



UNIVERSITÀ  
degli STUDI  
di CATANIA

DIPARTIMENTO DI  
MATEMATICA E INFORMATICA

# Puntatori e Riferimenti

Alessandro Ortis

Image Processing Lab - [iplab.dmi.unict.it](http://iplab.dmi.unict.it)

[www.dmi.unict.it/ortis/](http://www.dmi.unict.it/ortis/)



# Puntatori

- una variabile di tipo “*puntatore al tipo x*” contiene l'indirizzo di memoria di una variabile di *tipo x*

```
int n;
```

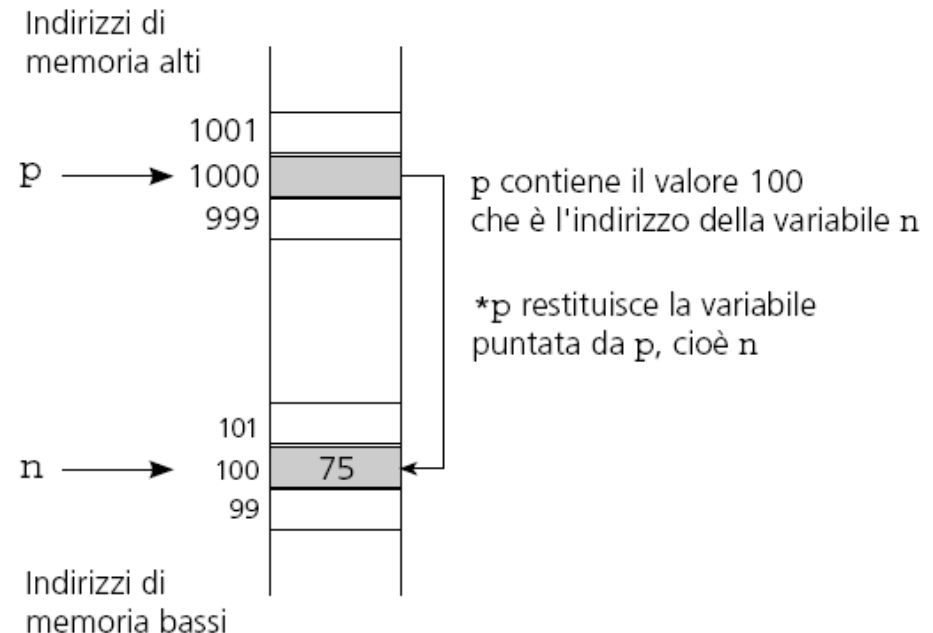
```
int* p; // p è variabile di tipo puntatore ad int
```

```
p = &n; // p contiene il valore dell'indirizzo di n
```

- *dereferenziare* il puntatore significa andare alla variabile puntata partendo da un suo puntatore

operatore di **indirizione**

- ~~\*~~p = 75;  
 // scrivo 75 nella  
 // variabile n



# Operazioni coi puntatori

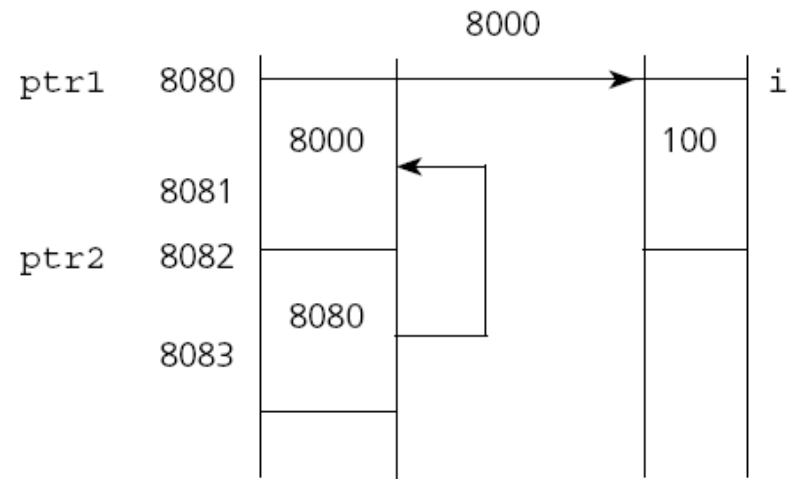
```
int i,j;  
int* p, q;  
p = &i;  
j = *p;  
q = p;  
*p = 3; // *p è un left value  
*q = *&j; // equivale a i = j  
*&j = 4; // equivale a j = 4  
if (p == &i) cout << "p punta la variabile i";  
if (q == p) cout << "q e p puntano la stessa variabile";  
if (p != 0) cout << "p punta a qualche variabile ...";  
if (p != NULL) cout << "... e quindi ...";  
if (p) cout << "... si può dereferenziare";  
cout << p; // stampa l'indirizzo in esadecimale
```

# Puntatori a puntatori

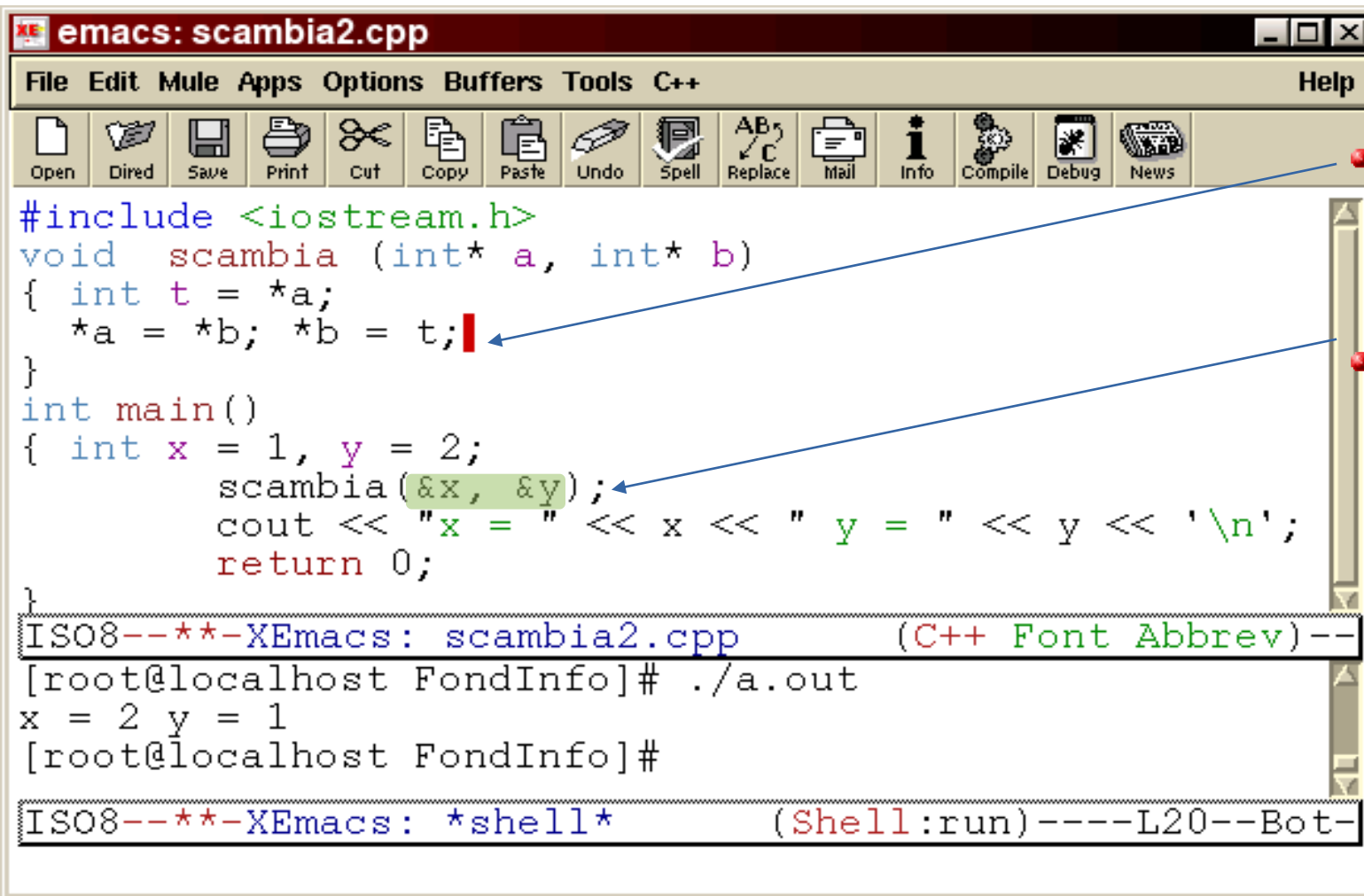
- un tipo puntatore può puntare qualunque tipo, anche un altro puntatore

```
int i = 1;  
int* ptr1 = &i;  
int** ptr2 = &ptr1;  
**ptr2 = 4; // equivale a i = 4
```

- `ptr1` è un puntatore ad interi e punta la variabile di tipo `int`  
`ptr2` è un puntatore a puntatore  
ad interi e punta la variabile `ptr1`



# Passaggio di puntatori a funzioni



```
emacs: scambia2.cpp
File Edit Mule Apps Options Buffers Tools C++ Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News

#include <iostream.h>
void scambia (int* a, int* b)
{ int t = *a;
  *a = *b; *b = t;
}
int main()
{ int x = 1, y = 2;
  scambia(&x, &y);
  cout << "x = " << x << " y = " << y << '\n';
  return 0;
}

ISO8--*-XEmacs: scambia2.cpp (C++ Font Abbrev)--
[root@localhost FondInfo]# ./a.out
x = 2 y = 1
[root@localhost FondInfo]#

ISO8--*-XEmacs: *shell* (Shell:run)----L20--Bot-
```

la funzione  
dereferenzia i  
puntatori

alla chiamata  
si passano gli  
indirizzi delle  
variabili come  
valore

# Restituzione argomenti puntatori

- le funzioni possono restituire puntatori

```
int* max(int* pa, int* pb)
{
    if (*pa >= *pb) return pa;
    return pb;
}

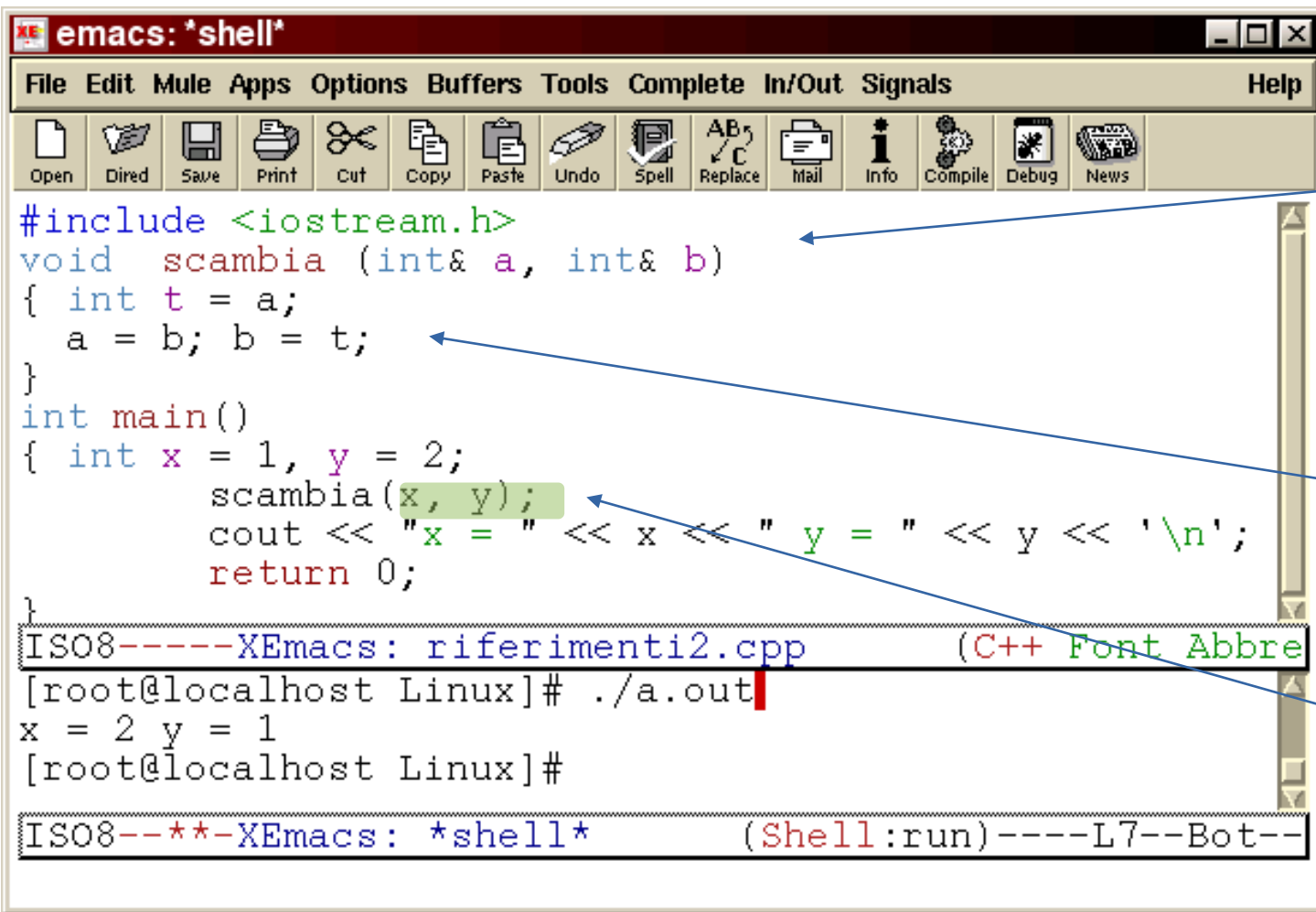
int main()
{ int x, y;
  cin >> x >> y;
  *max(&x, &y) = 0;
  cout << "x = " << x << " y = " << y << '\n';
  return 0;
}
```

# Riferimenti

- una variabile di tipo *riferimento al tipo x* è un *ulteriore nome* per una variabile x
  - `int n;`
  - `int& r = n;`
- una variabile di tipo “riferimento al tipo” deve essere inizializzata al momento della sua definizione

```
• void main()
• {
•     int n = 75;
•     int& r = n;          // r è un riferimento per n
•     cout << "n = " << n << ", r = " << r << endl;
•     cout << "&n = " << &n << ", &r = " << &r << endl;
• }
• esecuzione:
  n = 75, r = 75
  &n = 0x4fffd34, &r = 0x4fffd34
```

# Passaggio di riferimenti a funzioni



```
emacs: *shell*
File Edit Mule Apps Options Buffers Tools Complete In/Out Signals Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News

#include <iostream.h>
void scambia (int& a, int& b)
{ int t = a;
  a = b; b = t;
}
int main()
{ int x = 1, y = 2;
  scambia(x, y);
  cout << "x = " << x << " y = " << y << '\n';
  return 0;
}

ISO8-----XEmacs: riferimenti2.cpp (C++ Font Abbre
[root@localhost Linux]# ./a.out
x = 2 y = 1
[root@localhost Linux]#

ISO8--*-XEmacs: *shell* (Shell:run)----L7--Bot--
```

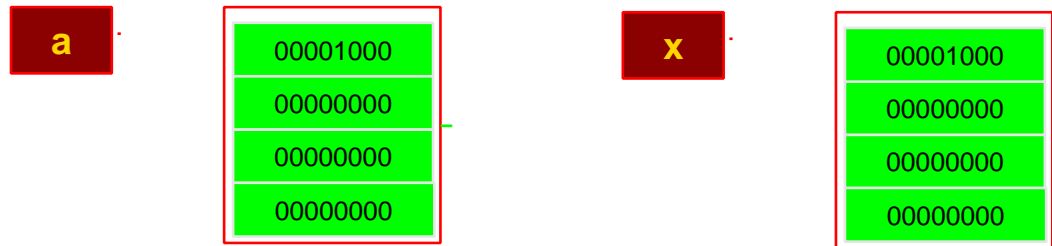
- i parametri formali sono dichiarati come **riferimenti**
- la funzione **non** dereferenzia puntatori
- alla chiamata si passano i nomi delle variabili



# Confronto passaggi

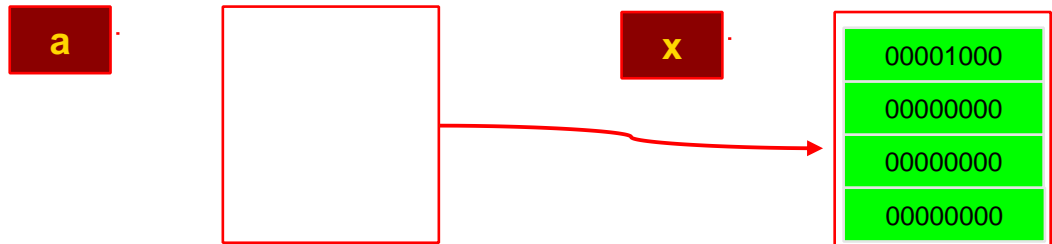
- passaggio per valore

```
int mia_funzione (int a) { ... }  
...  
int x = 8;  
...  
mia_funzione(x)
```



- passaggio per riferimento

```
int mia_funzione (int& a) { ... }  
...  
int x = 8;  
...  
mia_funzione(x)
```



# Restituzione argomenti riferimento

- le funzioni possono restituire riferimenti

```
int& max(int& ra, int& rb)
{
    if (ra >= rb) return ra;
    return rb;
}

int main()
{ int x, y;
  cin >> x >> y;
  max(x, y) = 0;
  cout << "x = " << x << " y = " << y << '\n';
  return 0;
}
```

# Riferimento: dietro le quinte

- un riferimento è un **puntatore costante**, il cui nome non è accessibile al programmatore, che ha per valore l'indirizzo dell'oggetto riferito: l'operazione che coinvolge l'oggetto riferito viene realizzata effettuando un'indirizzazione sul puntatore nascosto
- i riferimenti risultano utili per il programmatore, ma non aggiungono alcuna potenzialità ai programmi rispetto all'uso dei puntatori

# Confronto fra puntatori e riferimenti

Supponiamo che `p` valga `0x010`, qual è l'output del seguente codice ?

```
void func1(int * ptr) {  
    ptr = 0;  }  
void func2(int *& ptr) {  
    ptr = 0;  }  
  
int main() {  
    int* p;  
    cout << "p: " << p << endl;  
    func1(p);  
    cout << "p: " << p << endl;  
    func2(p);  
    cout << "p: " << p << endl;  
}
```

# Confronto fra puntatori e riferimenti

Supponiamo che `p` valga `0x010`, qual è l'output del seguente codice ?

```
void func1(int * ptr) {  
    ptr = 0; }  
void func2(int *& ptr) {  
    ptr = 0; }  
  
int main() {  
    int* p;  
    cout << "p: " << p << endl; // 0x010  
    func1(p);  
    cout << "p: " << p << endl; // 0x010  
    func2(p);  
    cout << "p: " << p << endl; // 0  
}
```

*func1()* modifica  
un parametro  
attuale

# Regola da ricordare su &

L'operatore & ha tre usi in C++:

- Usato come **prefisso** di una variabile ne restituisce l'indirizzo
- Usato come **suffisso** ad un tipo nella definizione di un riferimento, dichiara quest'ultimo come alias della variabile usata per la sua inizializzazione
- Usato come **suffisso** ad un tipo nella **dichiarazione** dei **parametri di una funzione** dichiara questi ultimi essere riferimenti delle variabili passati alla funzione

# Puntatori costanti e puntatori a costante

- puntatore costante

```
<tipo_dato>* const <nome_puntatore> = <indirizzo_variabile>;
```

```
int x, e;  
int* const p = &x;  
*p = e; // corretto  
p = &e // scorretto; p non puo' cambiare valore
```

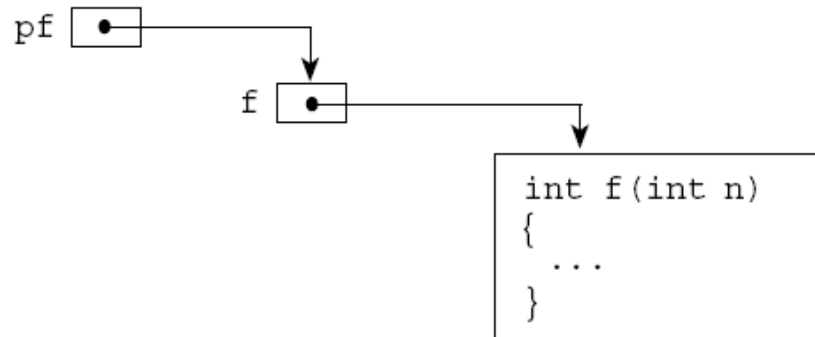
- puntatore a costante

```
const <tipo_dato>* <nome_puntatore> = <indirizzo_const>;
```

```
const int x = 25;  
const int e = 50;  
const int* p = &x;  
*p = 15; // scorretto; ciò che p punta non puo' cambiare  
valore  
p = &e // corretto
```

# Puntatori a funzioni

- non solo i dati, ma anche le istruzioni stanno in memoria a certi indirizzi; è possibile creare puntatori che puntino a funzioni, cioè al nome della funzione, che altro non è che un puntatore alla prima istruzione della funzione:
- `tipo_restituito (*PuntatoreFunzione) (argomenti);`
- `int f(int) { ... };` // definisce la funzione f
- `int (*pf)(int);` // definisce il puntatore pf alla funzione
- `pf = f;` // assegna l'indirizzo di f a pf





# Aritmetica dei puntatori

Se un puntatore a un tipo T viene incrementato di n, il suo valore viene in realtà incrementato di un **numero pari alla dimensione del tipo T espressa in byte**.

```
int x = 10;
int *p = &x;

cout << "&x " << &x << endl;
cout << "(&x +1)" << (&x + 1) << endl;

$ 0x004
$ ..???
```

# Aritmetica dei puntatori

Se un puntatore a un tipo T viene incrementato di n, il suo valore viene in realtà incrementato di un **numero pari alla dimensione del tipo T espressa in byte**.

```
int x = 10;  
int *p = &x;  
  
cout << "&x " << &x << endl;  
cout << "(&x +1)" << (&x + 1) << endl;
```

```
$ 0x004
```

```
$ 0x008
```

# Aritmetica dei puntatori

In generale, incrementare/decrementare di un numero intero  $n$  un puntatore significa incrementarlo/decrementarlo di un numero di byte pari a  **$n$  moltiplicato per la dimensione del tipo puntato.**

Basti pensare a quando incrementiamo di 1 un indice in un ciclo `for (i++)` per scorrere i valori di un array di interi. In realtà ogni incremento equivale ad uno spostamento di 4 byte (`sizeof di un int`).

# Aritmetica dei puntatori

La gestione degli array e dei puntatori da parte del C/C++ permette di affermare l'esistenza di un'equivalenza tra i puntatori e gli array.

La maggior parte della confusione riguardante questo argomento è data dal fraintendimento di questa assunzione.

Affermare che gli array e i puntatori sono equivalenti non significa assolutamente asserire che sono identici né che sono sempre intercambiabili.

In altre parole, **l'aritmetica dei puntatori e l'indicizzazione degli array sono equivalenti**, mentre puntatori e array sono differenti.

# Aritmetica dei puntatori

Il riferimento a un oggetto del tipo “array-di-T” che appare in un’espressione viene implicitamente convertito (decade), con alcune eccezioni, in un puntatore al suo primo elemento; il tipo del puntatore risultante è “puntatore-a-T”.

Per questo motivo possiamo ad esempio scrivere...

```
int arr = {1,2,3,4,5};  
int* p = arr; // equivale a int* p = &arr[0];
```

Posso infatti inizializzare un puntatore ad interi (p) con un array di interi (arr). Questo perché “arr” viene implicitamente convertito in un puntatore al suo primo elemento, ovvero

`&arr[0]`

# Aritmetica dei puntatori

Come sappiamo, possiamo interagire con un array tramite un puntatore mediante l'operatore [ ] in maniera equivalente.

```
int arr = {1,2,3,4,5};  
cout << arr[2];  
int* p = arr;  
cout << p[2];
```

# Aritmetica dei puntatori

```
int arr = {1,2,3,4,5};  
cout << arr[2];  
int* p = arr;  
cout << p[2];
```

Sfruttando l'aritmetica dei puntatori, l'indirizzo in “p” viene prelevato, incrementato di n e dereferenziato. Quindi:

$$p[n] \iff *(p + n)$$

Applicando l'operatore “[]” al nome di un array il discorso non cambia

$$arr[n] \iff *(&arr[0] + n)$$

# Esercizi

1. Scrivere un programma che mostri come un puntatore si può utilizzare per scorrere un array
2. Utilizzare l'aritmetica dei puntatori per stampare il contenuto di una matrice
3. Definire un metodo **void** che prende in input un array di interi e ne modifica il contenuto raddoppiando gli elementi di indice pari e triplicando quelli di indice dispari.