

# FUNZIONI...

---

Una funzione permette di

- *dare un nome* a una espressione
- rendendola *parametrica*

```
float f() {  
    return 2 + 3 * sin(0.75);  
}
```

```
float f1(int x) {  
    return 2 + x * sin(0.75);  
}
```

# E PROCEDURE

---

Una procedura permette di

- dare un nome a una istruzione
- rendendola *parametrica*
- non denota un valore, quindi non c'è tipo di ritorno → **void**

```
void p(int x) {  
    float y = x * sin(0.75);  
    printf("%f", y);  
}
```

# PROCEDURE COME COMPONENTI SOFTWARE

---

Una *procedura* è un *componente software* che cattura l'idea di “macro-istruzione”

- molti possibili parametri, che *possono anche essere modificati* mentre nelle funzioni non devono essere modificati
- nessun “valore di uscita” esplicito

# PROCEDURE COME SERVITORI

---

- Come una funzione, una procedura è un servitore
  - *passivo*
  - che serve *un cliente per volta*
  - che può trasformarsi in cliente *invocando se stessa o altre procedure*
- In C, una procedura ha la stessa struttura di una funzione, salvo il *tipo di ritorno* che è **void**

# RITORNO DA UNA PROCEDURA

---

- L'istruzione *return* provoca solo la restituzione del controllo al cliente
- non è seguita da una espressione da restituire
- quindi, *non è necessaria* se la procedura termina “spontaneamente” a fine blocco (cioè al raggiungimento della parentesi graffa di chiusura)

# COMUNICAZIONE CLIENTE SERVITORE

---

- Nel caso di una procedura, non esistendo valore di ritorno, cliente e servitore comunicano solo:
  - mediante i parametri
  - mediante *aree dati globali*
- Il passaggio per valore non basta più:
  - occorre il passaggio per riferimento per poter fare cambiamenti permanenti ai dati del cliente.

# PASSAGGIO DEI PARAMETRI

---

In generale, un parametro può essere trasferito dal cliente al servitore:

- **per valore o copia (*by value*)**
  - si trasferisce il valore del parametro attuale
- **per riferimento (*by reference*)**
  - si trasferisce un riferimento al parametro attuale

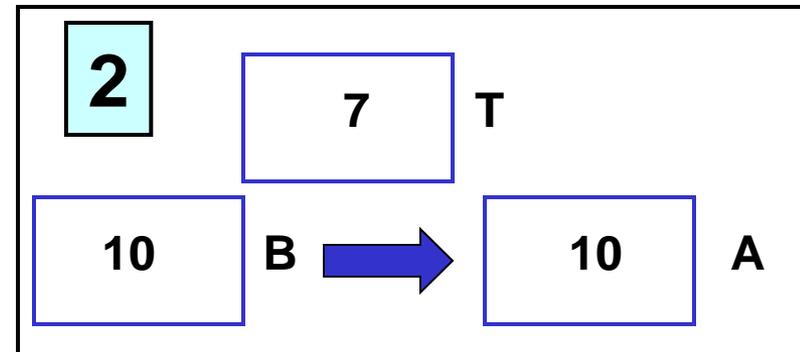
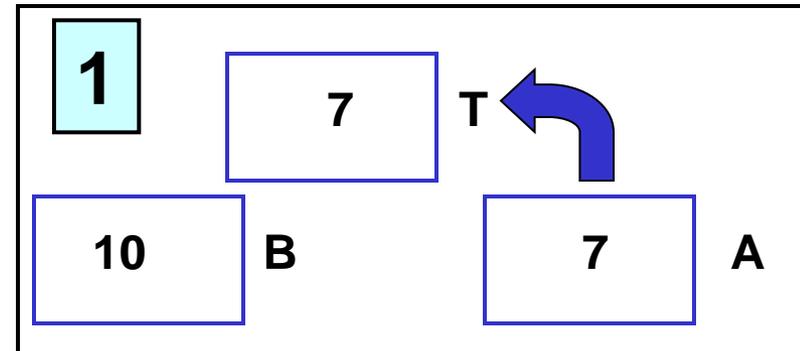
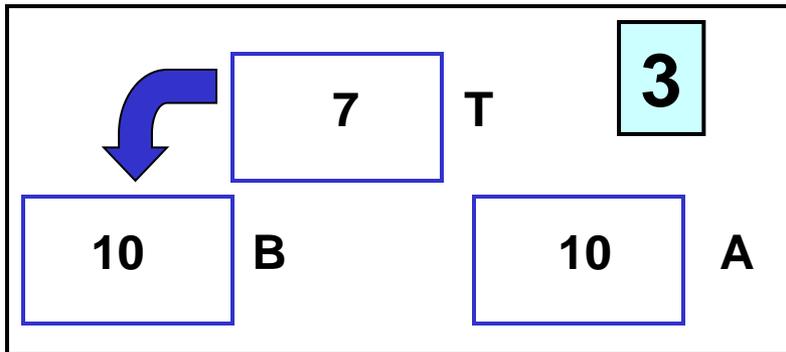
# ESEMPIO

## Perché il passaggio per valore non basta?

Problema: scrivere una procedura che *scambi i valori di due variabili intere*.

Specifica:

Dette A e B le due variabili, ci si può appoggiare a una *variabile ausiliaria T*, e fare una “triangolazione” in *tre fasi*.



# ESEMPIO

---

Supponendo di utilizzare, senza preoccuparsi, il passaggio per valore usato finora, la codifica potrebbe essere espressa come segue:

```
void scambia(int a, int b) {  
    int t;  
    t = a;    a = b;    b = t;  
    return; /* può essere omessa */  
}
```

# ESEMPIO

---

Il cliente invocherebbe quindi la procedura così:

```
int main() {
    int y = 5, x = 33;
    scambia(x, y);
    /* ora dovrebbe essere
       x=5, y=33 ...
       MA NON E' VERO !!
    */
    return 0; }
```

***Perché non funziona??***

# ESEMPIO

---

- La procedura ha *effettivamente scambiato* i valori di A e B al suo interno
- ma questa modifica non si è propagata al cliente, perché sono state scambiate *le copie locali alla procedura, non gli originali!*
- al termine della procedura, le sue variabili locali sono state distrutte → nulla è rimasto del lavoro fatto dalla procedura!!

X

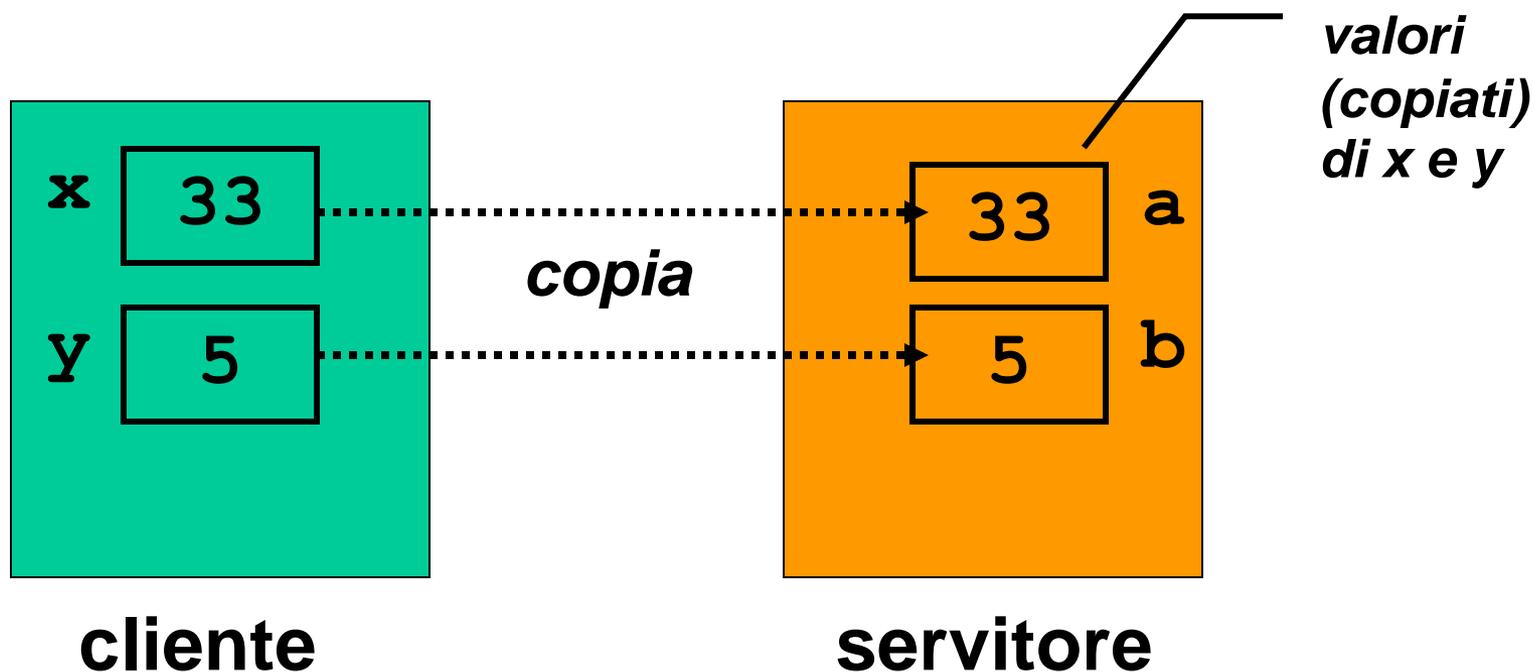
Y

~~A~~

~~B~~

# PASSAGGIO PER VALORE

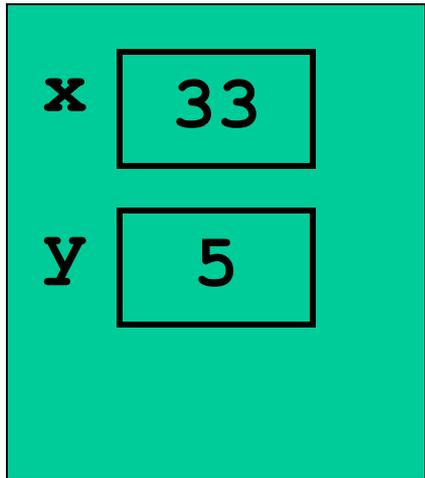
Ogni azione fatta su  $a$  e  $b$  è strettamente locale al servitore. Quindi  $a$  e  $b$  vengono scambiati ma quando il servitore termina, tutto scompare.



# PASSAGGIO PER VALORE

---

**... e nel cliente non è cambiato niente!!!**



**cliente**

# PASSAGGIO DEI PARAMETRI IN C

---

## Il C adotta **sempre il passaggio per valore**

- è sicuro: le variabili del cliente e del servitore sono *disaccoppiate*
- ma *non consente di scrivere componenti software il cui scopo sia diverso dal calcolo di una espressione*
- per superare questo limite occorre il passaggio per riferimento (*by reference*)

# PASSAGGIO PER RIFERIMENTO

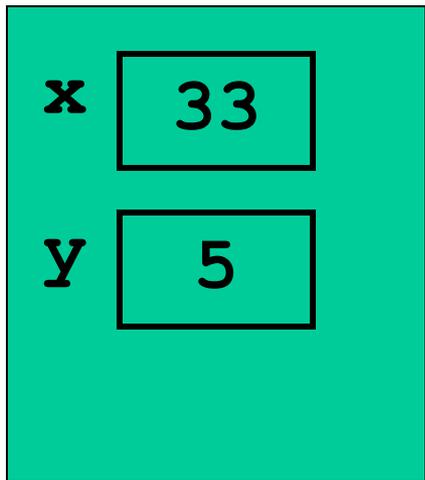
---

- Il passaggio per riferimento (*by reference*)
  - non trasferisce una copia del valore del parametro attuale
  - ma un riferimento al parametro, in modo da dare al servitore accesso diretto al parametro in possesso del cliente
  - il servitore, quindi, *accede direttamente* al dato del cliente *e può modificarlo*.

# PASSAGGIO PER RIFERIMENTO

---

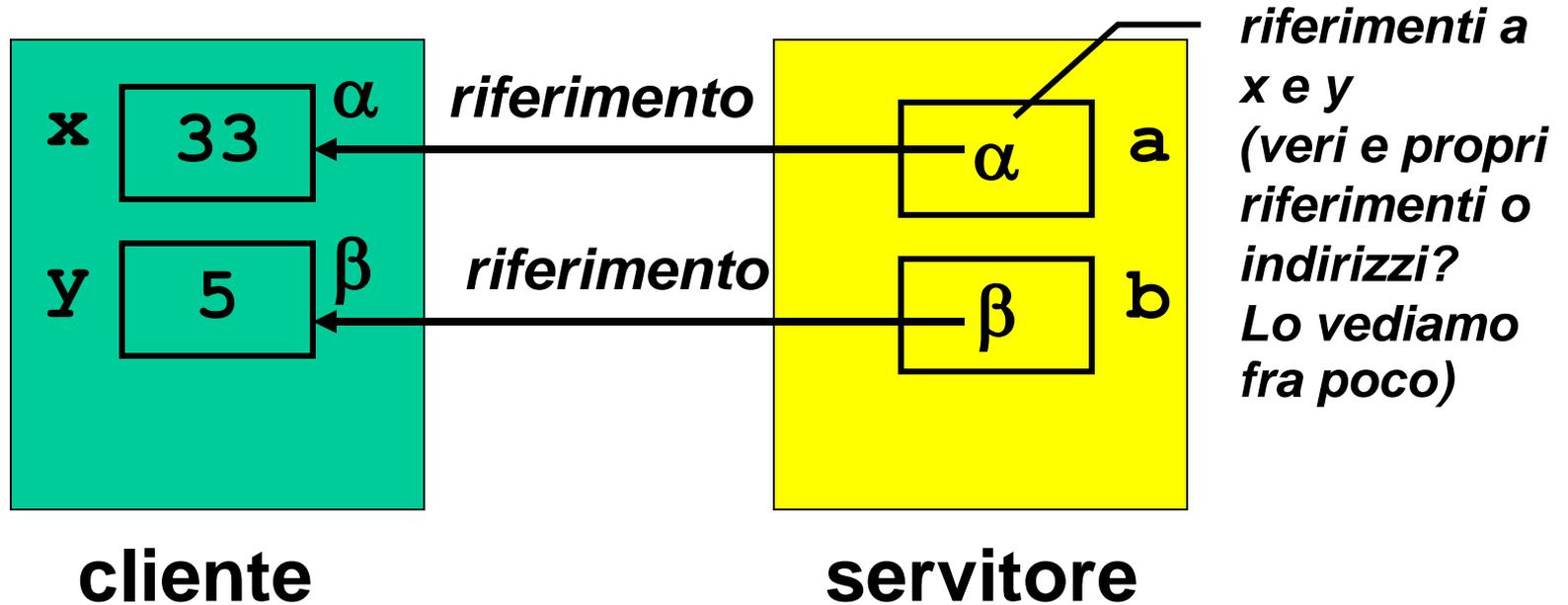
Si trasferisce un riferimento ai parametri attuali



**cliente**

# PASSAGGIO PER RIFERIMENTO

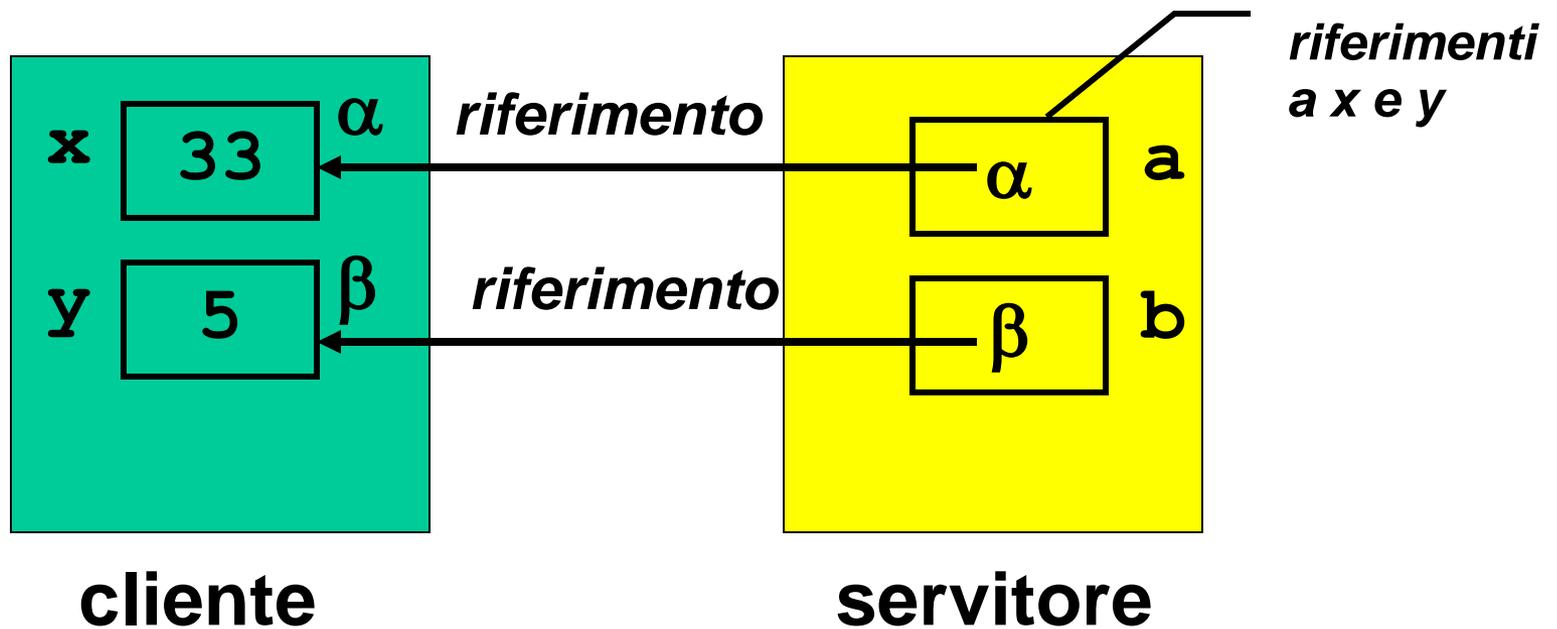
Ogni azione fatta su **a** e **b**  
*in realtà è fatta su **x** e **y***  
nell'environment del cliente



# PASSAGGIO PER RIFERIMENTO

---

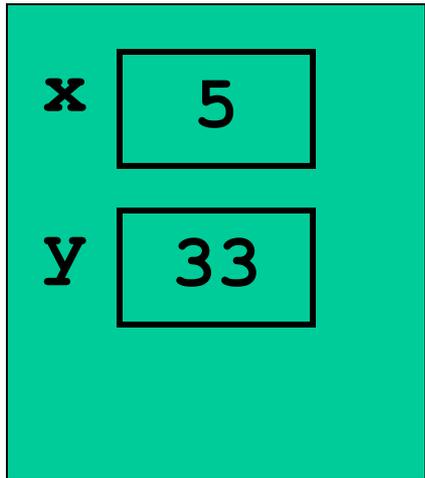
Quindi, scambiando  $a$  e  $b$  in realtà si scambiano  $x$  e  $y$



# PASSAGGIO PER RIFERIMENTO

---

... e alla fine *la modifica permane!*



**cliente**

# REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

---

- Il C *non* fornisce *direttamente* un modo per attivare il passaggio per riferimento, che però è *praticamente indispensabile*
- quindi, dobbiamo *costruircelo*.

## È possibile costruirlo? Come?

- Poiché passare un parametro per riferimento comporta la capacità di manipolare *indirizzi di variabili...*
- ... gestire il passaggio per riferimento implica la capacità di *accedere, direttamente o indirettamente, agli indirizzi delle variabili.*

# REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

---

In particolare occorre essere capaci di:

- *ricavare l'indirizzo* di una variabile
- *dereferenziare un indirizzo* di variabile, ossia “recuperare” la variabile dato il suo indirizzo.
- Nei linguaggi che offrono direttamente il passaggio per riferimento, *questi passi sono effettuati* in modo trasparente all'utente → l'utente non manipola alcun dettaglio della macchina fisica sottostante
- In C l'utente deve conoscere gli indirizzi delle variabili e quindi accedere alla macchina sottostante.

# INDIRIZZAMENTO E DEREFERENCING

---

Il C offre a tale scopo *due operatori*, che consentono di:

- ricavare l'indirizzo di una variabile

*operatore estrazione di indirizzo*      **&**

- dereferenziare un indirizzo di variabile

*operatore di dereferenziamento*      **\***

# INDIRIZZAMENTO E DEREFERENCING

---

- Se  $x$  è il nome di una variabile,  
 $\&x$  denota l'*indirizzo in memoria* di tale variabile:

$$\&x \equiv \alpha$$

- Se  $\alpha$  è l'indirizzo di una variabile,  
 $*\alpha$  denota *tale variabile*:

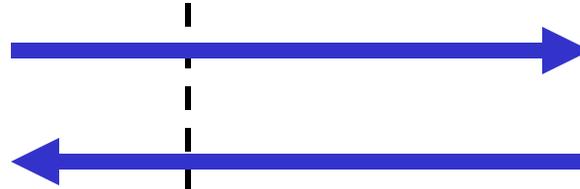
$$x \equiv *\alpha$$

# INDIRIZZAMENTO E DEREFERENCING

---

$$\&x \equiv \alpha$$

$$x \equiv * \alpha$$



*Livello di  
astrazione del  
linguaggio di alto  
livello*

**x**



**$\alpha$**

*Livello di  
astrazione della  
macchina fisica  
sottostante*

# PUNTATORI

---

- Un *puntatore* è il costrutto linguistico introdotto dal C (e da molti altri linguaggi) come *forma di accesso alla macchina sottostante* e in particolare agli indirizzi di variabili
- Un *tipo puntatore a T* è un tipo che denota l'indirizzo di memoria di una variabile di tipo T.
- Un *puntatore a T* è una variabile di “*tipo puntatore a T*” che può contenere l'indirizzo di una variabile di tipo T.

# PUNTATORI

---

- Definizione di una variabile puntatore:

`<tipo> * <nomevariabile> ;`

- Esempi:

```
int *p;
```

```
int* p;
```

```
int * p;
```

*Queste tre forme sono equivalenti, e definiscono p come "puntatore a intero"*

# REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

---

- In C per realizzare il passaggio per riferimento:
  - il cliente deve passare esplicitamente gli indirizzi
  - il servitore deve prevedere esplicitamente dei puntatori come parametri formali

# REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

---

```
void scambia(int* a, int* b) {  
    int t;  
    t = *a;  *a = *b;  *b = t;  
}
```

```
int main() {  
    int y = 5, x = 33;  
    scambia(&x, &y);  
    return 0;}
```

# OSSERVAZIONE

---

- Quando un puntatore è usato per realizzare il passaggio per riferimento, la funzione non dovrebbe mai alterare il valore del puntatore.
- Quindi, se **a** e **b** sono due puntatori:

**\*a = \*b**      **SI**

~~**a = b**~~      **NO**

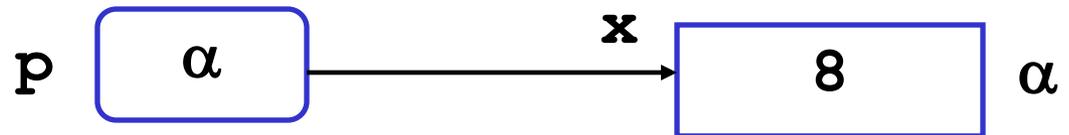
- In generale una funzione può modificare un puntatore, ma *non è opportuno che lo faccia se esso realizza un passaggio per riferimento*

# PUNTATORI

- Un *puntatore* è una variabile *destinata a contenere l'indirizzo di un'altra variabile*
- Vincolo di tipo: un puntatore a T può contenere solo l'indirizzo di variabili di tipo T.

- Esempio:

```
int x = 8;  
int* p;  
p = &x;
```



Da questo momento, **\*p** e **x** sono *due modi alternativi per denotare la stessa variabile*

# COMUNICAZIONE TRAMITE L'ENVIRONMENT GLOBALE

---

- Una procedura può anche comunicare con il suo cliente mediante *aree dati globali*: un esempio sono le *variabili globali del C*.
- Le *variabili globali* in C:
  - sono allocate nell'area dati globale (fuori da ogni funzione)
  - esistono *già prima* della chiamata del *main*
  - sono *inizializzate automaticamente a 0* salvo diversa indicazione
  - possono essere *nascoste* in una funzione da una variabile locale omonima
  - sono visibili, previa dichiarazione *extern*, in tutti i file dell'applicazione

# ESEMPIO

---

**Esempio:** Divisione intera  $x/y$  con calcolo di quoziente e resto. Occorre calcolare *due* valori che supponiamo di mettere in due variabili globali.

```
int quoziente, int resto;
```

variabili globali **quoziente** e **resto** visibili in tutti i blocchi

```
void dividi(int x, int y) {  
    resto = x % y; quoziente = x/y;  
}
```

```
int main() {  
    dividi(33, 6);  
    printf("%d%d", quoziente, resto);  
    return 0;}  
}
```

Il risultato è disponibile per il cliente nelle variabili globali **quoziente** e **resto**

# ESEMPIO

---

**Esempio:** Con il passaggio dei parametri per indirizzo avremmo il seguente codice

```
void dividi(int x, int y, int* quoziente, int* resto)
{
    *resto = x % y; *quoziente = x/y;
}
```

```
int main() {
    int k = 33, h = 6, quoz, rest;
    int *pq = &quoz, *pr = &rest;
    dividi(k, h, pq, pr);
    printf("%d%d", quoz, rest);
    return 0;}
```