

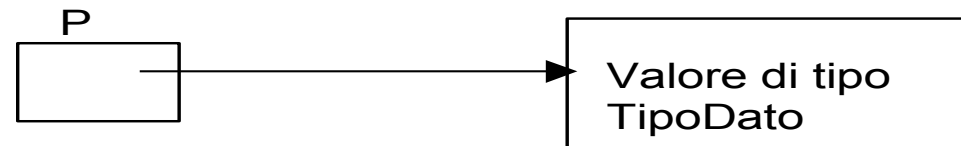
Puntatori in C

Il costruttore puntatore

- Ritorniamo sul problema dell'accesso alle variabili
 - Nel linguaggio di von Neumann attraverso il loro indirizzo
 - Nei linguaggi di alto livello attraverso il loro nome
- Però in taluni casi si vuol fare riferimento a celle diverse mediante lo stesso simbolo:
 - Ad es., $a[i]$ denota una cella diversa a seconda del valore di i
- Introduciamo ora un nuovo meccanismo di accesso simbolico legato all'indirizzo delle variabili
 - Potente e indispensabile nel linguaggio C, ma pericoloso e spesso difficile da manipolare



Dichiarazione e uso



```
typedef    TipoDato    *TipoPuntatore;  
TipoPuntatore    P;  
TipoDato    x;
```

Dereferenziazione: *P indica la cella di memoria il cui indirizzo è contenuto in P; essa può essere utilizzata esattamente come una qualsiasi altra variabile di tipo TipoDato, per cui sono legali...

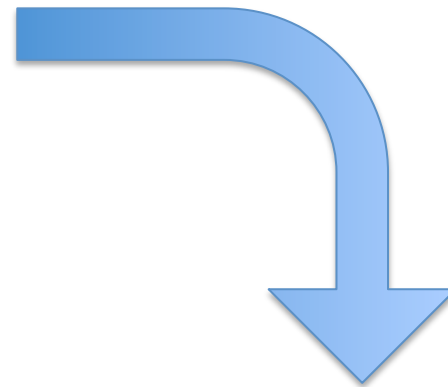
```
*P=x;
```

```
x=*P;
```

Cosa succede?

- Accedere a una variabile mediante puntatore, altro non è che accedere al contenuto di una cella in modo indiretto attraverso il suo indirizzo!

Identificatore	Indirizzo	Contenuto
P	1034	2435
...		
...		
x	2132	
...		
???	2435	10000



$x = *P;$	LOAD@ 1034; STORE 2132
$*P = x;$	LOAD 2132; STORE@ 1034

Costrutto &

L'operatore unario & significa "indirizzo di" ed è il duale dell'operatore '*' → coincidenza non casuale con scanf

Date le seguenti dichiarazioni di tipo e di variabili:

```
typedef          TipoDato          *TipoPuntatore;
```

```
TipoPuntatore  P, Q;
```

```
TipoDato       y, z;
```

è possibile assegnare al puntatore P e al puntatore Q l'indirizzo delle variabili y e z, rispettivamente, tramite le seguenti istruzioni:

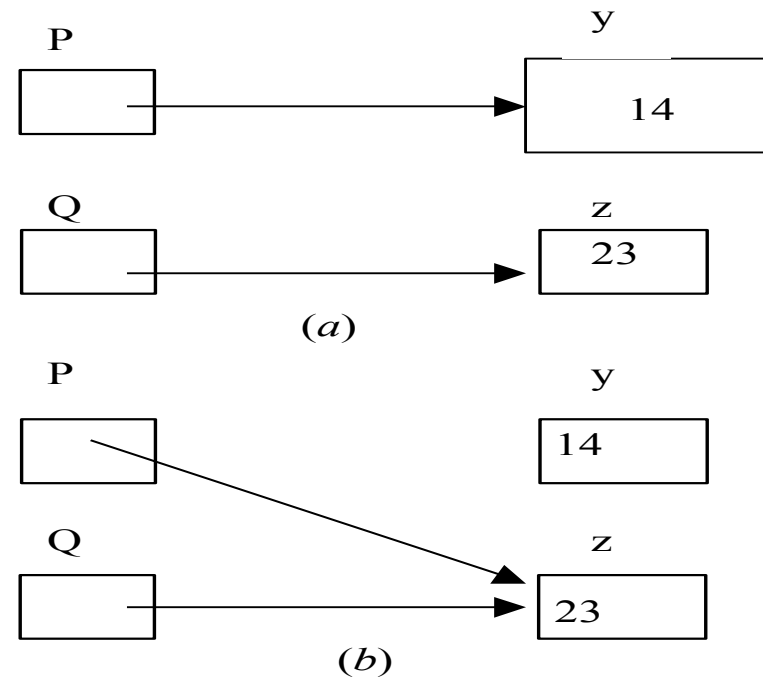
```
P = &y;
```

```
Q = &z;
```

E' possibile eseguire anche il seguente assegnamento:

```
P = Q;
```

Il risultato?



Si noti la differenza tra le istruzioni $P = Q;$ e $*P = *Q; !$

Costrutto *

- Il costrutto denotato dal simbolo * è a tutti gli effetti un costruttore di tipo
 - **typedef** TipoDato *TipoPuntatore;
 - **typedef** AltroTipoDato *AltroTipoPuntatore;
 - TipoDato *Punt;
 - forma abbreviata: unifica una dichiarazione di tipo e una dichiarazione di variabile
 - TipoDato **DoppioPunt;
 - TipoPuntatore P, Q;
 - AltroTipoPuntatore P1, Q1;
 - TipoDato x, y;
 - AltroTipoDato z, w;

Esempi

Istruzioni corrette

- `Punt = &y;`
- `DoppioPunt = &P;`
- `Q1 = &z;`
- `P = &x;`
- `P = Q;`
- `*P = *Q;`
- `*Punt = x;`
- `P = *DoppioPunt;`
- `z = *P1;`
- `Punt = P;`

Istruzioni scorrette

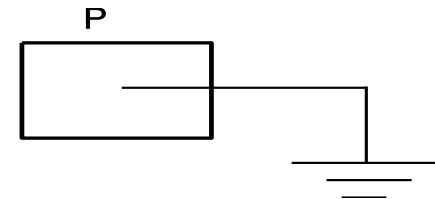
- `P1 = P;` (warning)
- `w = *P;` (error)
- `*DoppioPunt = y;` (error)
- `Punt = DoppioPunt;` (warning)
- `*P1 = *Q;` (error)

Una tipica abbreviazione del C

- **typedef struct** {
 int PrimoCampo;
 char SecondoCampo;
} TipoDato;
TipoDato *P;
- Accesso normale alla componente PrimoCampo della variabile cui P punta
 (*P).PrimoCampo = 12;
- Accesso abbreviato:
 P->PrimoCampo = 12;

Riassumendo e completando

- Le operazioni applicabili a variabili puntatori sono le seguenti:
 - l'assegnamento dell'indirizzo di una variabile tramite l'operatore unario & seguito dal nome della variabile;
 - l'assegnamento del valore di un altro puntatore;
 - l'assegnamento del valore speciale NULL;
 - se una variabile puntatore ha valore NULL, *P è indefinito:
P non punta ad alcuna informazione significativa
 - l'operazione di dereferenziazione, indicata dall'operatore *;
 - il confronto basato sulle relazioni ==, !=, >, <, <=, >=;
 - operazioni aritmetiche (trattato in seguito)
 - l'assegnamento di indirizzi di memoria a seguito di operazioni di allocazione esplicita di memoria (trattato in seguito)
- E' sempre meglio fare esplicitamente l'assegnamento per ogni puntatore (possibilmente con lo speciale valore NULL), benché alcune implementazioni del linguaggio C possano implicitamente assegnare NULL a ciascun nuovo puntatore
 - ciò garantisce che il programma venga eseguito correttamente sotto qualsiasi implementazione del linguaggio



Primo esempio

Il programma che segue tratta informazioni riguardanti i lavoratori di una fabbrica usando dati contenuti in due variabili: la prima, denominata `DatiLavoratori`, contiene informazioni su diversi lavoratori (dirigenti, impiegati, operai); la seconda, chiamata `Management`, permette di accedere rapidamente ai dati relativi ai dirigenti contenuti nella prima struttura dati. Si supponga di voler stampare i cognomi e gli stipendi di tutti i dirigenti che guadagnano più di 5 milioni: anziché selezionare tutti gli elementi dell'intero array `DatiLavoratori` sulla base della categoria del lavoratore, immagazziniamo nell'array `Management` (verosimilmente molto più corto) i puntatori a tutti gli elementi di `DatiLavoratori` che sono dirigenti (ciò, naturalmente, renderà l'aggiornamento dei due array più complicato, per esempio quando viene assunta una nuova persona).

Il frammento di codice di questo esempio presuppone che i dati contenuti nelle variabili `DatiLavoratori` e `Management` siano già disponibili, e restituisce il cognome e lo stipendio dei dirigenti che guadagnano più di 5 milioni.

Frammento di codice...

```
typedef enum      {dirigente, impiegato, operaio} CatLav;
typedef struct    {
    char    Nome[30];
    char    Cognome[30];
    CatLav  Categoria;
    int     Stipendio;
    char    CodiceFiscale[16];
} Lavoratore;
Lavoratore    DatiLavoratori[300];
Lavoratore    *Management[10];
...
i = 0;
while (i < 10) {
    if (Management[i]—>Stipendio > 5000000) {
        j = 0;
        while (j < 30) {
            printf("%c", Management[i]—>Cognome[j]); j = j + 1;
        }
        printf("%d \n", Management[i]—>Stipendio);
    }
    i = i + 1;
}
```

Un array di dieci
puntatori a variabili di
tipo Lavoratore...

Attenzione ai rischi!

- Effetti collaterali:

- *P = 3;

- *Q = 5;

- P = Q;

- *Q = 7;

A questo punto *P=5!

A questo punto *P=7, anche se
non abbiamo toccato P!
Abbiamo creato un **alias** per
quella cella di memoria...

.... ancora sui rischi

- Effetto collaterale perché un assegnamento esplicito alla variabile puntata da Q determina un assegnamento nascosto (quindi con il rischio di non essere voluto) alla variabile puntata da P
 - In realtà questo è un caso particolare di **aliasing**, ovvero del fatto che uno stesso oggetto viene identificato in due modi diversi
- Ulteriori rischi, verranno alla luce in seguito (i puntatori sono una “palestra di trucchi C”, ricca di usi ... ed abusi)

Esercizio

- Si scriva un programma che, dati due interi inseriti da Standard Input e memorizzati in due variabili v1 e v2, ne scambia il valore utilizzando puntatori ad interi.

Frammento di codice...

```
int    v1,v2,temp;
```

```
int    *P1,*P2;
```

```
...
```

```
scanf(&v1);
```

```
scanf(&v2);
```

```
...
```


```
P1 = &v1;
```

```
P2 = &v2;
```

```
temp = *P2;
```

```
v2=*P1;
```

```
*P2=temp;
```



La variabile temp è necessaria per evitare problemi di aliasing!

Array e puntatori

- L'operatore **sizeof** produce il numero di byte occupati da ciascun elemento di memoria
 - Può essere una singola variabile, un record, un elemento di un array, o un array nel suo complesso
- Supponendo di lavorare su un calcolatore che riserva quattro byte per la memorizzazione di un valore int e ipotizzando di aver dichiarato
 - `int a[5];`
 - `sizeof(a[2])` restituisce il valore 4
 - `sizeof(a)` restituisce il valore 20
- Il nome di una variabile di tipo array viene considerato in C come l'indirizzo della prima parola di memoria che contiene il primo elemento della variabile di tipo array (lo 0-esimo ...)
- Se ne deduce:
 - La variabile `a` “punta” a una parola di memoria esattamente come fa una variabile dichiarata di tipo puntatore
 - La variabile `a` punta sempre al primo elemento della variabile di tipo array (è un puntatore “fisso” al quale non è possibile assegnare l'indirizzo di un'altra cella)

Somme e sottrazioni

- C consente di eseguire operazioni di somma e sottrazione su puntatori
- Se una variabile p punta a un particolare tipo di dato, l'espressione $p+1$ fornisce l'indirizzo in memoria per l'accesso o per la corretta memorizzazione della prossima variabile di quel tipo
- Se p e a forniscono l'indirizzo di memoria di elementi di tipo opportuno, $p+i$ e $a+i$ forniscono l'indirizzo di memoria dell' i -esimo elemento successivo di quel tipo
- Riprendendo in considerazione l'array a dichiarato in precedenza:
 - se i è una variabile intera, le notazioni $a[i]$ e $*(a+i)$ sono equivalenti!
- Se p è dichiarato come puntatore a una variabile di tipo `int`, ne segue che:
 - $p = a$ è equivalente a $p = \&a[0];$
 - $p = a+1$ è equivalente a $p = \&a[1];$
- Mentre non sono ammessi i seguenti assegnamenti, perché non è possibile cambiare l'indirizzo di partenza di un array!
 - $a = p;$
 - $a = a + 1;$

Infine

- Se p e q puntano a due diversi elementi di un array
 - $p-q$ restituisce un valore intero pari al numero di elementi esistenti tra l'elemento cui punta p e l'elemento cui punta q
 - non la differenza tra il valore dei puntatori!
- Supponendo infatti che il risultato di $(p-q)$ sia pari a 3 e supponendo che ogni elemento dell'array risulti memorizzato in 4 byte, la differenza tra l'indirizzo contenuto in p e l'indirizzo contenuto in q darebbe 12
- Abbiamo cominciato ad “assaggiare” alcune di quelle peculiarità del C tanto amate dagli “specialisti” e tanto pericolose
- Non lasciarsi attrarre dai vari trucchetti e abbreviazioni (che costano gravi errori anche agli ... esperti)