Sottoprogrammi

- I sottoprogrammi possono servire per creare
  - nuovi operatori: funzioni, che possono essere usati nelle espressioni, forniscono un valore di ritorno

```
x=4*potenza(2+y,3);
```

  - nuove istruzioni: procedure, che non hanno un risultato

```
stampa(f);
```

- Il linguaggio C è basato sul concetto di espressione, quindi le funzioni fanno la parte del leone.
- Le procedure sono realizzate come caso particolare di funzioni: funzioni che non hanno un valore di ritorno. Per indicare il tipo del risultato si usa la parola chiave **`void`**.

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**

- Es: visualizzazione di una frazione

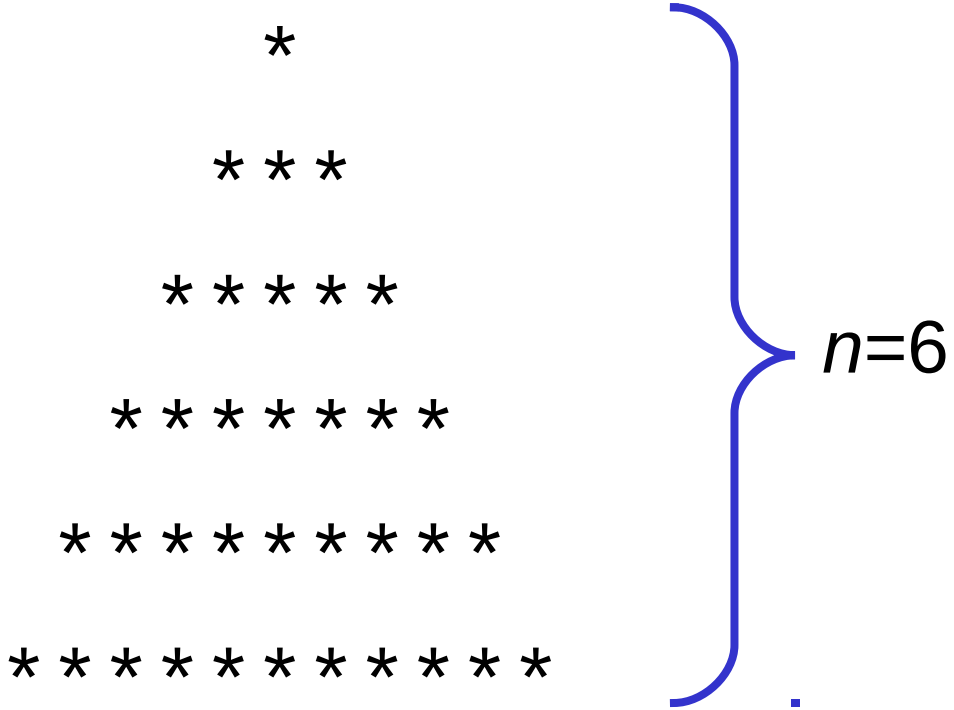
```
typedef struct { int num; int den; } frazione;

void stampaFrazione(frazione f)
{ printf("%d/%d",f.num,f.den);
}

main()
{ frazione f ={3,5};
  stampaFrazione(f);
}
```

`return` IN 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)

ESEMPIO

- Disegnare, per un dato *n*, la seguente figura



ESEMPIO

- Visualizzazione di *n* caratteri tutti uguali

```
void printN(char c, int n)
{  int i;
   for (i=0; i<n; i++)
      printf("%c",c);
}
```

utile per disegnare istogrammi.

Passaggio dei parametri

- In linguaggio C, i dati sono passati per copia, cioè il valore del parametro attuale viene copiato sul parametro formale
- Perché questa scelta?

Vantaggio 1

```
int potenza(int b, int e)
{ int p=1;
  for (i=0;i<e;i++)
     p = p*b;
  e--;
  return p;
}
main()
{ int x=2,y=3,z;
  z = potenza(x,y);
  printf("%d^%d=%d",x,y,z);
}
```

Vantaggio 2

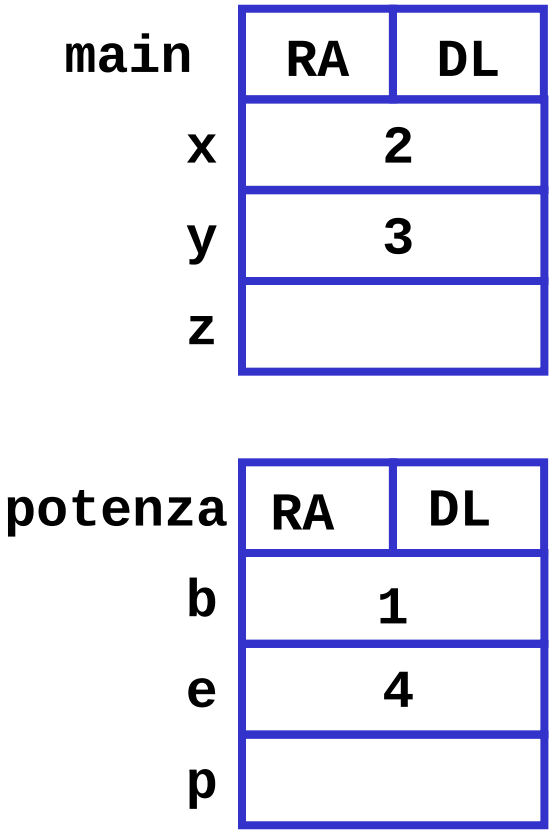
```
int potenza(int b, int e)
{ int p=1;
  while (e>0)
  {  p = p*b;
     e--;
  }
  return p;
}
main()
{ int x=2,y=3,z;
  z = potenza(x,y);
  printf("%d^%d=%d",x,y,z);
}
```

Vantaggio 3

```
main()
{ int x,y,z;
  x=2;
  y=x+1;
  z = potenza(x,y);
  printf("%d^%d=%d",x,y,z);
}
```

Vantaggio 4

```
int potenza(int b, int e)
{ int p=1;
  while (e>0)
  {  p = p*b;
     e--;
  }
  return p;
}
main()
{ int x=2,y=3,z;
  z = potenza(x-1,y+1);
  printf("%d^%d=%d",x,y,z);
}
```



Svantaggi

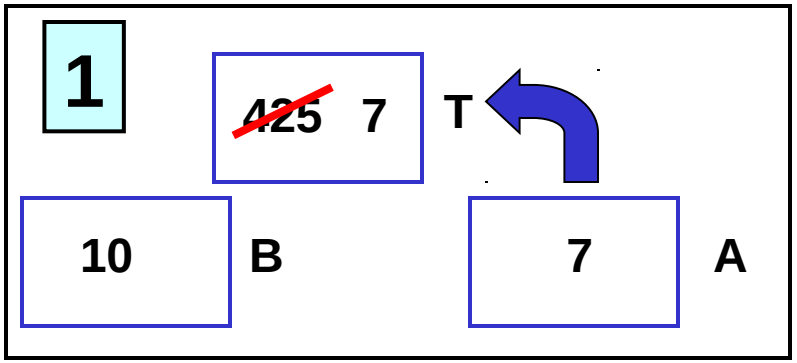
- Però in questo modo non posso modificare il valore dei parametri
- In genere è un vantaggio, ma in certi casi potrebbe servirmi
  - Posso trasferire i dati solo in una direzione (dalla funzione chiamante alla chiamata)
  - L'unico modo per avere una comunicazione dalla funzione invocata a quella invocante è con il valore di ritorno
    - E se mi serve più di un risultato?
  - Se ho una struttura dati molto grande, ricopiarla può essere costoso

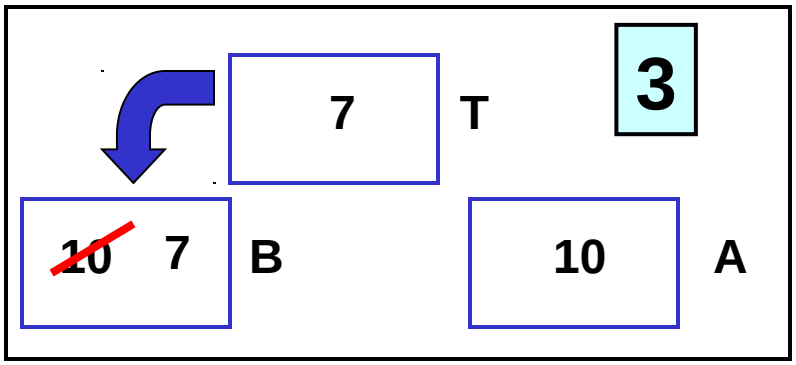
ESEMPIO

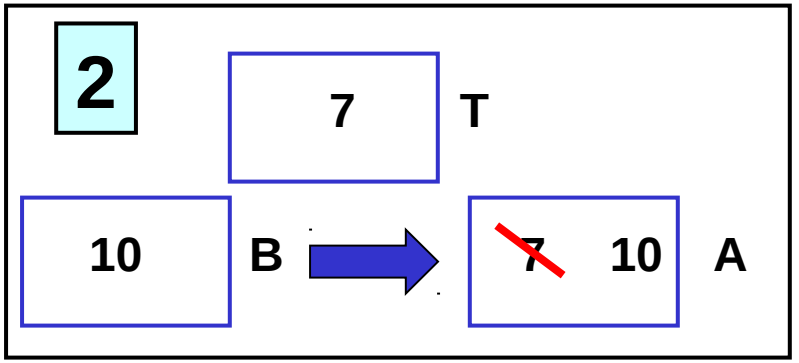
Perché il passaggio per valore non basta?

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;
}
```

ESEMPIO

Il **`main`** invocherebbe quindi la procedura così:

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

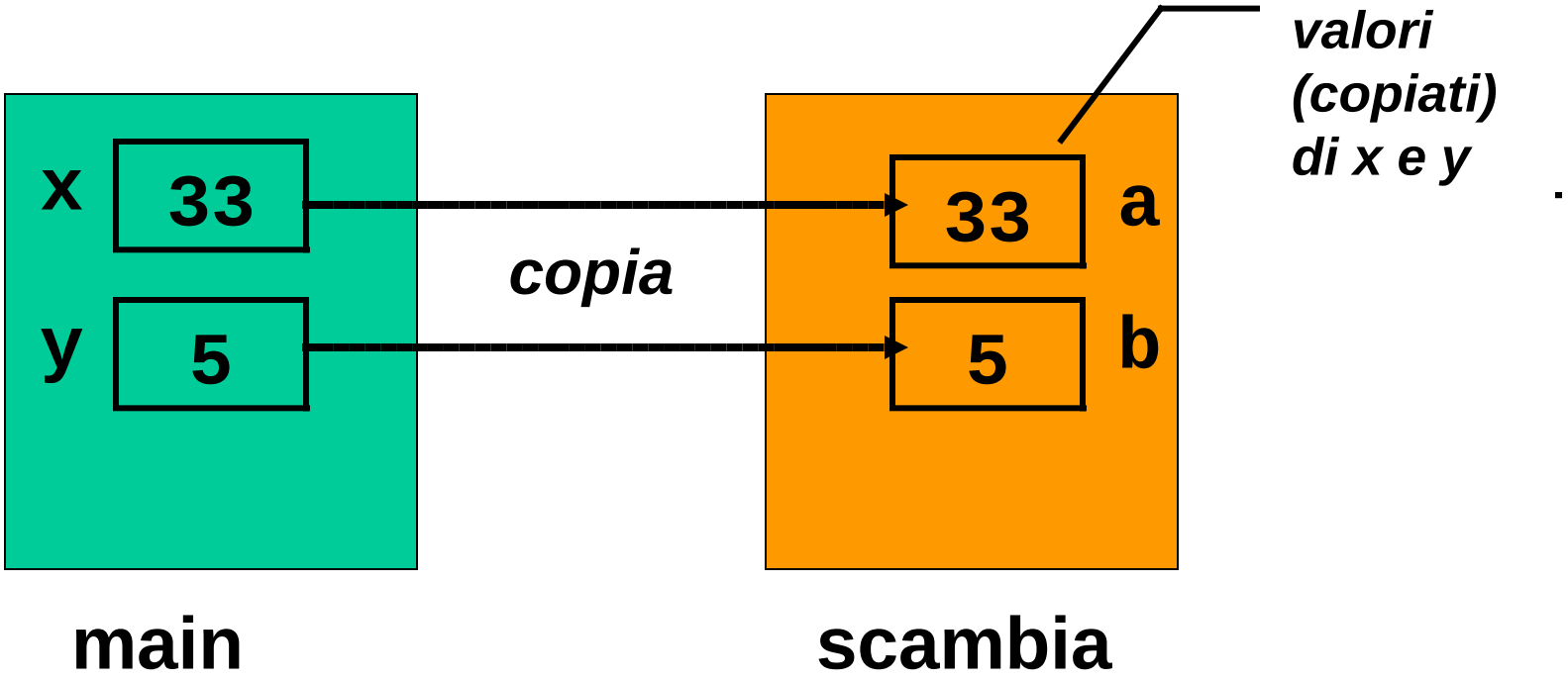
Perché non funziona??

ESEMPIO

- La procedura ha *effettivamente scambiato* i valori di A e B *al suo interno*
- *ma questa modifica non si è propagata al* **`main`**, 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!!

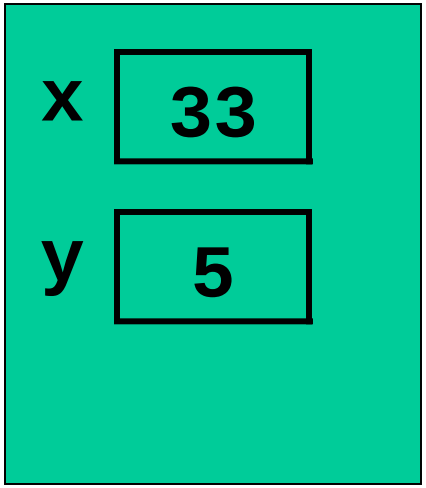
PASSAGGIO PER VALORE

Ogni azione fatta su a e b è <u>strettamente locale</u> alla procedura. Quindi a e b vengono scambiati ma quando la procedura termina, tutto scompare.



PASSAGGIO PER VALORE

... e nel main non è cambiato niente!!!



main

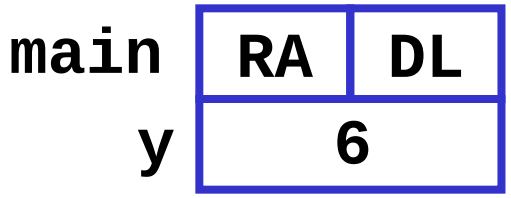
Esempio

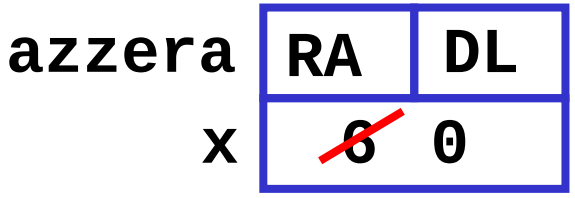
- Scrivere la procedura azzera, che assegna il valore zero ad una variabile

```
void azzera(int x)
{x=0;
}
```

Esempio

```
void azzera(int x)
{ x=0;
}

main()
{ int y=6;
  azzera(y);
}
```

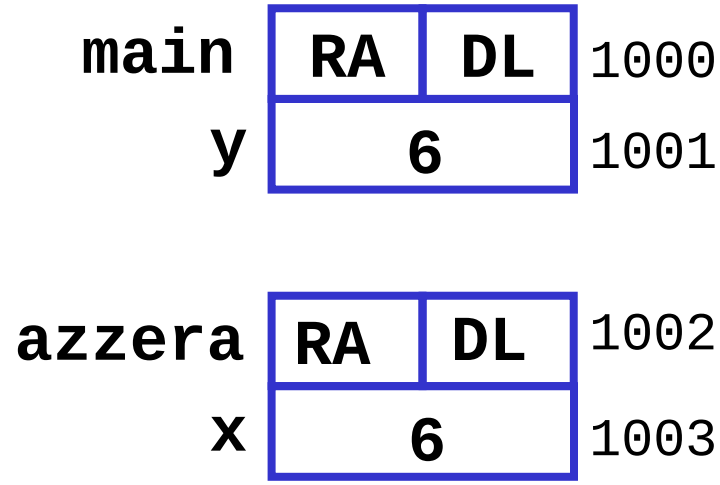




Come fare?

- Il problema è che la procedura azzera non sa qual è la variabile da azzerare: le viene passato solo il valore della variabile

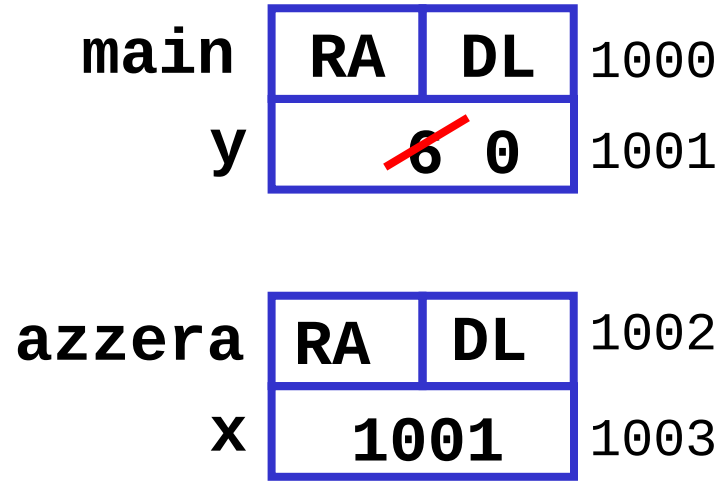
```
void azzera(int x)
{ x=0;
}
main()
{ int y=6;
  azzera(y);
}
```



Come fare?

- Ci piacerebbe poter passare alla funzione, invece del valore di **y**, il suo indirizzo. Poi dovremmo dire alla **azzera** di inserire 0 nella cella di cui sappiamo l'indirizzo

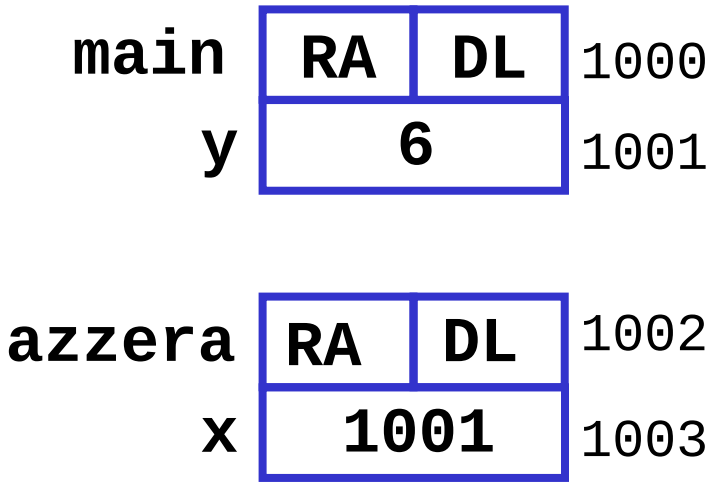
```
void azzera(int x)
{ x=0;
}
main()
{ int y=6;
  azzera(y);
}
```



Di cosa abbiamo bisogno?

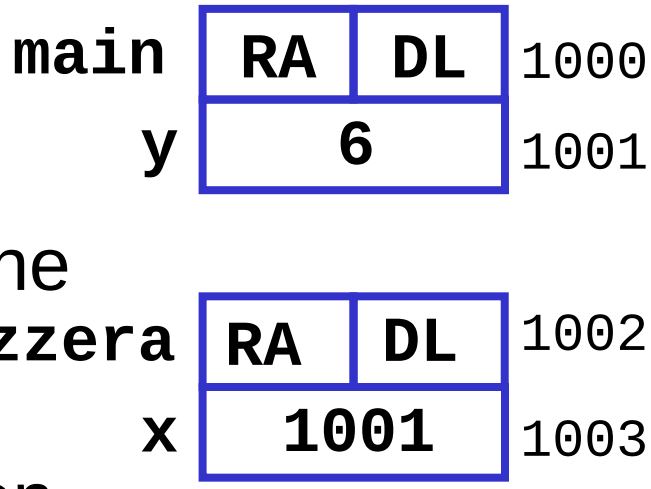
Abbiamo bisogno di 3 cose:

1. (nel **`main`**) Estrarre l'indirizzo di y (1001)
2. (nella **`azzera`**) Avere una variabile in cui possiamo mettere indirizzi
3. (nella **`azzera`**) Poter utilizzare una variabile di cui so l'indirizzo:
   - la variabile il cui indirizzo è contenuto in x



Estrazione dell'indirizzo

- Per ottenere l'indirizzo di una variabile, si usa l'operatore &
- Es:
  - &y = 1001
  - &x = 1003
- L'indirizzo di una variabile viene deciso dal compilatore
- L'operatore & può essere applicato solo alle variabili, non alle espressioni (non ha senso &(a+b), oppure &3).
- Se conosco l'indirizzo di una variabile, posso usare la variabile anche se non ne conosco il nome



La scanf

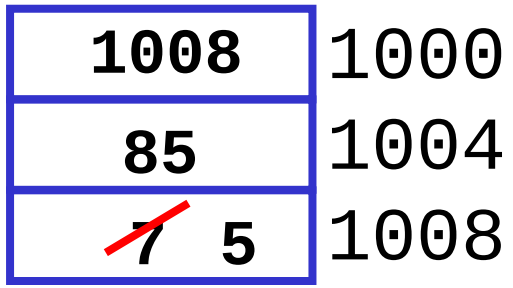
- questo è il motivo per cui nella scanf mettiamo la & davanti alle variabili: la **`scanf`** ha bisogno di sapere l'indirizzo della variabile per poterla modificare

```
scanf("%d",&x);
```

Tipo puntatore

- Per memorizzare indirizzi in memoria, si usa il tipo puntatore
- Una variabile di tipo puntatore può contenere (solo) indirizzi
- Se **p** è una variabile puntatore che contiene un indirizzo (es. 1008) posso usarlo per leggere/scrivere sulla cella 1008, usando l'operatore *****

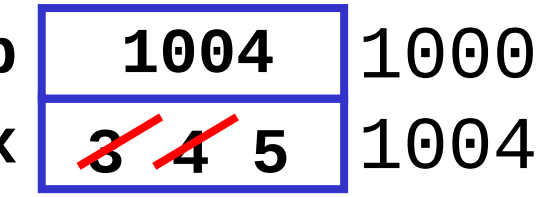
```
*p = 5;
```



Esempio

- Supponiamo che **p** sia di tipo puntatore

```
…
int x=3;
p = &x;
```

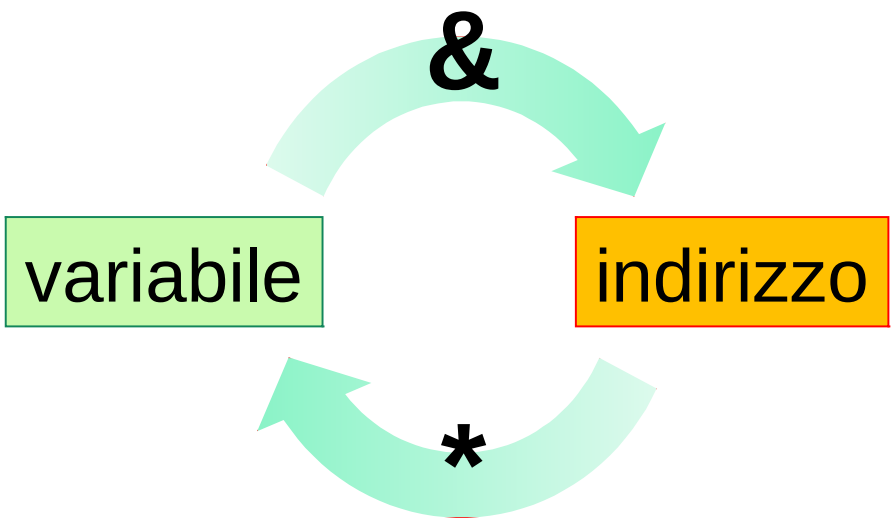


- A questo punto, dire **x** o dire ***p** è esattamente la stessa cosa: sono sinonimi, rappresentano la stessa variabile
- Quindi posso fare

```
printf("%d",*p);
*p=4;
(*p)++;
```

Operatori & e *

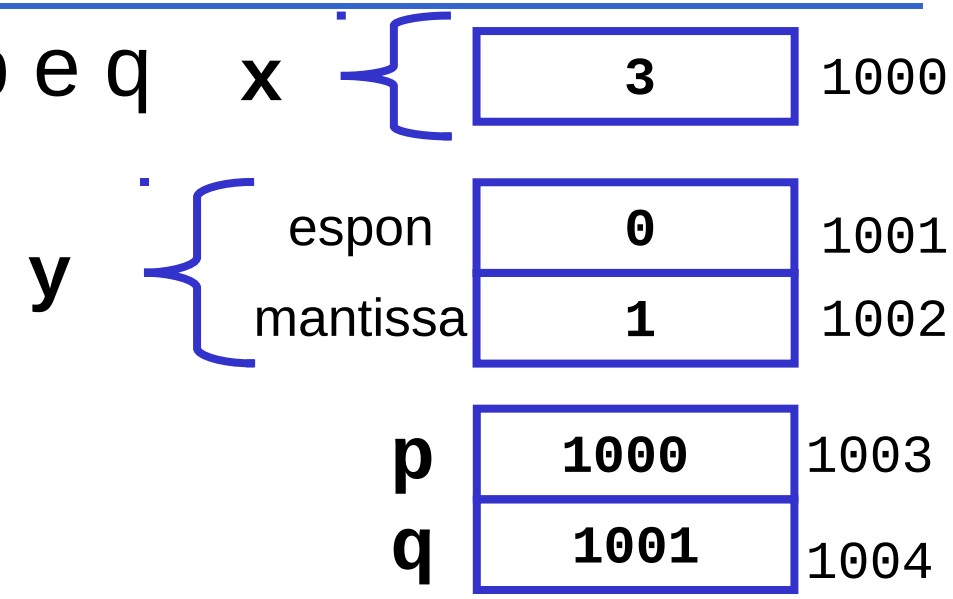
- I due operatori **&** e ***** sono uno l'inverso dell'altro
- **&** estrae l'indirizzo di una variabile
- ***** riferisce la variabile corrispondente a un indirizzo



Problema: di che tipo è *p?

- Ho due puntatori: p e q

```
char x=3;
float y=1e0;
p = &x;
q = &y;
*q = *p;
```



Quanti byte deve ricopiare?
Dovrebbe fare la promozione
`char → float,`
ma come fa a sapere che **`q`**
contiene l'indirizzo di un **`float`**?

Soluzione

- Soluzione: Quando dichiaro una variabile di tipo puntatore, dichiaro anche di che tipo è il dato di cui ho l'indirizzo.
- Quindi una variabile non è un puntatore *"e basta":*
  - un puntatore a **`int`** può contenere solo indirizzi di variabili di tipo **`int`**
  - un puntatore a **`float`** può contenere solo indirizzi di variabili di tipo **`float`**
  - un puntatore a **`struct frazione`** può contenere solo indirizzi di variabili di tipo **`struct frazione`**

SINTASSI

- Definizione di una variabile puntatore:

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

- Esempi:

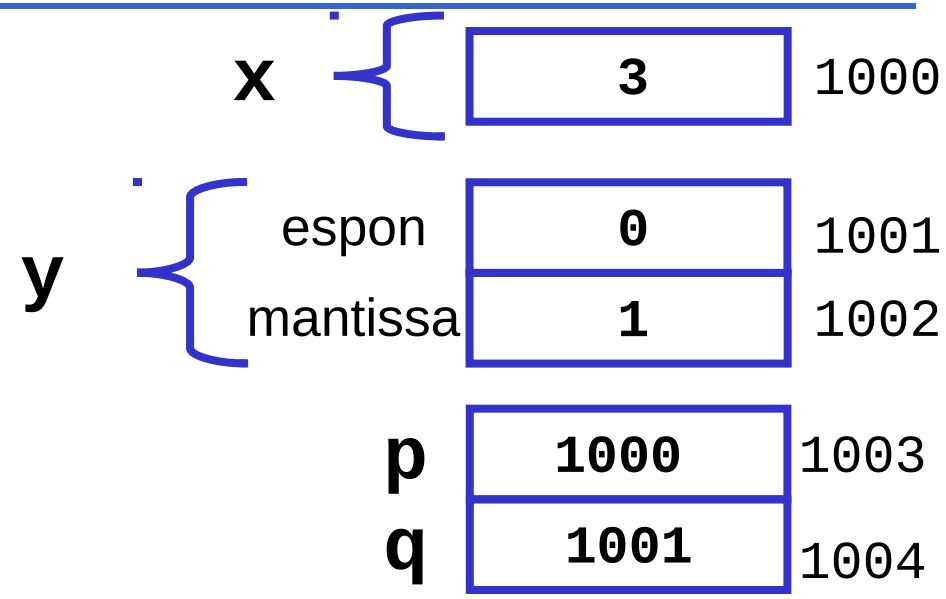
```
int  *p;
int*  p;
int * p;
```

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

Quindi

```
char x=3;
float y=1e0;
char *p;
float *q;
p = &x;
q = &y;
*q = *p;
```

float char

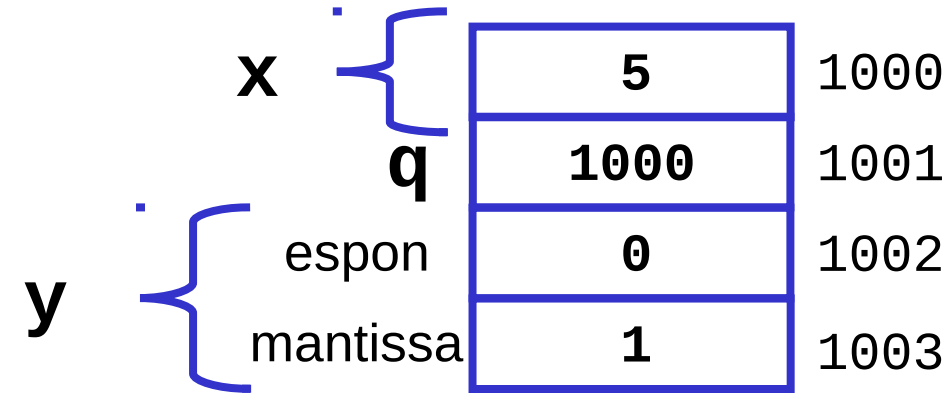


Si fa la promozione **char → float,** poi si assegna il valore **3e0** a ***q** (cioè a **y**)

e se facessi ... ?

```
char x=5;
float *q, y=1e0;
q=&x;
y = *q;
printf("%f\n",y);
```

warning: incompatible types




```
-107372584.000000
```

Passaggio per riferimento

```
void azzera(int *x)
{ *x = 0;
}

main()
{ int y=6;
  azzera(&y);
}
```

REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

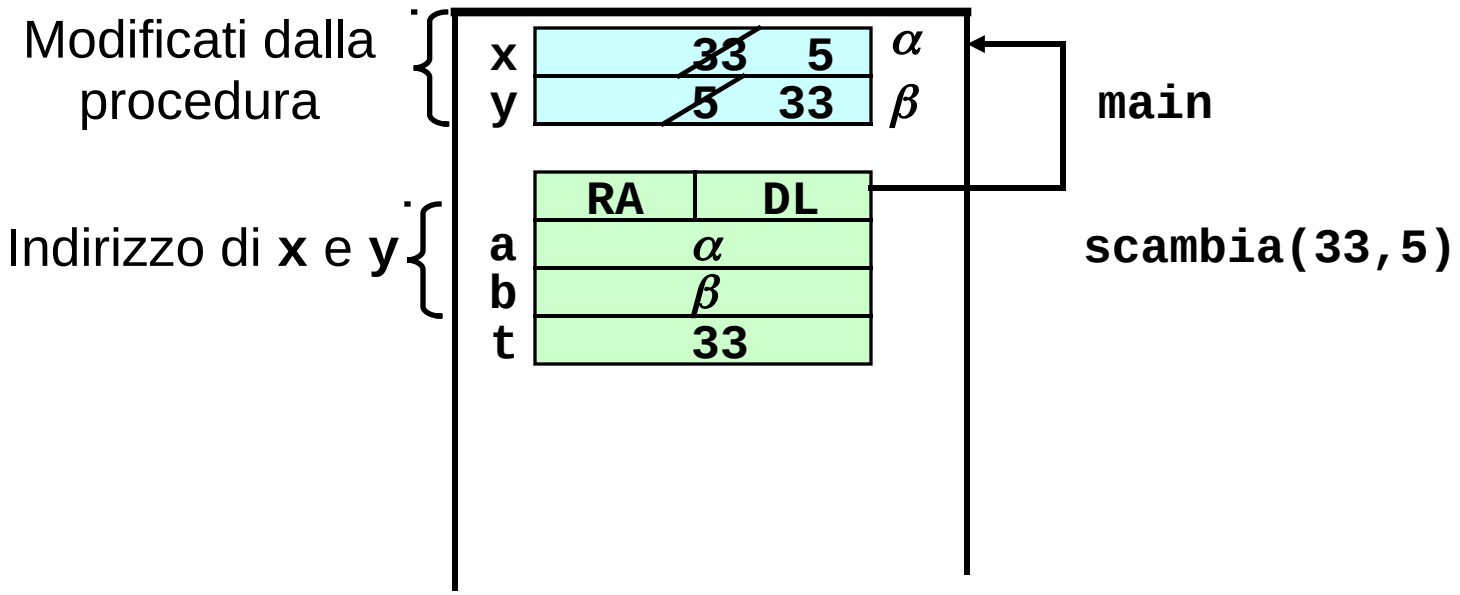
- In C per realizzare il passaggio per riferimento:
  - il cliente deve ***passare esplicitamente gli indirizzi***
  - il servitore deve ***prevedere esplicitamente dei puntatori come parametri formali***

REALIZZARE IL PASSAGGIO PER RIFERIMENTO IN C

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

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

ESEMPIO: RECORD DI ATTIVAZIONE

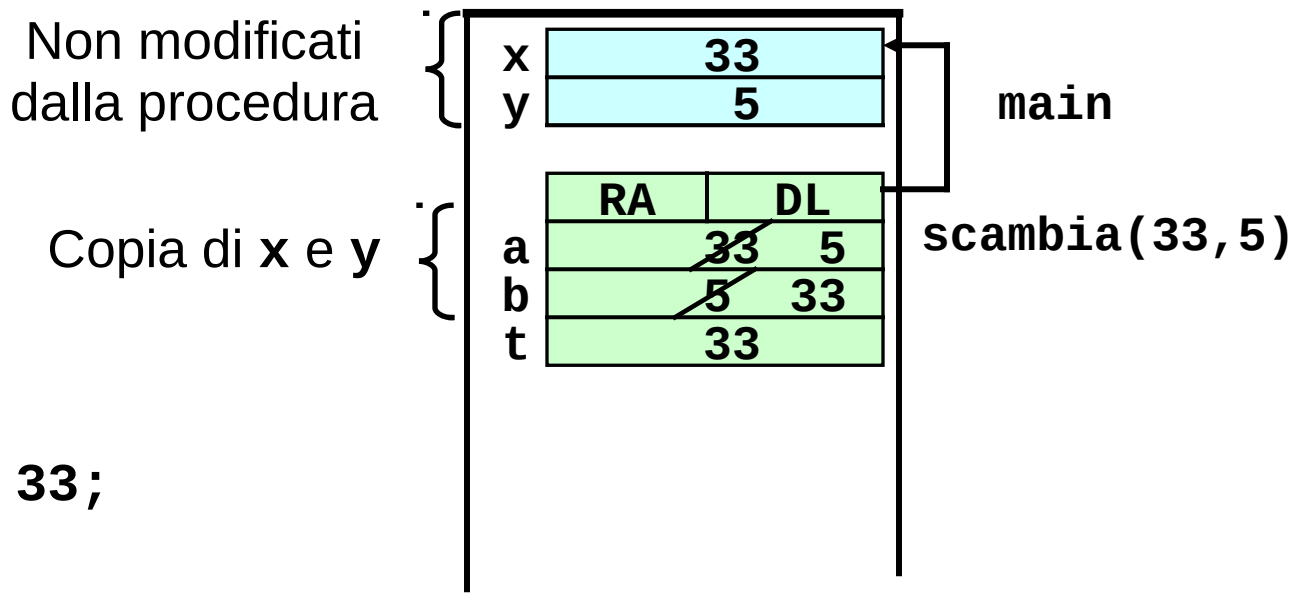
Caso del passaggio *per riferimento*:



ESEMPIO: RECORD DI ATTIVAZIONE

Se avessi usato il passaggio *per valore*:

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

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



Esercizio

- Si scriva una procedura che calcola quoziente e resto di una divisione intera

Esercizio

- Si scriva una procedura **`ordina2`** con due parametri *a* e *b*.
- Se *a*>*b*, scambia il valore di *a* e di *b*, usando la procedura **`scambia`** definita nei lucidi precedenti

Esercizio (analisi)

- Si mostri l'esecuzione del seguente programma con i record di attivazione

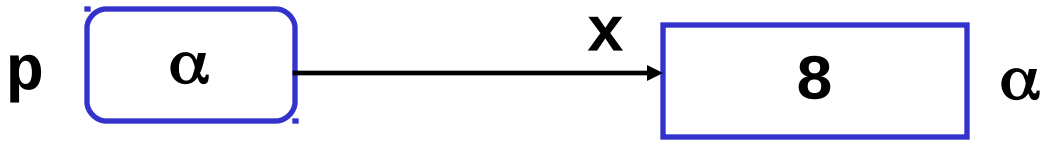
```
void g(int *h)
{ (*h)++;
}
int f(int a, int *b)
{ g(b);
  return a+(*b);
}
main()
{ int c=1, d=3, s=6;
  s=f(c,&d);
}
```

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:

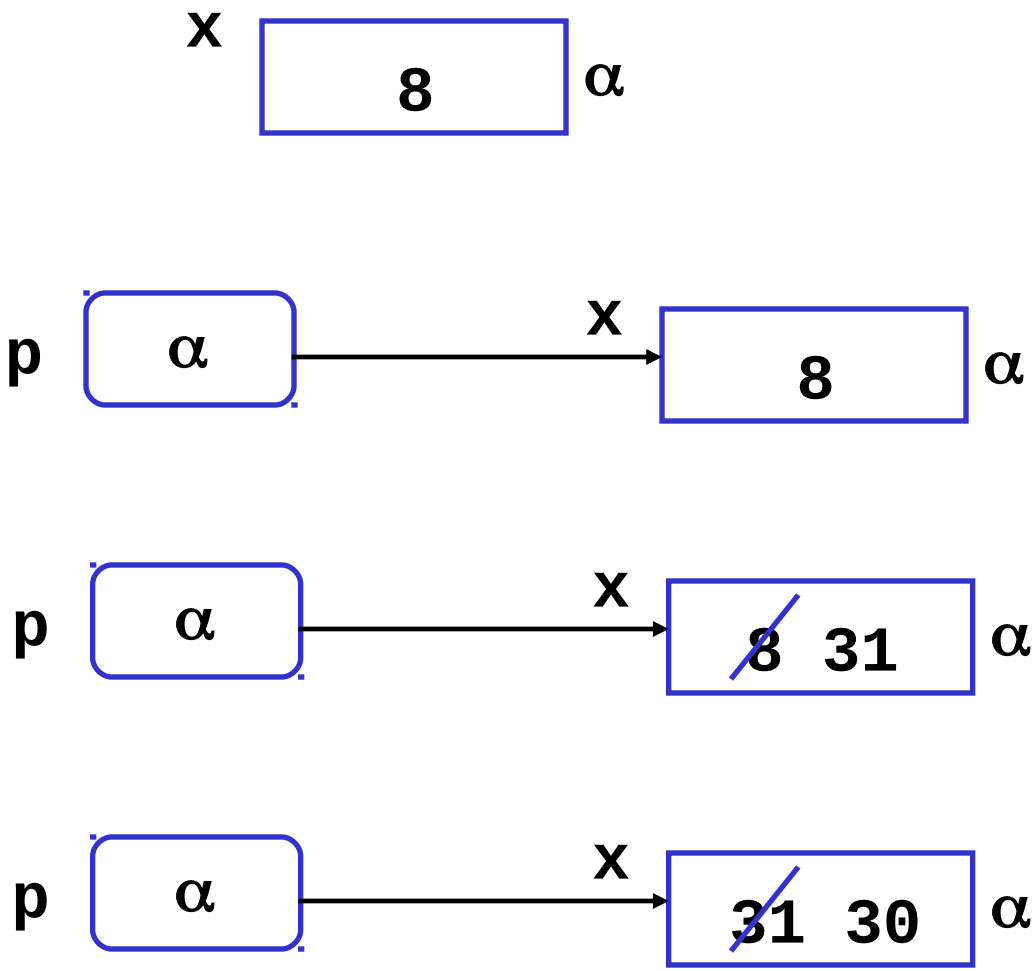



```
int  x = 8;
int* p;
p = &x;
```

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

PUNTATORI

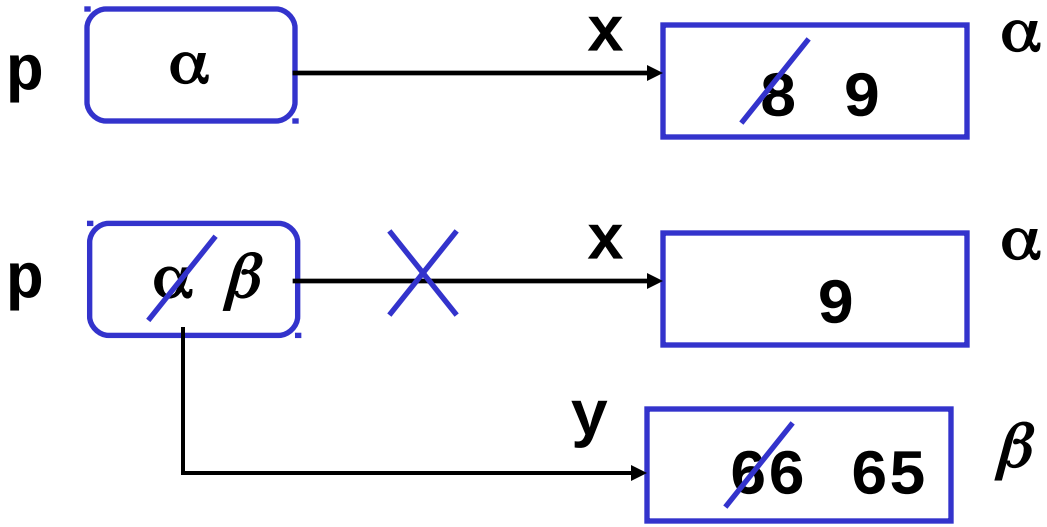
```
int  x = 8;

int* p = &x;

*p = 31;

x--;
```



PUNTATORI

Un puntatore non è legato per sempre alla stessa variabile: può puntare altrove.

```
int x = 8, y = 66;
int *p = &x;
(*p)++;

 p = &y;
(*p)--;
```



Le parentesi sono necessarie per riferirsi alla variabile puntata da **p**

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*

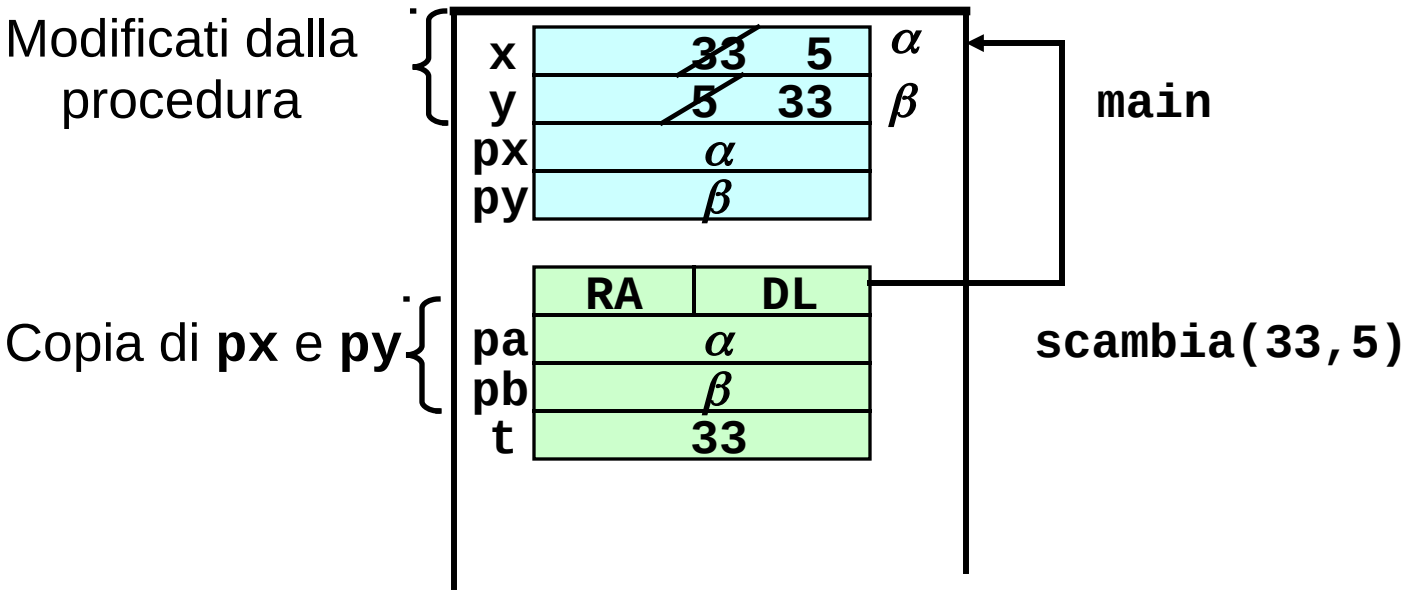
PUNTATORI

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

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.



Esercizio

- Si scriva un programma che legge una struttura frazione tramite una procedura e la stampa tramite un'altra procedura.

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

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;

void dividi(int x, int y)

{ resto = x % y; quoziente = x/y;
}

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

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

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

LEGGIBILITA`

Le variabili globali vanno usate il meno possibile, in quanto rendono meno leggibile il programma

```
int quoziente, int resto;

void dividi(int x, int y)

{ resto = x % y; quoziente = x/y;
}

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

Osservando il main program non capisco che **quoziente** e **resto** vengono modificate dalla **dividi** ☹

ESEMPIO

Con il passaggio dei parametri per indirizzo avremmo un codice più leggibile

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

main()
{ int quoz, rest;
  dividi(33, 6, &quoz, &rest);
  printf("%d %d", quoz,rest);
}
```

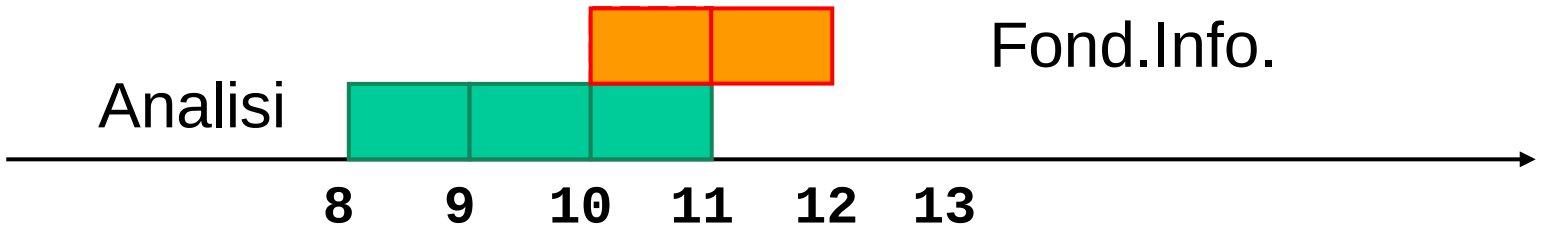
Vedo subito che **quoz** e **rest** sono passate per riferimento, quindi capisco che **dividi** può modificarne il valore ☺

Esercizio

- L'orario di una lezione è rappresentato dalla struttura:

```
typedef struct
{ char nomecorso[20];
  int orainizio, durata;
} orario;
```

- Si scriva una procedura o funzione che, date 2 lezioni, verifica se si sovrappongono e, qualora si sovrappongano, sposti in avanti la lezione che inizia più tardi, in modo che inizi appena finisce l'altra lezione
- Nota: non sappiamo a priori quale delle due lezioni comincia prima (bisogna controllare il valore di **`orainizio`**)



Esercizio

Si scriva una funzione (o procedura) che prende in ingresso due frazioni e fornisce in uscita

- la frazione quoziente
- un valore boolean che dice se il risultato è finito o se c’è stata divisione per zero