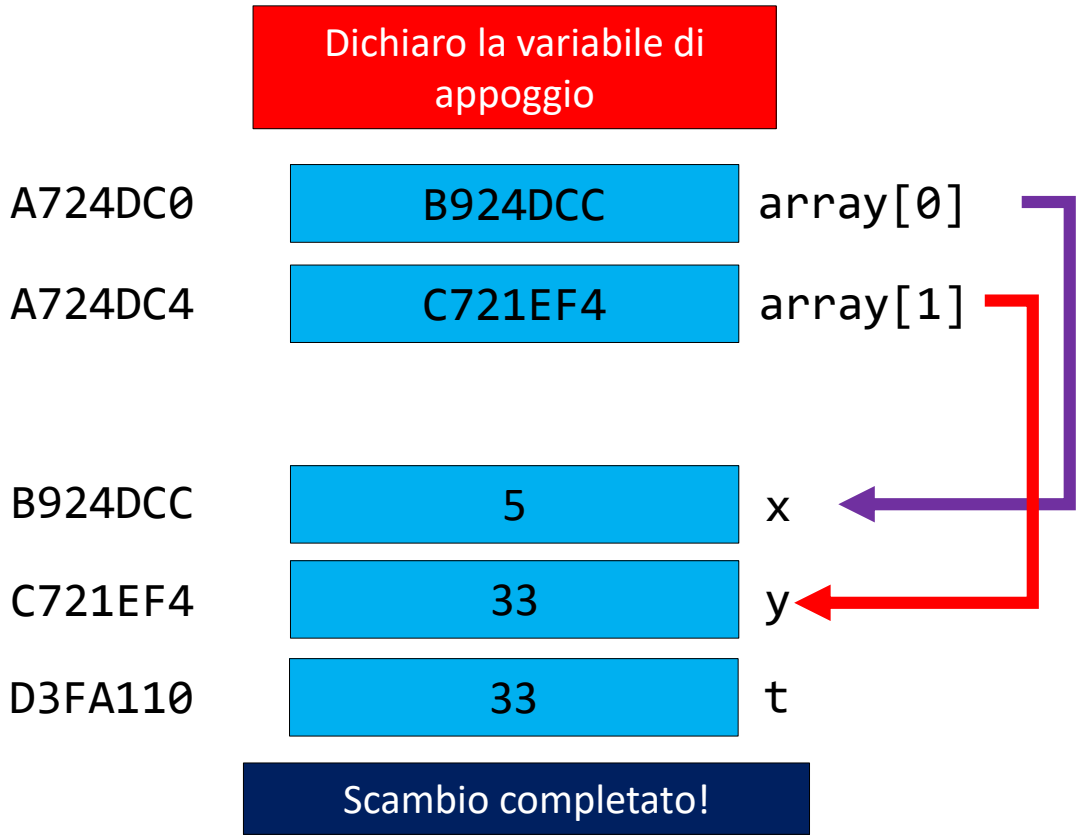


Array di Puntatori - Esempio

- Studiamo meglio la procedura di scambio

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```



Array di Puntatori - Esempio

- Studiamo meglio la procedura di scambio

```
void scambia(int* a[]) {
   int t; // variabile locale di appoggio
   t = *(*a);    // scambio dei valori
   *(*a) = *(*(a+1));
   *(*(a+1)) = t;}

int main() {
   int x = 33, y = 5;
   int* array[2] = {&x, &y}
   scambia(array);
   printf("x = %d, y = %d\n", x, y);
}
```

Quando si utilizzano i vettori di puntatori è necessario utilizzare **la notazione con il doppio operatore di indirezione.**

Il primo operatore di indirezione serve a risalire all'indirizzo di memoria del primo elemento.

Il secondo operatore di indirezione sere a risalire al suo valore, che è quello che ci serve.

Array di Puntatori

- I puntatori sono delle variabili a tutti gli effetti
  - Solo che possono contenere degli indirizzi
- I puntatori possono anche essere inseriti in un array o in una struct

Array di Puntatori

- I puntatori sono delle variabili a tutti gli effetti
  - Solo che possono contenere degli indirizzi
- I puntatori possono anche essere inseriti in un array o in una struct
  - **Applicazioni: un array di puntatori** ci serve ad esempio se abbiamo bisogno di un array di stringhe
  - Una struct con dentro un puntatore (o più puntatori) serve se dobbiamo implementare **delle strutture dati** (es. liste, pile)

Array di Puntatori

- I puntatori sono delle variabili a tutti gli effetti
  - Solo che possono contenere degli indirizzi
- I puntatori possono anche essere inseriti in un array o in una struct
  - **Applicazioni: un array di puntatori** ci serve ad esempio se abbiamo bisogno di un array di stringhe ← ci interessa ☺
  - Una struct con dentro un puntatore (o più puntatori) serve se dobbiamo implementare **delle strutture dati** (es. liste, pile) ← non ci interessa

Array di Puntatori

- Perché un array di stringhe è un array di puntatori?

Array di Puntatori

- Perché un array di stringhe è un array di puntatori?
  - Perché una stringa è un **array di caratteri**

Array di Puntatori

- Perché un array di stringhe è un array di puntatori?
  - Perché una stringa è un **array di caratteri**
  - Un array di caratteri è un puntatore al primo caratteri

Array di Puntatori

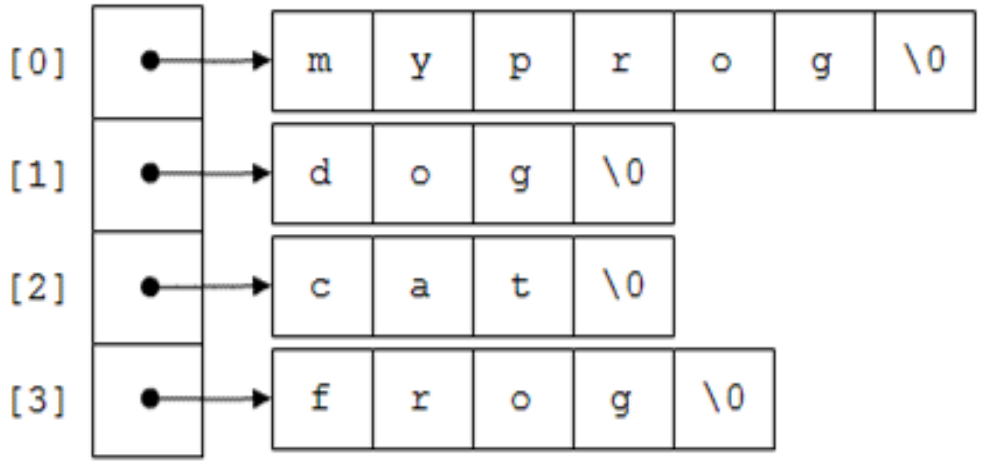
- Perché un array di stringhe è un array di puntatori?
  - Perché una stringa è un **array di caratteri**
  - Un array di caratteri è un puntatore al primo caratteri
  - Una stringa **è un puntatore al primo carattere**

Array di Puntatori

- Perché un array di stringhe è un array di puntatori?
  - Perché una stringa è un **array di caratteri**
  - Un array è un puntatore al primo elemento (in questo caso al primo carattere)
  - Una stringa **è un puntatore al primo carattere**
  - **Un array di stringhe è dunque un array di puntatori al primo carattere della stringa**

Array di Puntatori

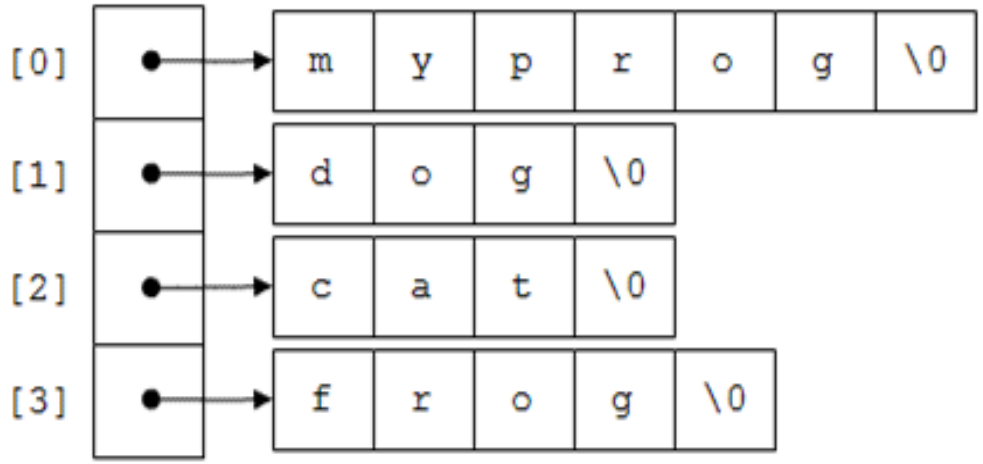
- Perché un array di stringhe è un array di puntatori?
  - Perché una stringa è un **array di caratteri**
  - Un array è un puntatore al primo elemento (in questo caso al primo carattere)
  - Una stringa **è un puntatore al primo carattere**
  - **Un array di stringhe è dunque un array di puntatori al primo carattere della stringa**



Vediamo come codificarlo!

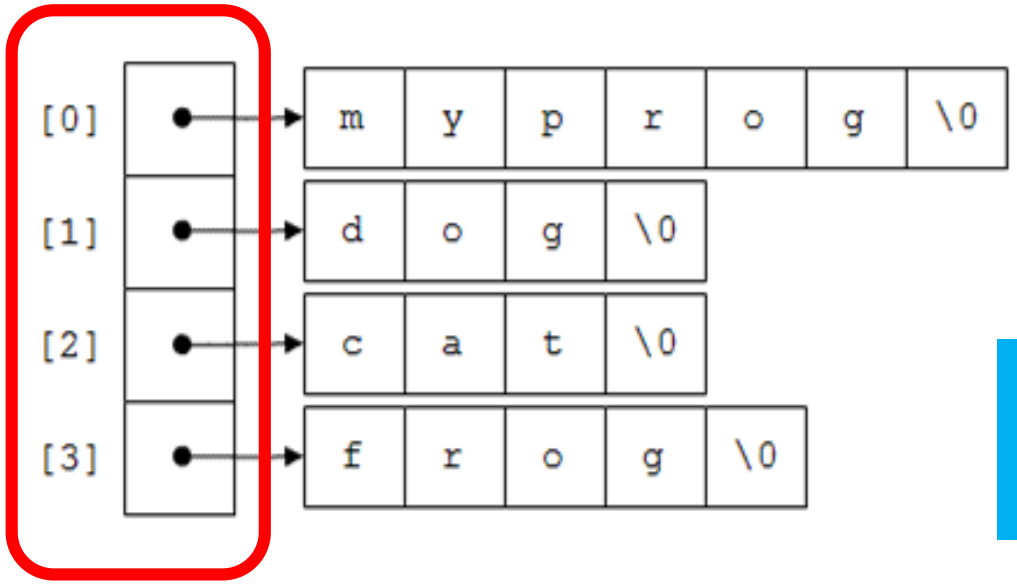
Array di Puntatori

```
char* array[4] = {"myprog", "dog", "cat" , "frog"};
```



Array di Puntatori

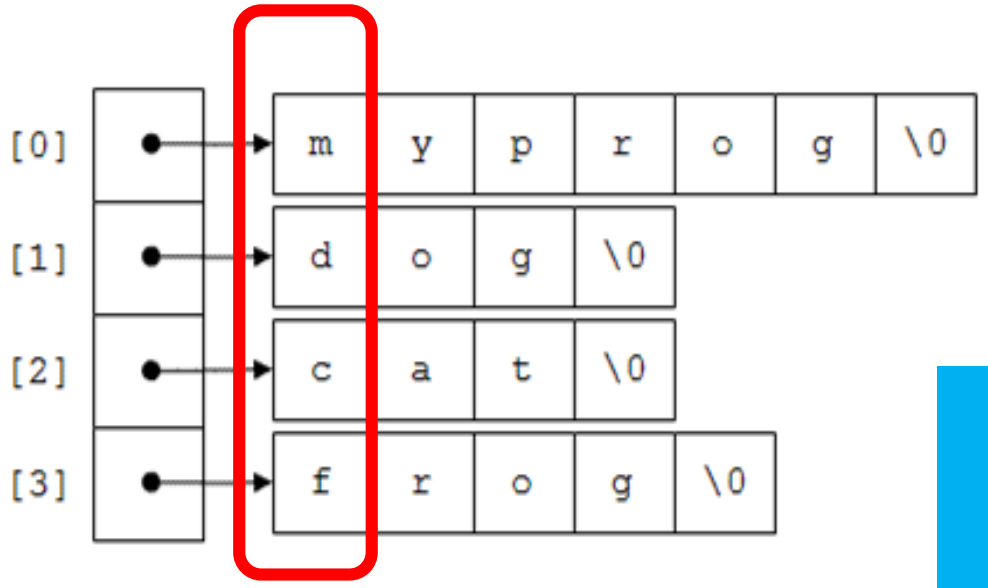
```
char* array[4] = {"myprog", "dog", "cat" , "frog"};
```



Istanziamo un
array di 4 puntatori

Array di Puntatori

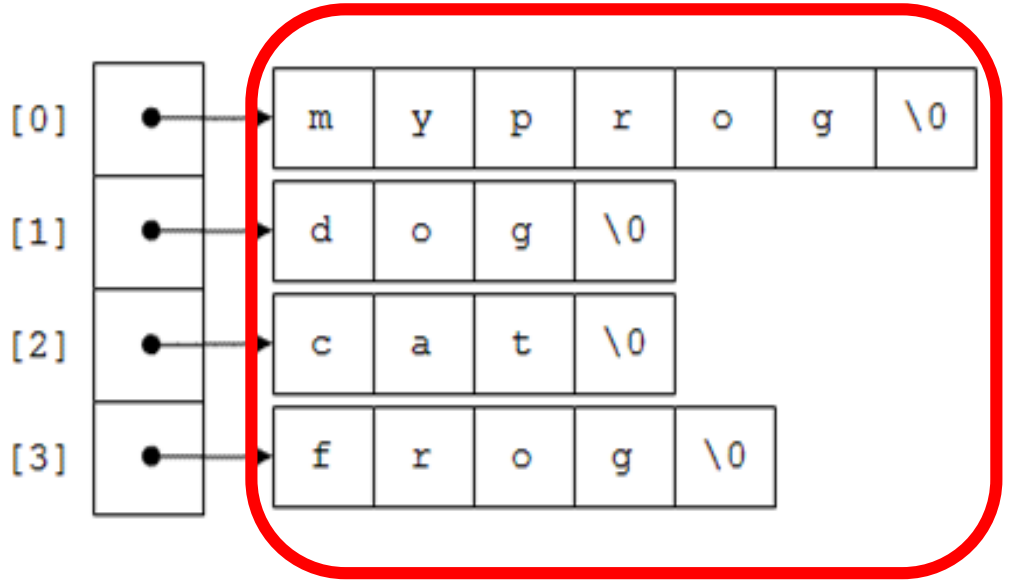
```
char* array[4] = {"myprog", "dog", "cat" , "frog"};
```



Ogni elemento è un puntatore a un char **(quindi una stringa)**

Array di Puntatori

```
char* array[4] = {"myprog", "dog", "cat" , "frog"};
```



Importante: gli elementi possono **anche avere lunghezza diversa**

Esercizio 9.1

- Scrivere una FUNZIONE che**, dati in input due array di stringhe composti da nome e cognome, inverta il nome** dei due individui **e lo stampi in output. La stampa deve avvenire nel main!**
- **Esempio:**
  - **Input: Mario Rossi , Luigi Bianchi**
  - **Output: Luigi Rossi, Mario Bianchi**

Nota

Non è necessario acquisire interattivamente i dati. **I vettori possono anche essere inizializzati nel codice**

Restituire un Puntatore

- Nuovamente
  - **I puntatori sono variabili a tutti gli effetti.**
  - **Possono essere dichiarate e possono essere utilizzate in array e struct**
  - **Possono ovviamente anche essere restituiti dalle funzioni**

Restituire un Puntatore

- Nuovamente
  - **I puntatori sono variabili a tutti gli effetti.**
  - **Possono essere dichiarate e possono essere utilizzate in array e struct**
  - **Possono ovviamente anche essere restituiti dalle funzioni**

- Per far restituire un puntatore ad una funzione bisogna però seguire **degli accorgimenti particolari**

Restituire un Puntatore

- Nuovamente
  - **I puntatori sono variabili a tutti gli effetti.**
  - **Possono essere dichiarate e possono essere utilizzate in array e struct**
  - **Possono ovviamente anche essere restituiti dalle funzioni**

- Per far restituire un puntatore ad una funzione bisogna però seguire **degli accorgimenti particolari**
  - **Motivo:** allocazione della memoria
  - **La memoria in C viene gestita staticamente.** Ciò significa che bisogna sapere a priori quali variabili saranno utilizzate e quando saranno grandi.

Restituire un Puntatore

- Nuovamente
  - **I puntatori sono variabili a tutti gli effetti.**
  - **Possono essere dichiarate e possono essere utilizzate in array e struct**
  - **Possono ovviamente anche essere restituiti dalle funzioni**

- Per far restituire un puntatore ad una funzione bisogna però seguire **degli accorgimenti particolari**
  - **Motivo:** allocazione della memoria
  - **La memoria in C viene gestita staticamente.** Ciò significa che bisogna sapere a priori quali variabili saranno utilizzate e quando saranno grandi.
    - **Questo è il motivo per cui le variabili devono tutte essere dichiarate prima dell'esecuzione e soprattutto bisogna assegnare un tipo alle variabili!**

Restituire un Puntatore

- Nuovamente
  - **I puntatori sono variabili a tutti gli effetti.**
  - **Possono essere dichiarate e possono essere utilizzate in array e struct**
  - **Possono ovviamente anche essere restituiti dalle funzioni**

- Per far restituire un puntatore ad una funzione bisogna però seguire **degli accorgimenti particolari**
  - **Motivo:** allocazione della memoria
  - **La memoria in C viene gestita staticamente.** Ciò significa che bisogna sapere a priori quali variabili saranno utilizzate e quando saranno grandi.
  - Nel caso dei puntatori può essere un problema.
    - **Perché?**

Restituire un Puntatore

- Nel caso dei puntatori può essere un problema.
  - Immaginiamo di dover scrivere una funzione che restituisce una stringa.
  - **Possiamo sapere a priori quanto sarà lunga una stringa? No!**

Restituire un Puntatore

- Nel caso dei puntatori può essere un problema.
  - Immaginiamo di dover scrivere una funzione che restituisce una stringa.
  - **Possiamo sapere a priori quanto sarà lunga una stringa? No!**
  - L'allocazione statica della memoria **non è sufficiente**
- **Soluzione:** Allocazione Dinamica della memoria
  - *(La studieremo meglio più avanti)*
  - Funzione **`malloc( ) e calloc( )`**
  - Serve a «riservare» una quantità di memoria al nostro programma, **non definita a priori**
- **Esempio:** scrivere una funzione che restituisce una Stringa

Restituire un Puntatore

- **Esempio:** scrivere una funzione che restituisce una Stringa

```
char* generatePassword (char* nome, char* cognome) {
     // alloco un numero sufficiente di caratteri
     char* s = (char*)  calloc(10, sizeof(char));

     ...

     // restituisco la nuova stringa
     return s;
}
```

Restituire un Puntatore

- **Esempio:** scrivere una funzione che restituisce una Stringa

```
char* generatePassword (char* nome, char* cognome) {
    // alloco un numero sufficiente di caratteri
    char* s = (char*)  calloc(10, sizeof(char));

    ...

    // restituisco la nuova stringa
    return s;
}
```

Chiede di allocare dinamicamente la memoria **per 10 caratteri**

Restituire un Puntatore

- **Esempio:** scrivere una funzione che restituisce una Stringa

```
char* generatePassword (char* nome, char* cognome) {
    // alloco un numero sufficiente di caratteri
    char* s = (char*)  calloc(10, sizeof(char));

    ...

    // restituisco la nuova stringa
    return s;
}
```

Chiede di allocare dinamicamente la memoria **per 10 caratteri**

Test: provate a eseguire un programma senza l'istruzione **`calloc`**. **Cosa succede?**

Esercizio 9.2

- Scrivere una funzione `char* generaPassword (char* nome, char* cognome)` che generi random una password per l'utente

- **La funzione prende in input nome e cognome (max. 20 caratteri)**
- **La funzione restituisce una stringa con i primi tre caratteri del nome, i primi tre caratteri del cognome e due simboli a caso**

- Inserire la funzione in un `main( )` che chieda interattivamente all'utente di inserire nome e cognome e stampi la password generata
  - **Utilizzare `calloc( )` per allocare la memoria.**

Esercizio 9.3

- Estendere l'Esercizio 9.2 scrivendo una funzione che offuschi la password generata dall'utente, visualizzando soltano i primi 2 e gli ultimi 2 caratteri.
  - **Input:** `catmus$#`
  - **Output:** `ca****$#`

- **Riflettere sul prototipo della funzione**
  - **Che opzioni abbiamo?**
    - **`char* offuscaPassword(char* password)`**

Esercizio 9.3

- Estendere l'Esercizio 9.2 scrivendo una funzione che offuschi la password generata dall'utente, visualizzando soltano i primi 2 e gli ultimi 2 caratteri.
  - **Input:** `catmus$#`
  - **Output:** `ca****$#`

- **Riflettere sul prototipo della funzione**
  - **Che opzioni abbiamo?**
    - **`char* offuscaPassword(char* password)`**
  - **Oppure**
    - **`void offuscaPassword(char* password)`**

Esercizio 9.3

- Estendere l'Esercizio 9.2 scrivendo una funzione che offuschi la password generata dall'utente, visualizzando soltano i primi 2 e gli ultimi 2 caratteri.
  - **Input:** `catmus$#`
  - **Output:** `ca****$#`

- **Riflettere sul prototipo della funzione**
  - **Che opzioni abbiamo?**
    - **`char* offuscaPassword(char* password)`**
  - **Oppure**
    - **`void offuscaPassword(char* password)`**

Sono entrambe corrette! La prima è più comprensibile e dà maggiormente l'idea di «restituire» un valore differente (offuscato). La seconda è comunque corretta, ma MODIFICA il valore originale della password! **Bisogna valutare gli effetti collaterali**

Puntatori Costanti

- Come visto nell'esercizio 9.3, quando passiamo un puntatore possiamo **modificare il contenuto puntato**
  - Questo potrebbe essere un comportamento non desiderato

Puntatori Costanti

- Come visto nell'esercizio 9.3, quando passiamo un puntatore possiamo **modificare il contenuto puntato**
  - Questo potrebbe essere un comportamento non desiderato

- **Preferiamo sempre il passaggio per valore**
  - In alcuni casi questo non è possibile
    - Array
  - In altri casi, questo non è conveniente
    - Strutture di dimensioni molto elevate

(torniamo a…) Puntatori Costanti

- Come visto nell'esercizio 9.3, quando passiamo un puntatore possiamo **modificare il contenuto puntato**
  - Questo potrebbe essere un comportamento non desiderato
- **Preferiamo sempre il passaggio per valore**
  - In alcuni casi questo non è possibile
    - Array
  - In altri casi, questo non è conveniente
    - Strutture di dimensioni molto elevate
- Il modificatore `const` indica al compilatore di **controllare possibili accessi in scrittura al contenuto puntato**

Puntatori Costanti

```
char* offuscaPassword(char* password)
  {
     // implementazione
  }
```

```
char* offuscaPassword(const char* password)
  {
     // implementazione
  }
```

Se siamo sicuri di non dover operare modifiche a un vettore (o più in generale, a una qualsiasi variabile presente nel codice!) , utilizziamo il quantificatore **`const` per indicare al compilatore che il valore di quella variabile non deve essere modificato durante l'esecuzione.**

Utile a prevenire comportamenti imprevedibili nel codice e a ridurre i bug.

Puntatori Costanti

- Quando passiamo un puntatore, possiamo **modificare il contenuto puntato**
  - Questo potrebbe essere un comportamento non desiderato
- **Preferiamo sempre il passaggio per valore**
  - In alcuni casi questo non è possibile
    - Array
  - **In altri casi, questo non è conveniente**
    - **Strutture di dimensioni molto elevate**

Approfondiamo.

Passaggio di Parametri con le **`struct`**

- Le variabili tradizionali vengono normalmente passate per valore.
- **Gli array vengono passati per riferimento**
- **…e le struct?**

Passaggio di Parametri con le **`struct`**

- Le variabili tradizionali vengono normalmente passate per valore.
- **Gli array vengono passati per riferimento**
- **…e le struct?**
  - Di default **`le struct`** vengono passate per valore.
  - Nel passaggio per valore viene effettuata **una copia dell'intera struttura.**

Passaggio di Parametri con le `struct`

- Le variabili tradizionali vengono normalmente passate per valore.
- **Gli array vengono passati per riferimento**
- **…e le struct?**
  - Di default **`le struct`** vengono passate per valore.
  - Nel passaggio per valore viene effettuata **una copia dell'intera struttura.**

- **Problema**
  - Per strutture particolarmente grandi, **l'operazione di copia può essere dispendiosa**
  - Il passaggio per riferimento può essere un'alternativa più efficiente.
    - **Si passa l'indirizzo della `struct` alla funzione**
    - Bisogna stare attenti a non modificare accidentalmente i valori, perché eventuali modifiche sarebbero ereditate dalla funzione chiamante

Passaggio di Parametri con le **`struct` - Esempio**

```
1   // Laboratorio di Informatica
2   // Informatica - TPS - 2016/2017
3   // docente: Cataldo Musto
4
5   #include <stdio.h>
6
7   int main() {
8
9   // Dichiaro una struct di tipo "data"
10  typedef struct {
11    int giorno;
12    char mese[15];
13    int anno;
14  } data ;
15
16  data d1;
17  data d2;
18
```

Passaggio di Parametri con le **`struct` - Esempio**

```
1   // Laboratorio di Informatica
2   // Informatica - TPS - 2016/2017
3   // docente: Cataldo Musto
4
5   #include <stdio.h>
6
7   int main() {
8
9   // Dichiaro una struct di tipo "data"
10  typedef struct {
11    int giorno;
12    char mese[15];
13    int anno;
14  } data ;
15
16  data d1;
17  data d2;
18
```

Qual è la dimensione di questa **`struct`?**

24 byte (4 + 15 + 4 + 1 padding)

Passaggio di Parametri con le **struct - Esempio**

```
1   // Laboratorio di Informatica
2   // Informatica - TPS - 2016/2017
3   // docente: Cataldo Musto
4
5   #include <stdio.h>
6
7   int main() {
8
9   // Dichiaro una struct di tipo "data"
10  typedef struct {
11    int giorno;
12    char mese[15];
13    int anno;
14  } data ;
15
16  data d1;
17  data d2;
18
```

```
// Prototipi di funzione
void function_data (data d);
void function_data_ptr (data* dPtr);

void function_data (data d) {
. . .
}
void function_data_ptr (data* dPtr) {
. . .
}

// main
int main() {
        function_data(d1);
        function_data_ptr(&d2);
}
```

Passaggio di Parametri con le **`struct` - Esempio**

```
1   // Laboratorio di Informatica
7   int main() {
8
9   // Dichiaro una struct di tipo "data"
10  typedef struct {
11    int giorno;
12    char mese[15];
13    int anno;
14  } data ;
15
16  data d1;
17  data d2;
18
```

Le funzioni sono identiche. La seconda però effettua il passaggio per riferimento, quindi è più efficiente. **Bisogna stare attenti agli 'effetti collaterali'**

```
// Prototipi di funzione
void function_data (data d);
void function_data_ptr (data* dPtr);

void function_data (data d) {
. . .
}
void function_data_ptr (data* dPtr) {
. . .
}

// main
int main() {
        function_data(d1);
        function_data_ptr(&d2);
}
```

Note Conclusive

- **Puntatori**
  - Particolare tipologia di variabili. Possono memorizzare solo **indirizzi di memoria**. Si inizializzano a NULL.
  - Per risalire all'indirizzo di una variabile si utilizza **l'operatore indirizzo (&)**
  - Per risalire al valore puntato da un indirizzo si usa **l'operatore di indirezione (*)**

Note Conclusive

- **Puntatori**
  - Particolare tipologia di variabili. Possono memorizzare solo **indirizzi di memoria**. Si inizializzano a NULL.
  - Per risalire all'indirizzo di una variabile si utilizza **l'operatore indirizzo (&)**
  - Per risalire al valore puntato da un indirizzo si usa **l'operatore di indirezione (*)**

- **Applicazioni**
  - **Passaggio dei parametri per riferimento.** Si passa a una funzione l'indirizzo di una variabile, invece del suo valore.
  - Obbligatorio per gli **`array`** (un array è già un puntatore al primo elemento)
  - Spesso utile per le **`struct`** (motivi di efficienza)
  - Valutare l'uso **per le altre tipologie di variabili**
    - Se la funzione ha bisogno di modificare la variabile (es. scambio valori), è necessario passare il parametro per indirizzo. Altrimenti è sufficiente il passaggio per valore (più sicuro.)

Note Conclusive (cont.)

- **Aritmetica dei Puntatori**
  - **I puntatori possono anche essere usati dentro espressioni matematiche**
  - Ci forniscono una modalità alternativa per accedere e modificare i valori negli array
  - *Imparare l'aritmetica dei puntatori: possibili esercizi.*

Note Conclusive (cont.)

- **Aritmetica dei Puntatori**
  - **I puntatori possono anche essere usati dentro espressioni matematiche**
  - Ci forniscono una modalità alternativa per accedere e modificare i valori negli array
  - *Imparare l'aritmetica dei puntatori: possibili esercizi.*

- **Utilizzi Avanzati**
  - **I puntatori possono essere inseriti a loro volta dentro `array` e dentro `struct`**
  - **Applicazione: array di stringhe == array di puntatori a `char`**
  - Se una funzione deve restituire **una stringa**, è necessario utilizzare **`malloc()`** e **`calloc()`**
  - Il quantificatore **`const`** serve ad aiutarci ad evitare modifiche non necessarie alle variabili. Utile quando preferiamo utilizzare i puntatori per motivi di efficienza, ma vogliamo evitare modifiche accidentali. **Utile adottarlo per identificare le variabili che non saranno modificate!**


DOMANDE?
imgflip.com