

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) {  
    x = x * 2;  
    printf("%d", x);  
}
```

PROCEDURE COME SERVITORI

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

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

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**

PROCEDURE

L'istruzione *return* provoca solo la restituzione del controllo al cliente e non è seguita da una espressione da restituire -> *non è necessaria* se la procedura termina "spontaneamente" a fine blocco

Nel caso di una procedura, non esistendo valore di ritorno, cliente e servitore comunicano solo:

- mediante ***parametri***
- mediante ***aree dati globali***

➡ Occorre il ***passaggio per riferimento*** per 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

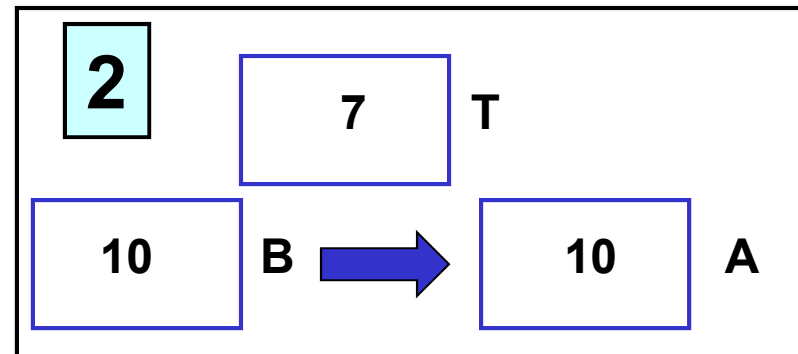
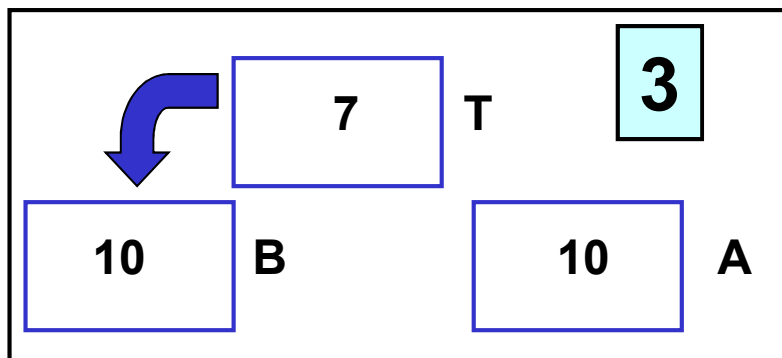
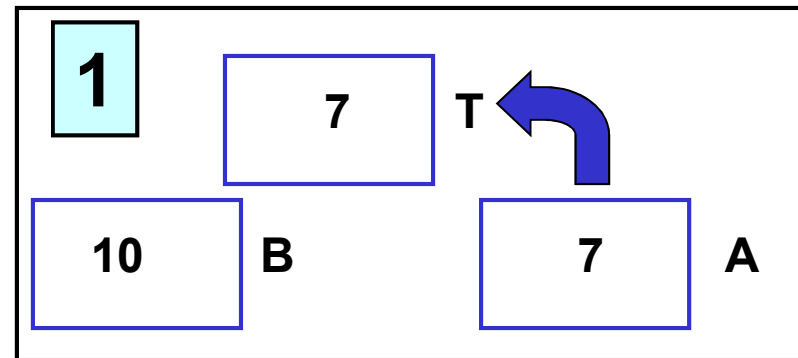
ESEMPIO

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 svolgere lo scambio 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 È VERO  
    */  
}
```

Perché non funziona?

ESEMPIO

- La procedura ha *effettivamente scambiato* i valori di A e B al suo interno (in C nel suo record di attivazione)
- 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 svolto dalla procedura

X

33

Y

5

~~A

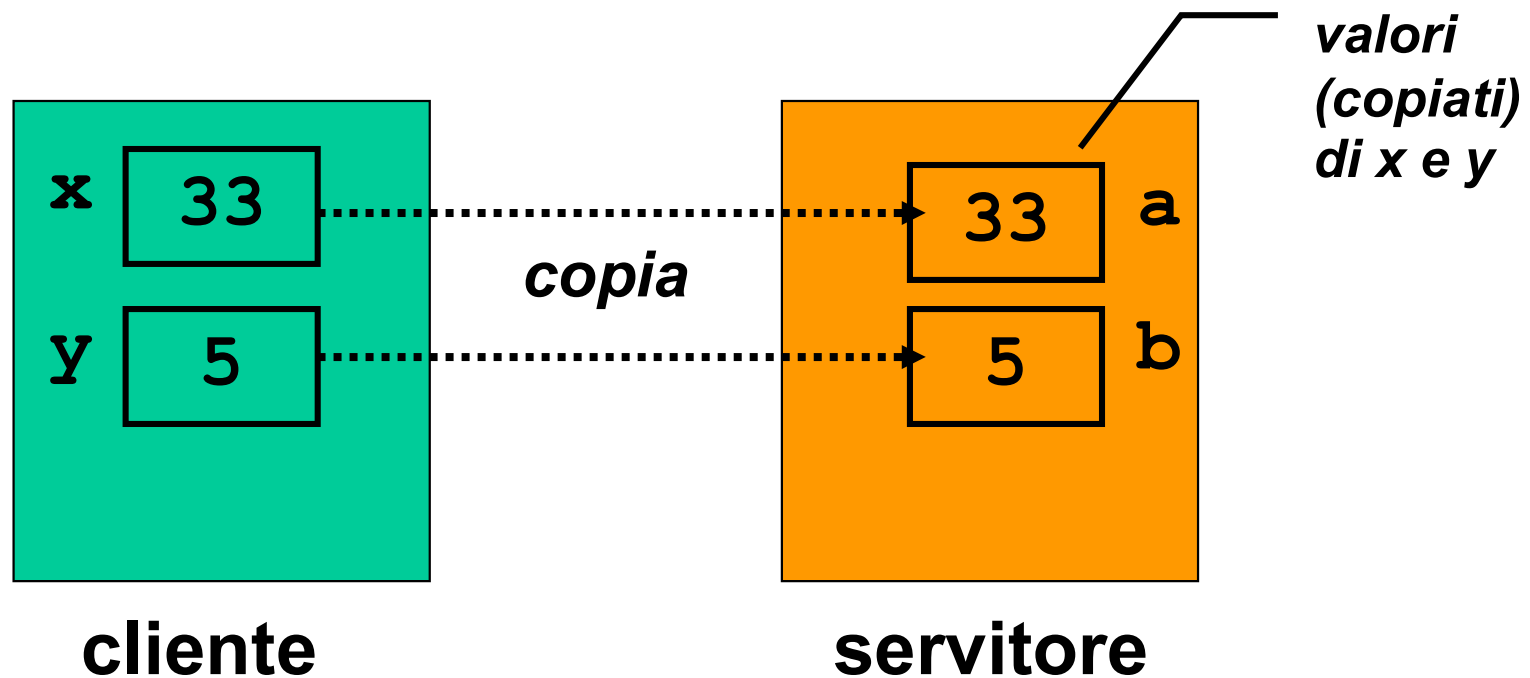
5

B

33

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 DEI PARAMETRI IN C

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

- 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 DEI PARAMETRI IN C

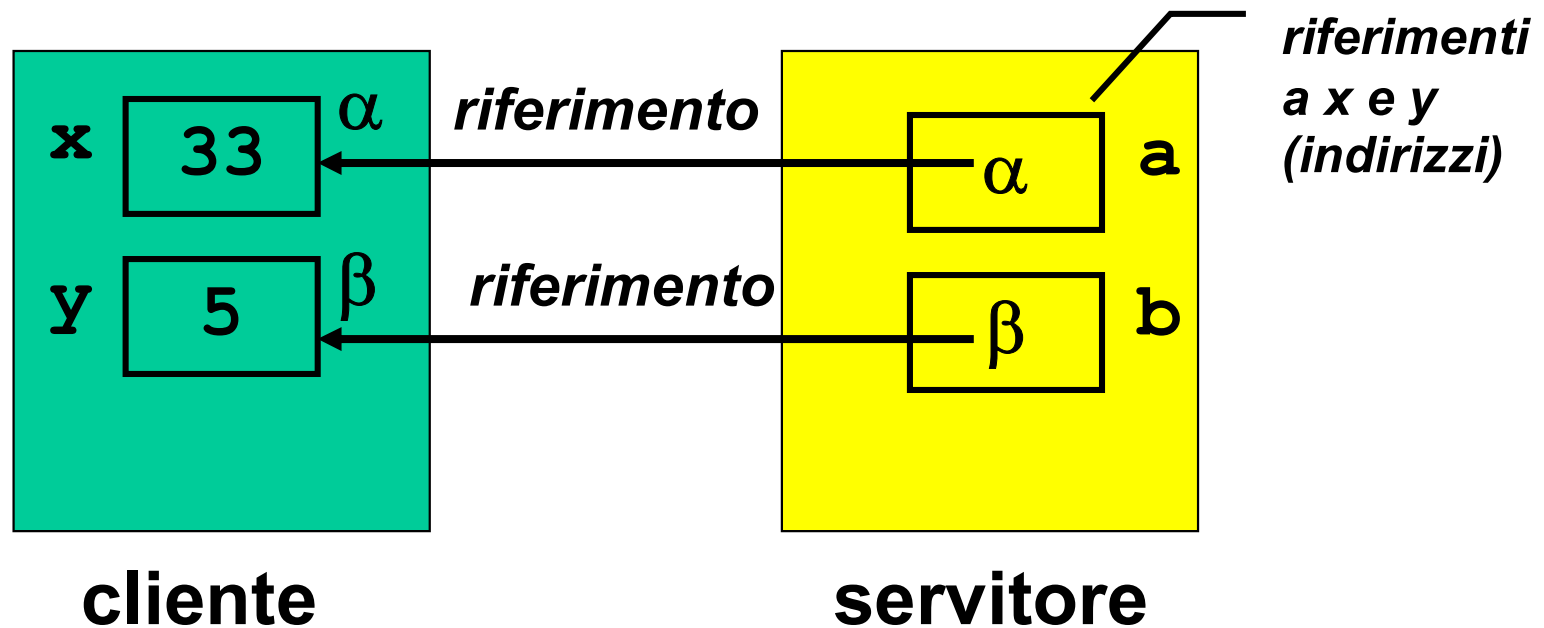
Il linguaggio C **NON** supporta *direttamente* il **passaggio per riferimento**

- è una grave mancanza
- viene fornito indirettamente solo per alcuni tipi di dato
- occorre quindi **costruirlo quando serve**

PASSAGGIO PER RIFERIMENTO

Si trasferisce un riferimento ai parametri attuali (cioè i loro indirizzi)

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



REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

Il C *non* fornisce *direttamente* un modo per attivare il passaggio per riferimento -> a volte occorre *costruirselo*

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” il valore dato l'indirizzo della variabile

Nei linguaggi che offrono direttamente il passaggio per riferimento, *questi passi sono effettuati* in modo trasparente all'utente

In C il ***programmatore 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, denotando la variabile (e il valore contenuto in quell'indirizzo)
operatore di dereferenziamento *****

INDIRIZZAMENTO E DEREFERENCING

Se x è una variabile,

$\&x$ denota l'*indirizzo in memoria* di tale variabile:

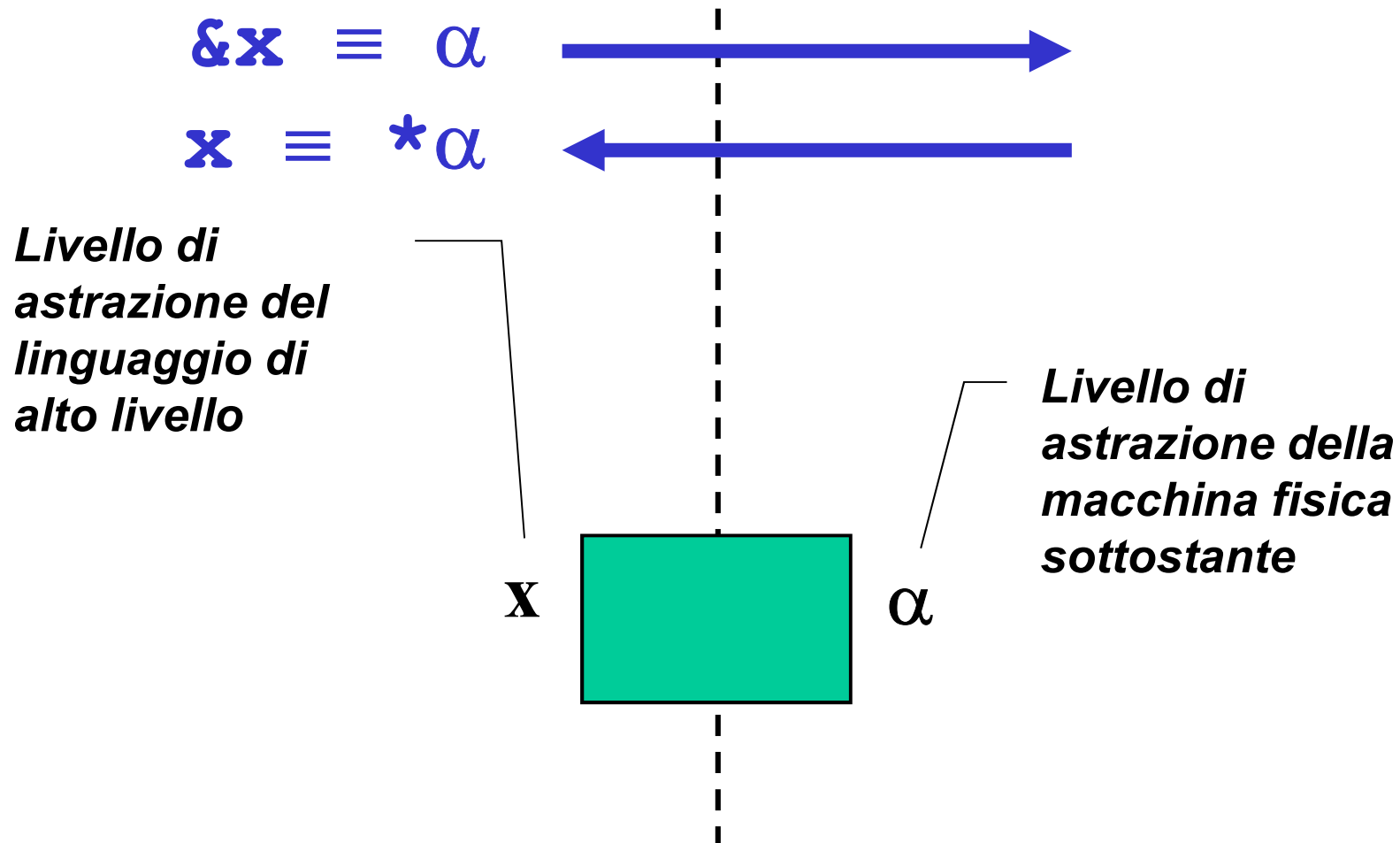
$$\&x \equiv \alpha$$

Se α è l'indirizzo di una variabile,

$*\alpha$ denota *tale variabile*:

$$x \equiv *\alpha$$

INDIRIZZAMENTO E DEREFERENCING



PUNTATORI

Un *puntatore* è il costrutto linguistico introdotto dal C (e da 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

Una variabile puntatore è una variabile che permette di memorizzare un indirizzo di memoria (o la costante **NULL**):

```
<tipo> * <nomevariabile> ;
```

Il tipo del puntatore indica il tipo del dato contenuto nell'indirizzo di memoria puntato.

Esempi:

```
int *p;
```

```
int* p;
```

```
int * p;
```

Queste tre forme sono equivalenti e definiscono p come “puntatore a intero” ovviamente diverso significato da **int p;**

E' anche possibile specificare puntatori generici per cui non si specifica il tipo: **void * p;** necessario poi un **cast**

PASSAGGIO PER RIFERIMENTO IN C

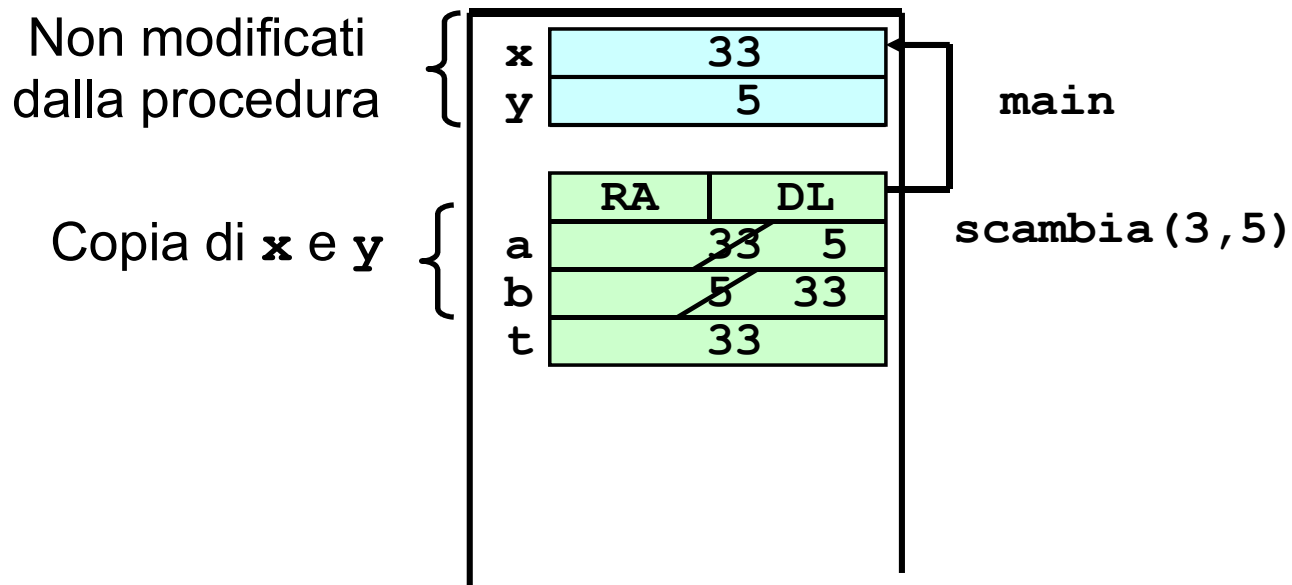
- il cliente deve passare esplicitamente gli indirizzi
- il servitore deve prevedere esplicitamente dei puntatori come parametri formali

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

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

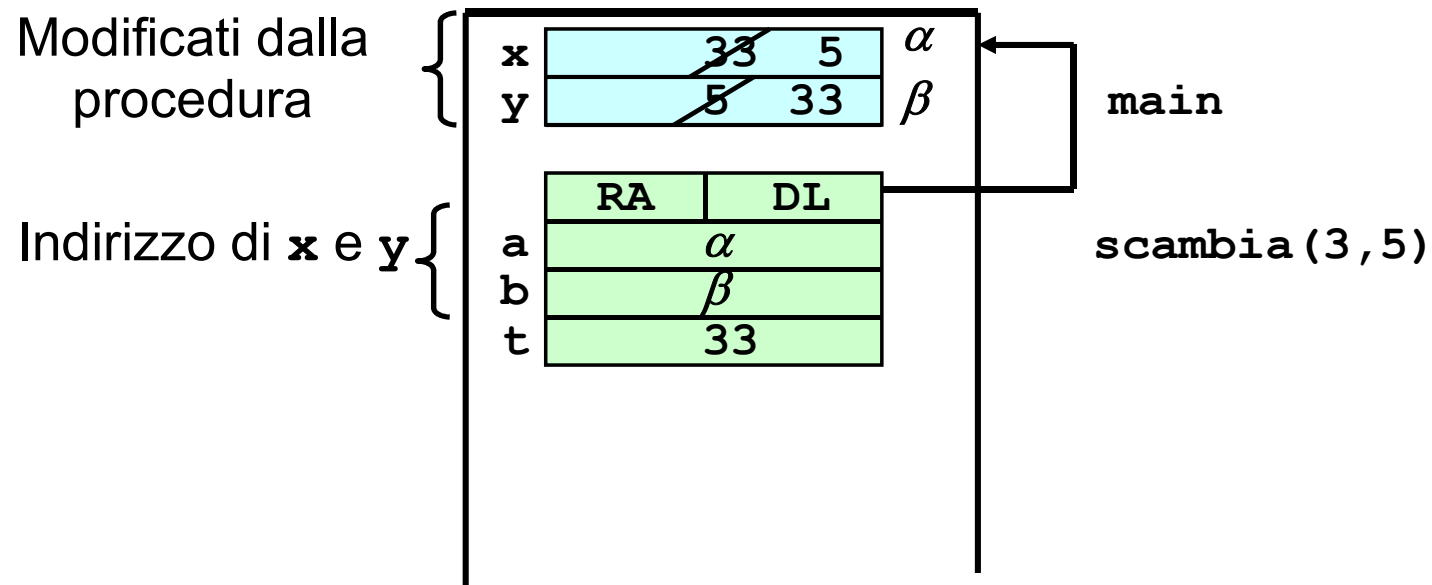
ESEMPIO: RECORD DI ATTIVAZIONE

Caso del *passaggio per valore*:



ESEMPIO: RECORD DI ATTIVAZIONE

Caso del *passaggio per riferimento*:



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**

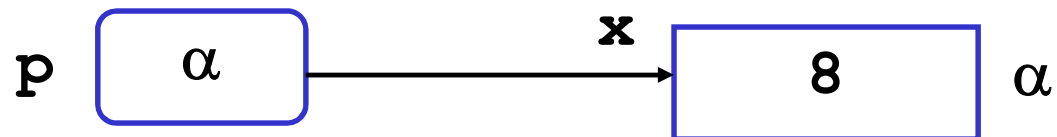
~~**a = b**~~ **NO**

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*

La costante NULL

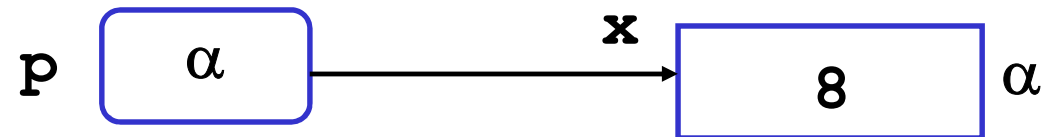
- E' una macro definita in `stdlib.h`
- Rappresenta un indirizzo di memoria non accessibile, il puntatore con valore `NULL` non punta a nulla.
- Equivalenza fra la costante `0` e `NULL`

PUNTATORI

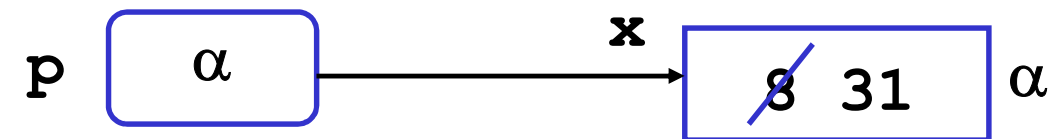
```
int x = 8;
```



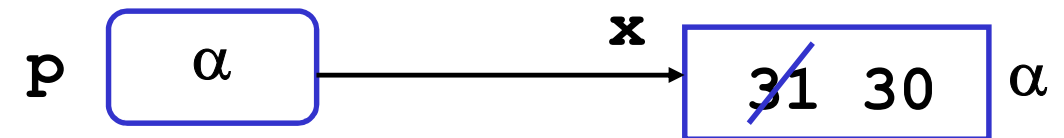
```
int* p = &x;
```



```
*p = 31;
```



```
x--;
```



Note sui puntatori

- Dereferenziare puntatori (operatore *) non inizializzati o inizializzati in modo scorretto o con valore NULL per cui non puntano ad una cella di memoria effettiva, causa **comportamento non definito**.

- `int * p; *p=1; scorretto!`

- `int * p; p=-1; scorretto!`

- `int * p =NULL; *p=1; scorretto!`

- E' possibile definire puntatori di puntatori:
 - `int X=1; int *p=&X; int **q=&p;`
 - `printf("%d%d", *p, **q);` stampa due volte il contenuto della variabile X

PUNTATORI

Un puntatore non è legato per sempre alla stessa variabile; può essere modificato

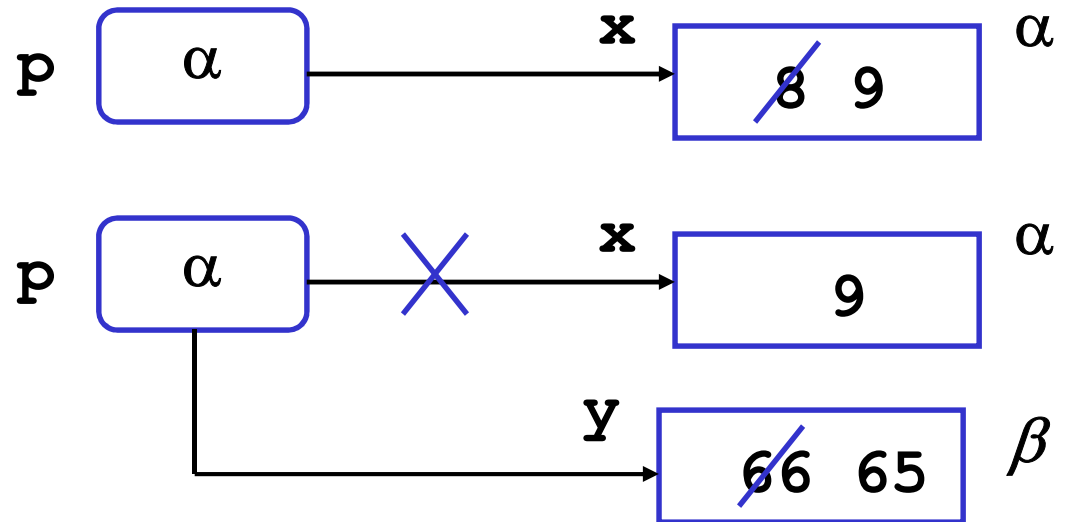
```
int x = 8, y = 66;
```

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

```
(*p) ++;
```

```
p = &y;
```

```
(*p) --;
```



Le parentesi sono necessarie? La scrittura $*p--$ equivale a $*(p--)$, si estrae il valore puntato da p e poi si decrementa il puntatore p .

PUNTATORI

Un puntatore a T può contenere solo *l'indirizzo di variabili di tipo T*: ***puntatori a tipi diversi sono incompatibili tra loro***

Esempio:

```
int x=8, *p;    float *q;  
p = &x;        /* OK */  
q = p;        /* NO! */
```

MOTIVO: il tipo *del puntatore* serve per dedurre il tipo dell'oggetto puntato, che è una ***informazione indispensabile per effettuare il dereferencing***

PUNTATORI

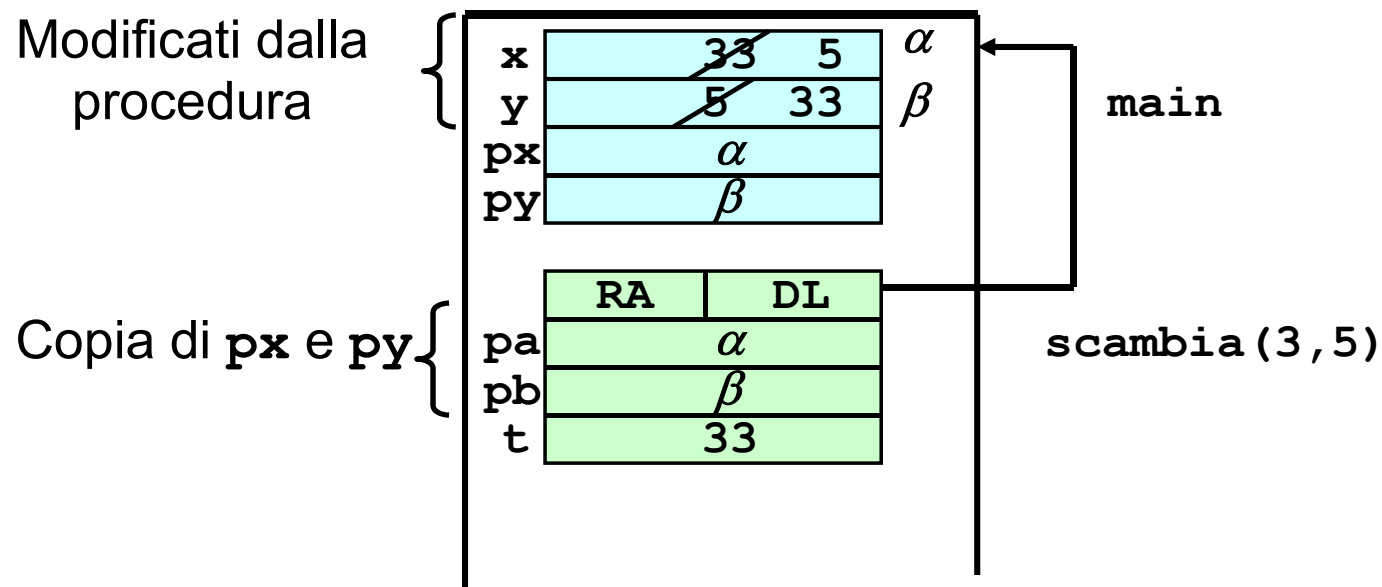
```
void scambia(int* pa, int* pb) {  
    int t;  
    t = *pa;  *pa = *pb;  *pb = t;  
}
```

```
int main() {  
    int y = 5, x = 33;  
    int *py = &y, *px = &x;  
    scambia(px, py);  
}
```

Variazione dall'esempio precedente: i puntatori sono memorizzati in `px` e `py` prima di passarli alla procedura

ESEMPIO: RECORD DI ATTIVAZIONE

Il record di attivazione si modifica come segue



COMUNICAZIONE TRAMITE AMBIENTE GLOBALE

Una procedura può anche comunicare con il cliente *mediante aree dati (variabili) globali*

Le *variabili globali* in C:

- sono allocate ***nell'area dati globale*** (fuori da ogni funzione)
- esistono ***prima*** della chiamata del ***main*** (***quindi prima dell'esecuzione del programma***).
- ***Il tempo di vita*** si estende per tutta ***la durata del programma***
- sono visibili, previa dichiarazione ***extern***, in tutti i file dell'applicazione
- sono ***inizializzate automaticamente a 0*** salvo diversa indicazione
- possono essere ***nascoste*** in una funzione da una variabile locale omonima

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);  
}
```

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

SOLUZIONE ALTERNATIVA

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(33, 6, pq, pr);  
    printf("%d%d", quoz, rest);  
}
```