

## 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`

A00E	...
A010	5
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: il valore **5** viene copiato a quell'indirizzo.
  - ▶ nell'espressione **b = a**; (dove **b** è ancora un intero) **a** si riferisce al valore: il valore associato ad **a** viene copiato all'indirizzo di **b**
- È ragionevole avere anche variabili che memorizzino indirizzi.

# 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`).
- ▶ Gli indirizzi si utilizzano nelle variabili di tipo **puntatore**, dette semplicemente **puntatori**.

## 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 *variabile-1, ..., *variabile-n;
```

**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 così':



Prima

`p=&a`

Dopo

A00E	...	
A010	5	a
A012	...	
	...	
A200	?	pi
A202	...	

	...	
A200	A010	pi
A202	...	

## Operatore di dereferenziazione “\*”

- ▶ Applicato ad una variabile puntatore fa riferimento all'oggetto puntato. (mentre `&` fa riferimento all'indirizzo)

### 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 */
```

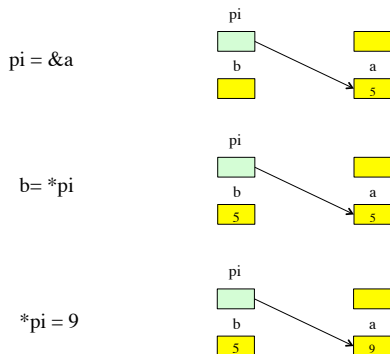
- ▶ N.B. Se `pi` è di tipo `int *`, allora `*pi` è di tipo `int`.
- ▶ Non confondere le due occorrenze di “\*”:
  - ▶ “\*” in una dichiarazione serve per dichiarare una variabile di tipo puntatore, es. `int *pi;`
  - ▶ “\*” in un'espressione è l'operatore di dereferenziazione, es. `b = *pi;`

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

- ▶ hanno priorità più elevata degli operatori binari
- ▶ “\*” 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 esempio, non può essere usato a sinistra di “=”)



# 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 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?

## 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("%lu %lu %lu ", sizeof(pc), sizeof(pi), sizeof(pd));
printf("%lu %lu %lu\n", sizeof(char *), sizeof(int *),
        sizeof(double *));
printf("%lu %lu %lu ", sizeof(*pc), sizeof(*pi),
        sizeof(*pd));
printf("%lu %lu %lu\n", sizeof(char), sizeof(int),
        sizeof(double));
```

4	4	4	4	4	4
1	4	8	1	4	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;`

## 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** è in realtà un **puntatore**, inizializzato all'indirizzo dell'elemento di indice 0.

```
int vet[10]; ⇒ vet e &vet[0] hanno lo stesso valore (un indirizzo)  
⇒ printf("%p %p", vet, &vet[0]); stampa 2 volte lo stesso  
indirizzo.
```

- ▶ Possiamo far puntare un puntatore al 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:
 

<code>&amp;vet[3]</code>	equivale a	<code>pi+3</code>	equivale a	<code>vet+3</code>
<code>*&amp;vet[3]</code>	equivale a	<code>*(pi+3)</code>	equivale a	<code>*(vet+3)</code>
- ▶ 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` e' 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
```