

Puntatori

"The name of the song is called "Haddocks' Eyes".

"Oh, that's the name of the song, is it?" Alice said, trying to feel interested.

"No, you don't understand," the Knight said, looking a little vexed.

"That's what the name is called. The name really is 'The Aged Aged Man.' "

"Then I ought to have said 'That's what the song is called' ?" Alice corrected herself.

"No, you oughtn't: that's quite another thing! The song is called 'Ways and Means,' but that is only what it's called, you know!"

"Well, what is the song, then?" said Alice, who was by this time completely bewildered.

"I was coming to that," the Knight said. "The song really is 'A-sitting On A Gate,' and the tune's my own invention."

Lewis Carroll. Through the Looking Glass.

Carroll sottolinea la distinzione tra le cose, i nomi delle cose e i nomi dei nomi delle cose.

Puntatori

L'astrazione offerta dai puntatori cattura la distinzione tra una cosa e il suo nome:

COSA

pagina web dipartimento

indirizzo e-mail

numero ufficio

studente

int a

NOME

`http://pages.di.unipi.it/bodei/`

`chiara.bodei@unipi.it`

`322DO`

`445654`

`&a`

Cosa è una variabile?

Quando si dichiara una variabile, ad es. `int a;` si rende noto il nome e il tipo della variabile. Il compilatore

- ▶ alloca l'opportuno numero di byte di memoria per contenere il valore associato alla variabile (ad es. `4`).
- ▶ aggiunge il simbolo `a` alla tavola dei simboli e l'indirizzo del blocco di memoria ad esso associato (ad es. `A010` che è un indirizzo esadecimale)
- ▶ Se poi troviamo l'assegnamento `a = 5;` ci aspettiamo che al momento dell'esecuzione il valore `5` venga memorizzato nella locazione di memoria assegnata alla variabile `a`
- ▶ In C è possibile conoscere l'indirizzo della cella di memoria in cui è memorizzata una variabile. Per ottenere l'indirizzo di `a` basta usare l'operatore di indirizzamento `&a`.

A00E	...	
A010	5	← <code>a</code> dichiarata come <code>int a = 5;</code>
A012	...	

Cosa è una variabile?

Alla variabile **a** si associa quindi:

- ▶ il valore della locazione di memoria, ovvero l'indirizzo **A010** e
- ▶ il valore dell'intero che vi viene memorizzato, ovvero **5**.
- ▶ Nell'espressione **a = 5**; con **a** ci riferiamo alla locazione di memoria associata alla variabile (**lvalue**): il valore **5** viene copiato all'indirizzo **A010**.
- ▶ nell'espressione **b = a**; (dove **b** è ancora un intero) **a** si riferisce al valore (**rvalue** o **object**): il valore associato ad **a** viene copiato all'indirizzo di **b**

È ragionevole avere anche variabili apposite che memorizzino indirizzi e che dipendano dal tipo di variabili memorizzate a quegli indirizzi.

A00E	...	
A010	5	← a
A012	A010	← pi

Puntatori

- ▶ Proprietà della variabile `a` nell'esempio:

nome: `a`

tipo: `int`

valore: `5`

indirizzo: `A010` (che è fissato una volta per tutte)

- ▶ In C è possibile **denotare** e quindi **manipolare** gli indirizzi di memoria in cui sono memorizzate le variabili.
- ▶ Abbiamo già visto nella `scanf`, l'**operatore indirizzo** “&”, che applicato ad una variabile, denota l'indirizzo della cella di memoria in cui è memorizzata (nell'es. `&a` ha valore `0xA010`).
- ▶ Le variabili di tipo **puntatore** o **puntatori** servono per memorizzare gli indirizzi in memoria delle variabili.

Tipo di dato: Puntatore

Un **puntatore** è una variabile che contiene l'indirizzo in memoria di un'altra variabile (del tipo dichiarato)

Esempio: dichiarazione `int *pi;`

- ▶ La variabile `pi` è di tipo **puntatore a intero**
- ▶ È una variabile come tutte le altre, con le seguenti proprietà:

nome: `pi`

tipo: **puntatore ad intero** (ovvero, indirizzo di un intero)

valore: inizialmente casuale

indirizzo: fissato una volta per tutte

- ▶ Più in generale:

Sintassi `tipo *variabile;`

- ▶ Al solito, più variabili dello stesso tipo possono essere dichiarate sulla stessa linea

`tipo *variabile1, ..., *variabilen;`

Esempio:

```
int *pi1, *pi2, i, *pi3, j;  
float *pf1, f, *pf2;
```

- ▶ Abbiamo dichiarato:

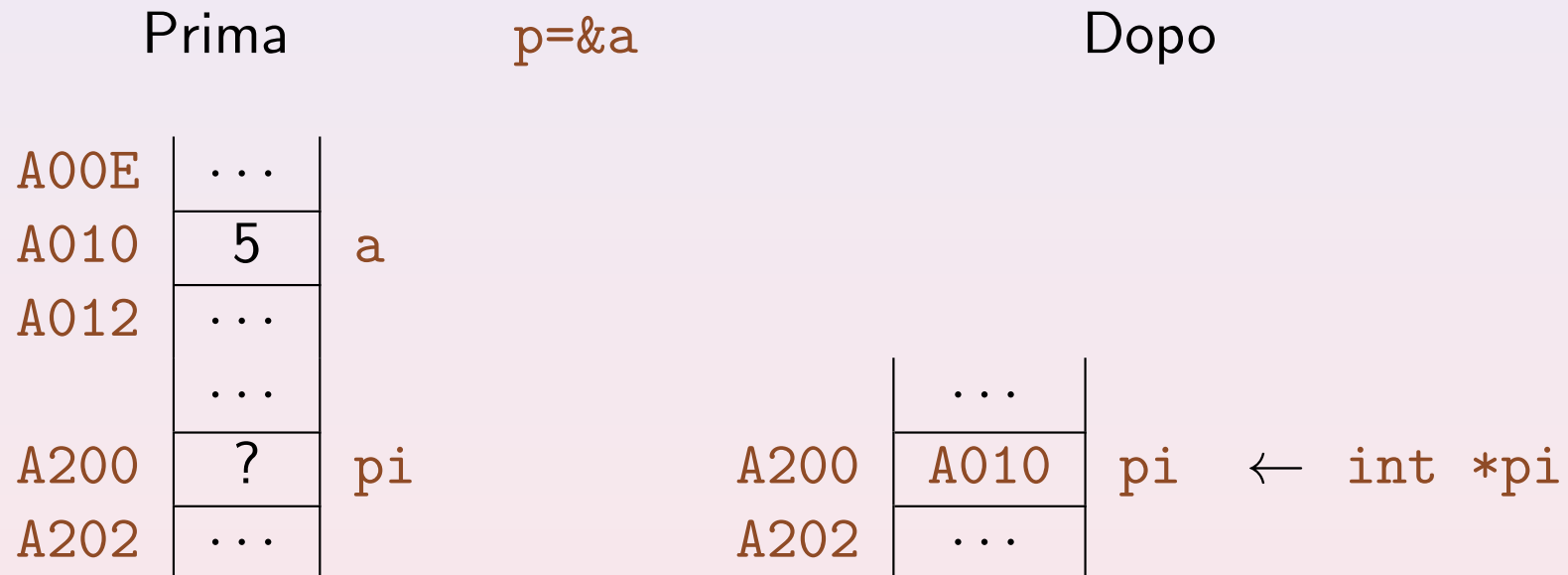
```
pi1, pi2, pi3  di tipo puntatore ad int  
i, j           di tipo int  
pf1, pf2      di tipo puntatore a float  
f             di tipo float
```

- ▶ Una variabile puntatore può essere inizializzata usando l'operatore di indirizzo.

Esempio: `pi = &a;`

- ▶ il valore di `pi` viene inizializzato all'indirizzo della variabile `a`
- ▶ si dice che `pi` **punta** ad `a` o che `a` è l'**oggetto puntato** da `pi`
- ▶ lo rappresenteremo spesso come nel seguito:





Operatore di dereferenziazione “*”

- ▶ Si può **accedere al contenuto** della cella di memoria puntata, attraverso l'operatore “*”.

- ▶ * fa riferimento all'oggetto puntato, mentre & all'indirizzo.

- ▶ può essere utilizzato per l'assegnamento **Esempio:**

```
int *pi;    /* dich. di puntatore ad intero */  
int a = 5, b; /* dich. variabili intere */
```

```
pi = &a;    /* pi punta ad a ==> *pi sta per a */  
b = *pi;   /* assegna a b il valore della var.puntata  
           da pi, ovvero il valore di a: 5 */  
*pi = 9;   /* assegna 9 alla variabile puntata da pi,  
           ovvero ad a */
```

- ▶ Non confondere le due occorrenze di “*”:

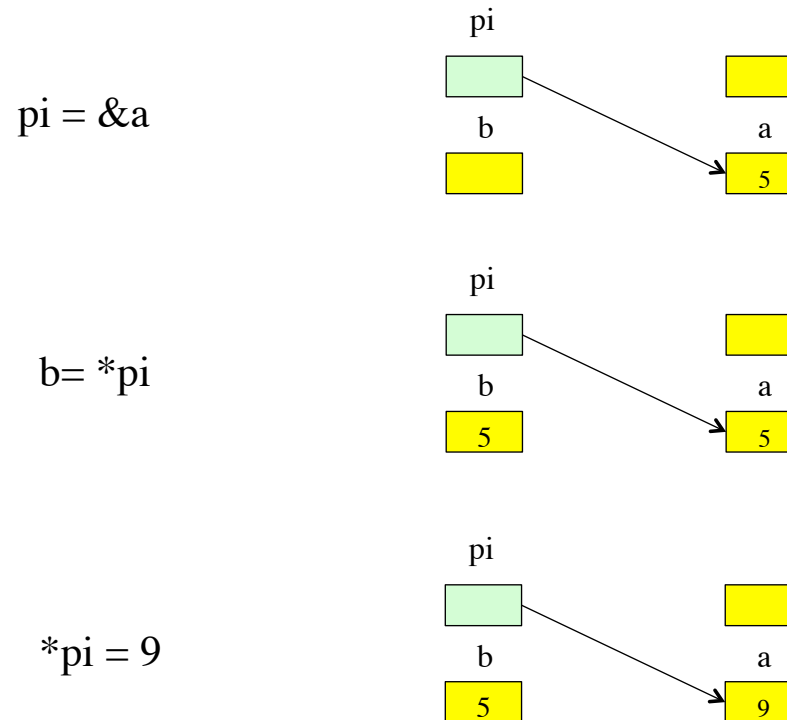
- ▶ “*” in una dichiarazione serve per dichiarare una variabile di tipo puntatore, ad es. **int *pi;**

- ▶ “*” in un'espressione è l'operatore di dereferenziazione, ad es. **b = *pi;**

Operatori di dereferenziamento “*” e di indirizzo “&”

- ▶ hanno priorità più elevata degli operatori binari, ad es. in `y = *pi + 2`, prima si incrementa l'oggetto puntato da `pi` e poi si assegna il risultato a `y`.
- ▶ “*” è associativo a destra
Es.: `**p` è equivalente a `*(*p)`
- ▶ “&” può essere applicato **solo** ad una variabile;
`&a` non è una variabile \implies “&” non è associativo
- ▶ “*” e “&” sono uno l'inverso dell'altro
 - ▶ data la dichiarazione `int a;`
`*&a` è un modo alternativo per denotare `a` (sono entrambi variabili)
 - ▶ data la dichiarazione `int *pi;`
`&*pi` ha valore (un indirizzo) uguale al valore di `pi`
però:
 - `pi` è una variabile
 - `&*pi` non lo è (ad es., non può essere usato a sinistra di “=”)

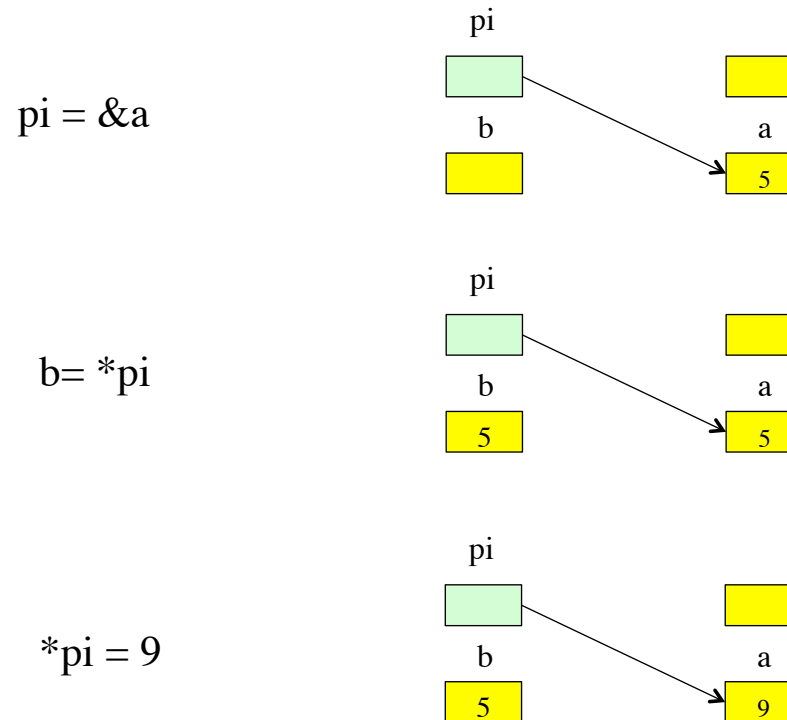
Operatori di dereferenziamento “*” e di indirizzo “&”



Ricapitolando

- ▶ se `a` è un intero, allora `&a` è un puntatore a `a` (indirizzo di memoria di `a`)
- ▶ `pi` è un puntatore a un intero, allora `*pi` è l'intero e può apparire dovunque possa apparire l'intero corrispondente `a`. Ad es.:
`*pi = *pi + 1`

Operatori di dereferenziazione “*” e di indirizzo “&”



Stampa di puntatori

- ▶ I puntatori si possono stampare con `printf` e specificatore di formato `“%p”` (stampa in formato esadecimale).

Esempio:

A00E	...	
A010	5	a
A012	A010	pi
	...	

```
int a = 5, *pi;
pi = &a;
printf("ind. di a = %p\n", &a); /* stampa 0xA010 */
printf("val. di pi = %p\n", pi); /* stampa 0xA010 */
printf("val. di *&pi = %p\n", *&pi); /* stampa 0xA010 */
printf("val. di a = %d\n", a); /* stampa 5 */
printf("val. di *pi = %d\n", *pi); /* stampa 5 */
printf("val. di *&a = %d\n", *&a); /* stampa 5 */
```

- ▶ Si può usare `%p` anche con `scanf`, ma ha poco senso leggere un indirizzo.

Esempio: Scambio del valore di due variabili.

```
int a = 10, b = 20, temp;  
temp = a;  
a = b;  
b = temp;
```

Tramite puntatori:

```
int a = 10, b = 20, temp;  
int *pa, *pb;  
  
pa = &a;    /* *pa diventa un alias per a */  
pb = &b;    /* *pb diventa un alias per b */  
  
temp = *pa;  
*pa = *pb;  
*pb = temp;
```

Inizializzazione di variabili puntatore

- ▶ I puntatori (come tutte le altre variabili) devono essere inizializzati prima di poter essere usati.

⇒ È un **errore** dereferenziare una variabile puntatore non inizializzata.

Esempio: `int a, *pi;`

A00E	...	
A010	?	a
A012	F802	pi
	...	
F802	412	
F804	...	

`a = *pi;` ⇒ ad **a** viene assegnato il valore **412**

`*pi = 500;` ⇒ scrive **500** nella cella di indirizzo **F802**

- ▶ Non sappiamo a cosa corrisponde questa cella di memoria!!!
⇒ la memoria può venire corrotta

Tipo di variabili puntatore

- ▶ Il tipo di una variabile puntatore è “puntatore a **tipo**”. Il suo valore è un **indirizzo**.
- ▶ I tipi puntatore sono **indirizzi** e **non interi**.

```
int a, *pi;  
a = pi;
```

- ▶ Compilando si ottiene un avvertimento o *warning*:
“assignment makes integer from pointer without a cast”
- ▶ Due variabili di tipo **puntatore a tipi diversi sono incompatibili**.

```
int x, *pi; float *pf;  
x = pi;      assegnazione int* a int  
             warning: “assignment makes integer from pointer ...”  
pf = x;      assegnazione int a float*  
             warning: “assignment makes pointer from integer ...”  
pi = pf;     assegnazione float* a int*  
             warning: “assignment from incompatible pointer type”
```

- ▶ Perché il C distingue tra puntatori di tipo diverso?
- ▶ Se tutti i tipi puntatore fossero identici non sarebbe possibile determinare a tempo di compilazione il tipo di `*p`.

Esempio:

```
puntatore p;  
int i; char c; float f;
```

- ▶ Potrei scrivere:

```
p = &c;  
p = &i;  
p = &f;
```

- ▶ Il tipo di `*p` verrebbe a dipendere dall'ultima assegnazione che è stata fatta (nota solo a tempo di esecuzione).
- ▶ Ad esempio, quale sarebbe il significato di `/` in `i/*p`: divisione intera o reale?

Possibili rischi con i puntatori: aliasing

```
*pi = 3;  
*qi = 7;  
pi = qi;  
*qi = 5;
```

- ▶ Adesso il valore `*qi` puntato da `qi` è 5, ma come **effetto collaterale** 5 è anche il valore `*pi` puntato da `pi`.
- ▶ Il problema è una conseguenza dell'*aliasing*, ovvero del fatto che uno stesso oggetto venga chiamato in due modi diversi.
- ▶

Funzione `sizeof` con puntatori

- ▶ La funzione `sizeof` restituisce l'occupazione in memoria in byte di una variabile (anche di tipo `puntatore`) o di un tipo.
- ▶ I puntatori occupano lo spazio di un indirizzo.
- ▶ L'oggetto puntato ha invece la dimensione del tipo puntato.

```
char *pc;
```

```
int *pi;
```

```
double *pd;
```

```
printf("%d %d %d ", sizeof(pc), sizeof(pi), sizeof(pd));
```

```
printf("%d %d %d\n", sizeof(char *), sizeof(int *),  
        sizeof(double *));
```

```
printf("%d %d %d ", sizeof(*pc), sizeof(*pi), sizeof(*pd));
```

```
printf("%d %d %d\n", sizeof(char), sizeof(int),  
        sizeof(double));
```

```
4 4 4   4 4 4
```

```
1 2 8   1 2 8
```

Operazioni con puntatori

Sui puntatori si possono effettuare diverse **operazioni**:

- ▶ di **dereferenziamento**

Esempio:

```
int *p, i;
```

```
...
```

```
i = *p;
```

Il valore della variabile intera **i** è ora lo stesso del valore dell'intero puntato da **p**.

- ▶ di **assegnamento**

Esempio: `int *p, *q;`

```
...
```

```
p = q;
```

- ▶ N.B. **p** e **q** devono essere dello stesso tipo (altrimenti bisogna usare l'operatore di cast).

Dopo l'assegnamento precedente, **p** punta allo stesso intero a cui punta **q**.

► di confronto

Esempio:

```
if (p == q) ...
```

I due puntatori hanno lo stesso valore.

Esempio:

```
if (p > q) ...
```

Ha senso? Con quello che abbiamo visto finora no. Vedremo che ci sono situazioni in cui ha senso.

Aritmetica dei puntatori

Sui puntatori si possono anche effettuare operazioni **aritmetiche**, con opportune limitazioni

- ▶ **somma** o **sottrazione** di un intero
- ▶ **sottrazione** di un puntatore da un altro

Somma e sottrazione di un intero

Se p è un puntatore a **tipo** e il suo valore è un certo indirizzo **ind**, il significato di $p+1$ è il primo indirizzo utile dopo **ind** per l'accesso e la corretta memorizzazione di una variabile di tipo **tipo**.

Esempio:

```
int *p, *q;
```

```
.....
```

```
q = p+1;
```

Se il valore di p è l'indirizzo **100**, il valore di q dopo l'assegnamento è **104** (assumendo che un intero occupi 4 byte).

- ▶ Il valore calcolato in corrispondenza di un'operazione del tipo $p+i$ dipende dal tipo T di p (analog. per $p-i$):

Op. Logica: $p = p+1$ Op. Algebrica: $p = p + \text{sizeof}(T)$

Esempio:

```
int *pi;  
*pi = 15;  
pi=pi+1;     $\implies$   $pi$  punta al prossimo int (4 byte dopo)
```

Esempio:

```
double *pd;  
*pd = 12.2;  
pd = pd+3;     $\implies$   $pd$  punta a 3 double dopo (24 byte dopo)
```

Esempio:

```
char *pc;  
*pc = 'A';  
pc = pc - 5;     $\implies$   $pc$  punta a 5 char prima (5 byte prima)
```

- ▶ Possiamo anche scrivere: $pi++$; $pd+=3$; $pc-=5$;
- ▶ Questa operazione ci servirà per i puntatori ad array.

Puntatore a puntatore

- ▶ Le variabili di tipo puntatore sono variabili come tutte le altre: in particolare hanno un **indirizzo** che può costituire il valore di un'altra variabile di tipo **puntatore a puntatore**.

Esempio:

```
int *pi, **ppi, x=10;
pi = &x;
ppi = &pi;
printf("pi = %p ppi = %p *ppi = %p\n", pi, ppi, *ppi);
printf("*pi = %d **ppi = %d x = %d\n", *pi, **ppi, x);
```

```
pi = 0x22ef34    ppi = 0x22ef3c    *ppi = 0x22ef34
*pi = 10        **ppi = 10        x = 10
```

Esempi

```
int a, b, *p, *q;  
a=10;  
b=20;  
p = &a;  
q = &b;  
*q = a + b;  
a = a + *q;  
q = p;  
*q = a + b;  
printf("a=%d b=%d *p=%d *q=%d", a,b,*p,*q);
```

Quali sono i valori stampati dal programma?

Esempi (contd.)

```
int *p, **q;  
int a=10, b=20;  
q = &p;  
p = &a;  
*p = 50;  
**q = 100;  
*q = &b;  
*p = 50;  
a = a+b;  
printf("a=%d   b=%d   *p=%d   **q=%d\n", a, b, *p, **q);
```

Quali sono i valori stampati dal programma?

Relazione tra vettori e puntatori

- ▶ In generale non sappiamo cosa contengono le celle di memoria adiacenti ad una data cella.
- ▶ L'unico caso in cui sappiamo quali sono le locazioni di memoria successive e cosa contengono è quando utilizziamo dei vettori.
- ▶ In C il **nome di un vettore** `vet` è in realtà un **puntatore** alla prima locazione dell'array `vet[0]`. (*Infatti non posso assegnare un array ad un altro*)

`int vet[10];` \implies `vet` e `&vet[0]` hanno lo stesso valore (un indirizzo), basta provare a stamparli

- ▶ Un puntatore può puntare il primo elemento di un vettore.

```
int vet[5];
```

```
int *pi;
```

```
pi = vet;      è equivalente a   pi = &vet[0];
```

Accesso agli elementi di un vettore

Esempio:

```
int vet[5];  
int *pi = vet;  
*(pi + 3) = 28;
```

- ▶ `pi+3` punta all'elemento di indice **3** del vettore (il quarto elemento).
- ▶ **3** viene detto **offset** (o scostamento) del puntatore.
- ▶ N.B. Servono le `()` perchè `*` ha priorità maggiore di `+`. Che cosa denota `*pi + 3` ?
- ▶ Osservazione:
`&vet[3]` equivale a `pi+3` equivale a `vet+3`
`*&vet[3]` equivale a `*(pi+3)` equivale a `*(vet+3)`
- ▶ Inoltre, `*&vet[3]` equivale a `vet[3]`
 - ▶ In C, `vet[3]` è solo un modo alternativo di scrivere `*(vet+3)`.
- ▶ Notazioni per gli elementi di un vettore:
 - ▶ `vet[3]` \implies notazione con **puntatore e indice**
 - ▶ `*(vet+3)` \implies notazione con **puntatore e offset**

- ▶ Un esempio che riassume i modi in cui si può accedere agli elementi di un vettore.

```
int vet[5] = {11, 22, 33, 44, 55};  
int *pi = vet;  
int offset = 3;
```

```
/* assegnamenti equivalenti */
```

```
vet[offset] = 88;  
*(vet + offset) = 88;  
pi[offset] = 88;  
*(pi + offset) = 88;
```

- ▶ **Attenzione:** a differenza di un normale puntatore, il nome di un vettore è un puntatore **costante**
 - ▶ il suo valore **non** può essere modificato!

- ▶ `int vet[10];`
`int *pi;`

`pi = vet;` **corretto**

`pi++;` **corretto**

`vet++;` **scorretto:** `vet` è un puntatore costante!

- ▶ È questo il vero motivo per cui non è possibile assegnare un vettore ad un altro utilizzando i loro nomi

```
int a[3]={1,1,1}, b[3] i;  
for (i=0; i<3; i++)  
    b[i] = a[i];
```

ma non `b=a` (`b` è un puntatore costante!)

Modi alternativi per scandire un vettore

```
int a[LUNG]= {.....};  
int i, *p=a;
```

- ▶ I seguenti sono tutti modi equivalenti per stampare i valori di **a**

```
for (i=0; i<LUNG; i++)  
    printf("%d", a[i]);
```

```
for (i=0; i<LUNG; i++)  
    printf("%d", p[i]);
```

```
for (i=0; i<LUNG; i++)  
    printf("%d", *(a+i));
```

```
for (i=0; i<LUNG; i++)  
    printf("%d", *(p+i));
```

```
for (p=a; p<a+LUNG; p++)  
    printf("%d", *p);
```

- ▶ Non è invece lecito un ciclo del tipo

```
for ( ; a<p+LUNG; a++)  
    printf("%d", *a);
```

perché? Perché `a++` è un assegnamento sul puntatore costante `a!`.

Differenza tra puntatori

- ▶ Il parallelo tra vettori e puntatori ci consente di capire il senso di un'operazione del tipo $p-q$ dove p e q sono puntatori allo stesso tipo.

```
int *p, *q;  
int a[10]={0};  
int x;  
...  
x=p-q;
```

- ▶ Il valore di x è il numero di interi compresi tra l'indirizzo p e l'indirizzo q .
- ▶ Quindi se nel codice precedente $...$ sono le istruzioni:

```
q = a;  
p = &a[5];
```

il valore di x dopo l'assegnamento è 5.

Esempio

```
double b[10] = {0.0};  
double *fp, *fq;  
char    *cp, *cq;
```

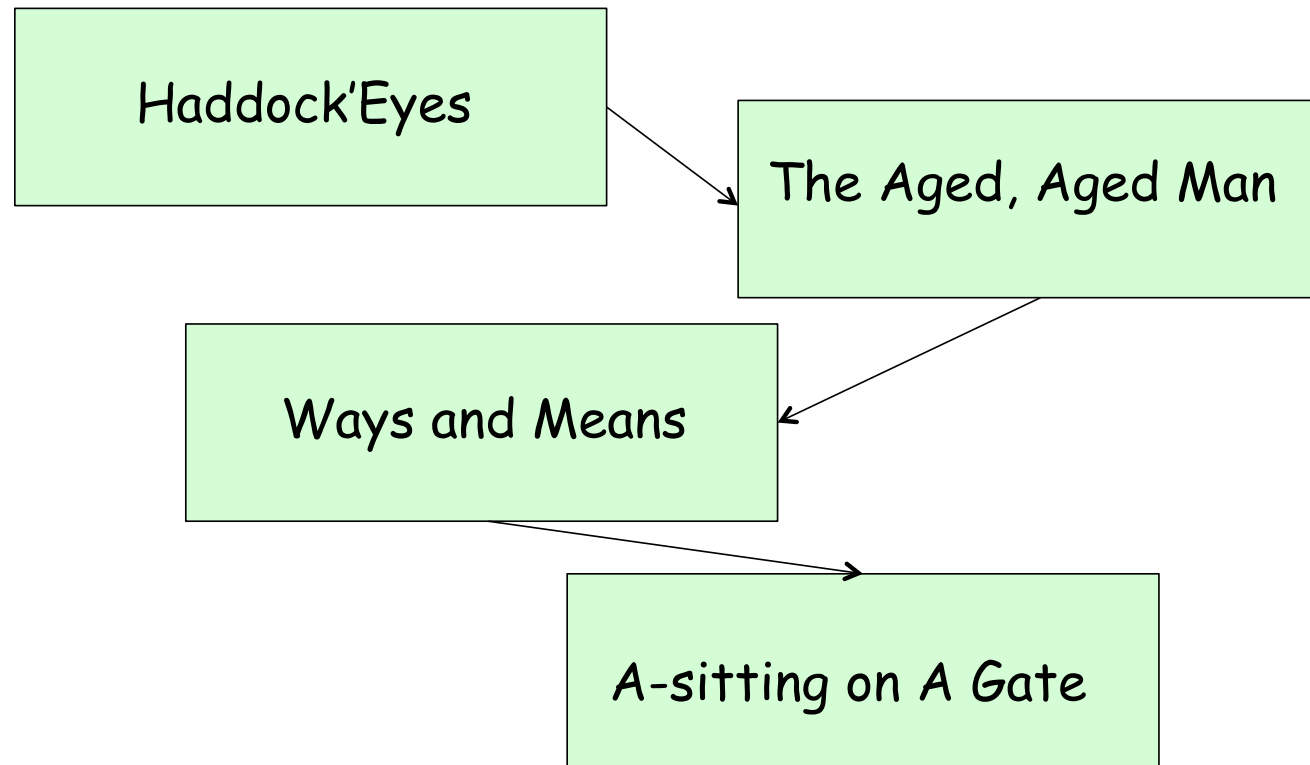
```
fp = b+5;  
fq = b;
```

```
cp = (char *) (b+5);  
cq = (char *) b;
```

```
printf("fp=%p  cp=%p  fq=%p  cq=%p\n", fp, cp, fq, cq);  
printf("fp-fq= %d,  cp-cq=%d\n", fp-fq, cp-cq);
```

```
fp=0x22fe3c  cp=0x22fe3c  fq=0x22fe14  cq=0x22fe14  
fp-fq=5     cp-cq=40
```

Diagramma di Alice



Passaggio dei parametri per indirizzo nelle funzioni

- ▶ Per poter modificare il valore dei parametri attuali, alcuni linguaggi (es. Pascal) prevedono un'ulteriore modalità di passaggio dei parametri, il passaggio **per indirizzo**
 - ▶ che informalmente, fa in modo che, al momento della chiamata, il parametro formale costituisca un modo **alternativo** per accedere al parametro attuale (che **deve** essere una variabile e non una generica espressione)
 - ▶ durante l'esecuzione del corpo, ogni riferimento (e/o modifica) al parametro formale è un riferimento al parametro attuale.
- ▶ In C esiste solo il passaggio per valore, ma quello per indirizzo si può realizzare come segue:
 1. si utilizza un parametro formale di tipo **puntatore**
 2. all'interno del corpo della procedura ogni riferimento al parametro formale avviene attraverso l'operatore di dereferenziazione ***** (parametro attuale e il parametro formale si riferiscono alla stessa cella)
 3. al momento della chiamata, si utilizza come parametro attuale un indirizzo di una variabile (utilizzando, se necessario, l'operatore di indirizzo **&**).

Esempio: Riprendiamo l'esempio del valore assoluto.

```
void abs(int *x)
{
    if (*x < 0)
        *x = -(*x);
}
```

- ▶ Nel chiamante si utilizzano chiamate del tipo `abs(&z)` per ottenere l'effetto di rimpiazzare il valore della variabile `z` con il suo valore assoluto.

```
int z = -5;
abs(&z);    ⇒    x = &z;
              if (*x < 0) *x = -(*x);
```

- ▶ **N.B.** Il passaggio dei parametri è sempre per **valore**, ma questa volta viene passato un valore puntatore che consente di accedere alla variabile del chiamante.
- ▶ Nell'esempio, l'assegnamento `*x = -(*x);` ha come effetto la modifica della variabile `z` del chiamante, in quanto il valore di `x` è `&z` e dunque `*x` è proprio `z`.

Esempio: Procedura per lo scambio di due variabili intere

```
void swap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

- ▶ Esempio di utilizzo: programma che legge due valori interi e li stampa ordinati.

```
main()    {
    int a, b;
    scanf("%d", &a);
    scanf("%d", &b);
    if (a > b) swap(&a, &b);
    printf("Valore minimo: %d\n", a);
    printf("Valore massimo: %d\n", b);    }
```