

# Gestione della memoria

Statica, a pila, con heap. Implementazione delle regole di scope

Domanda: Come viene gestita la MEMORIA CENTRALE (RAM) ?

M. Gabbielli, S. Martini

Linguaggi di programmazione:

*principi e paradigmi*

McGraw-Hill Italia, 2005

# Tipi di allocazione della memoria

COSA SERVE LA MEMORIA nell'esecuzione dei programmi?

SERVE PER  
memorizzare i dati  
(in genere) del programma  
e memorizzare anche informazioni  
per il controllo dell'esecuzione  
(es. analogo del program counter)

- La vita di un oggetto corrisponde (in genere) con tre meccanismi di allocazione di memoria:

alloc = riserva (la memoria)

– **statica**: memoria allocata a tempo di compilazione (es. variabili globali)

– **dinamica**: memoria allocata a tempo d'esecuzione

• pila (stack):

– oggetti allocati con politica LIFO

• heap:

– oggetti allocati e deallocati in qualsiasi momento (puntatori)

Posso gestire la memoria solo in maniera statica?  
E.g. ho salv valori interi

→ Se ho memoria, serve una parte di gestione della memoria a mem-time (dinamico)

## Allocazione statica

- Un oggetto ha un indirizzo assoluto che è mantenuto per tutta l'esecuzione del programma
- Solitamente sono allocati staticamente:
  - variabili globali
  - variabili locali sottoprogrammi (senza ricorsione)
  - costanti determinabili a tempo di compilazione
  - tabelle usate dal supporto a run-time (per type checking, garbage collection, ecc.)
- Spesso usate zone protette di memoria

# Allocazione statica per sottoprogrammi

→ quando nel mio programma ho  
molti sottoprogrammi

→ alloco la memoria che mi serve

Registri Salvati

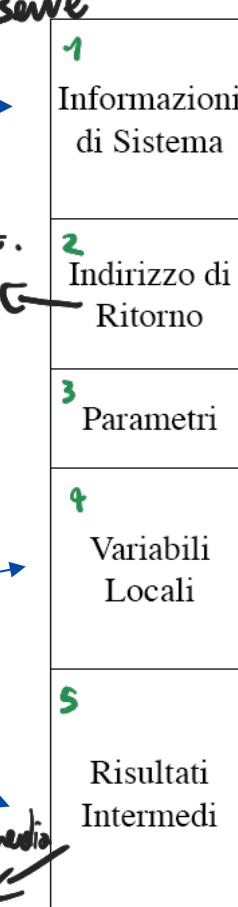
Informazione  
debugging

quando una funz.  
termina, e' l'  
indirizzo di memoria  
dove devo andare e dove  
trovo i valori di ritorno

spesso nei registri

Spazio per i risultati intermedi

- esplaz. molto lunghe,  
per spezzarle in pezzi → i cui risultati vengono salvati li
- risultati delle chiamate ricorsive

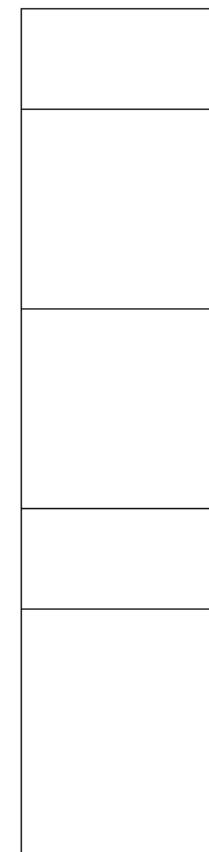


$n$  procedure / Funzioni

Non posso avere due istanze di procedure attive assieme



...



Procedure  $n$

# L'allocazione statica non permette ricorsione

## FORTRAN.

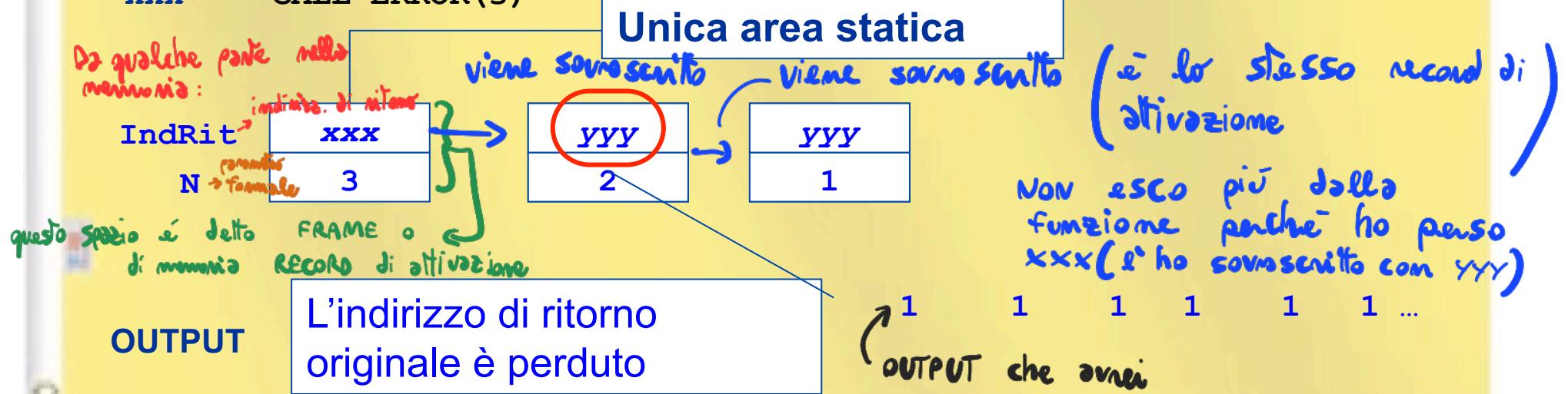
Programma sintatticamente illegale: non ammessa chiamata ricorsiva

indirizzi di memoria:

```
FUNCTION  
SUBROUTINE ERROR (N)  
IF (N.LE.1) RETURN  
CALL ERROR (N-1)  
PRINT N  
END
```

Supponiamolo legale:  
eseguiamolo nel modello  
di memoria statica

...  
xxx CALL ERROR (3) <sup>può essere un qualsiasi numero</sup>



## Allocazione dinamica: pila

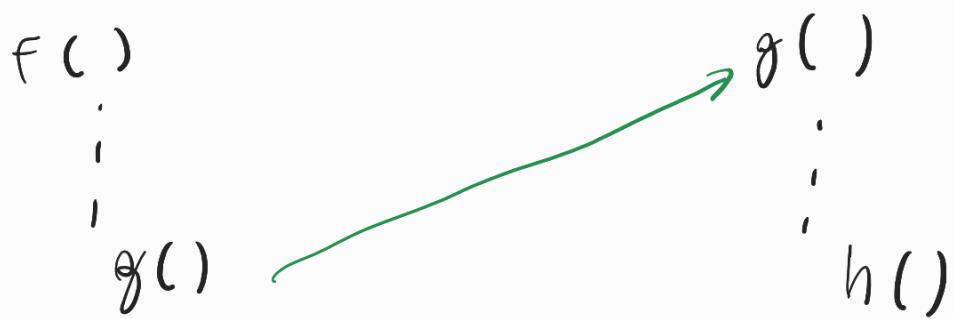
(stack di sistema)

Last in - First out: ultimo record di attivaz. messo è il primo che esce

- Con ricorsione l'allocazione statica non basta:
  - a run time possono esistere più istanze della stessa variabile locale di una procedura  
→ nell'allocazione statica non può succedere
- Ogni istanza di un sottoprogramma a run-time ha una porzione di memoria detta record di attivazione (o frame) contenente le informazioni relative alla specifica istanza (indirizzo ritorno!!)
- Analogamente, ogni blocco ha un suo record di attivazione
  - (più semplice)
- La Pila (LIFO) è la struttura dati naturale per gestire i record di attivazione perché le chiamate di procedura (anche ricorsiva) ed i blocchi sono annidati uno dentro l'altro
- Anche in un linguaggio senza ricorsione può essere utile usare la pila per memorizzare le variabili locali per risparmiare memoria

→ anziché usare l'allocazione statica per le variabili locali

Esempio:



## Gestione di memoria: PILA

→ Non posso far terminare  $f()$  prima di aver terminato  $g()$  (nei linguaggi sequenziali)

unica eccezione: le eccezioni (in cui tale scenario potrebbe verificarsi)

# Record di attivazione per blocchi anonimi

Puntatore di Catena Dinamica

→ È il puntatore che punta al record di attivazione (Rda) successivo

Variabili Locali

Risultati Intermedi

## Allocazione dinamica con pila

→ Ricordo che ogni blocco ha un suo record di attivazione

- La gestione della pila è compiuta mediante:
  - sequenza di chiamata (il codice eseguito dal chiamante immediatamente prima della chiamata)
  - prologo (codice eseguito all'inizio del blocco)
  - epilogo (codice eseguito alla fine del blocco)
  - sequenza di ritorno (il codice eseguito dal chiamante immediatamente dopo la chiamata)
- Indirizzo di un RdA non è noto a compile-time. (*moto solo a run-time*)
- Il Puntatore RdA (o SP) punta al RdA del blocco attivo.
- Le info contenute in un RdA sono accessibili per offset rispetto allo SP:  $\rightarrow \text{STACK POINTER}$   
indirizzo-info = contenuto(SP)+offset
- Offset determinabile staticamente dal compilatore
- Somma SP+offset eseguita con unica istruzione macchina **load o store**

## CHIAMATA DI PROCEDURA:

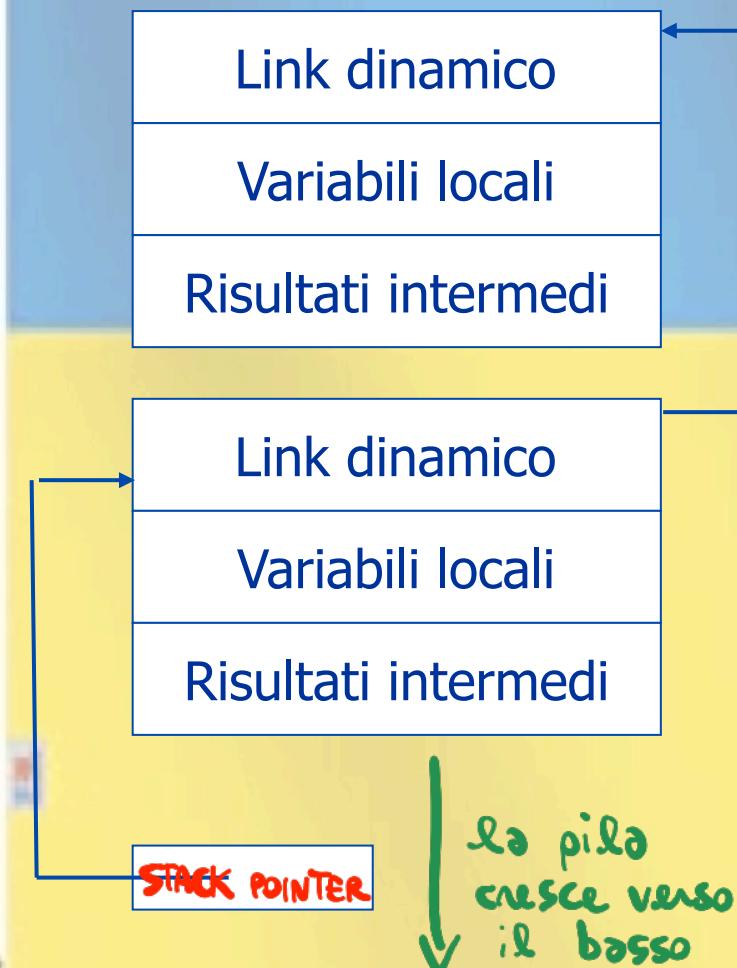
SC  
!  
Sequenza  
di chiamate  
(di procedura)

{ prologo  
epilogo

→ Quando apro delle parentesi

## Record di attivazione per blocchi in-line

Sp → STACK POINTER punta al campo link dinamico dell' Rda corrente (in cima alla pila)



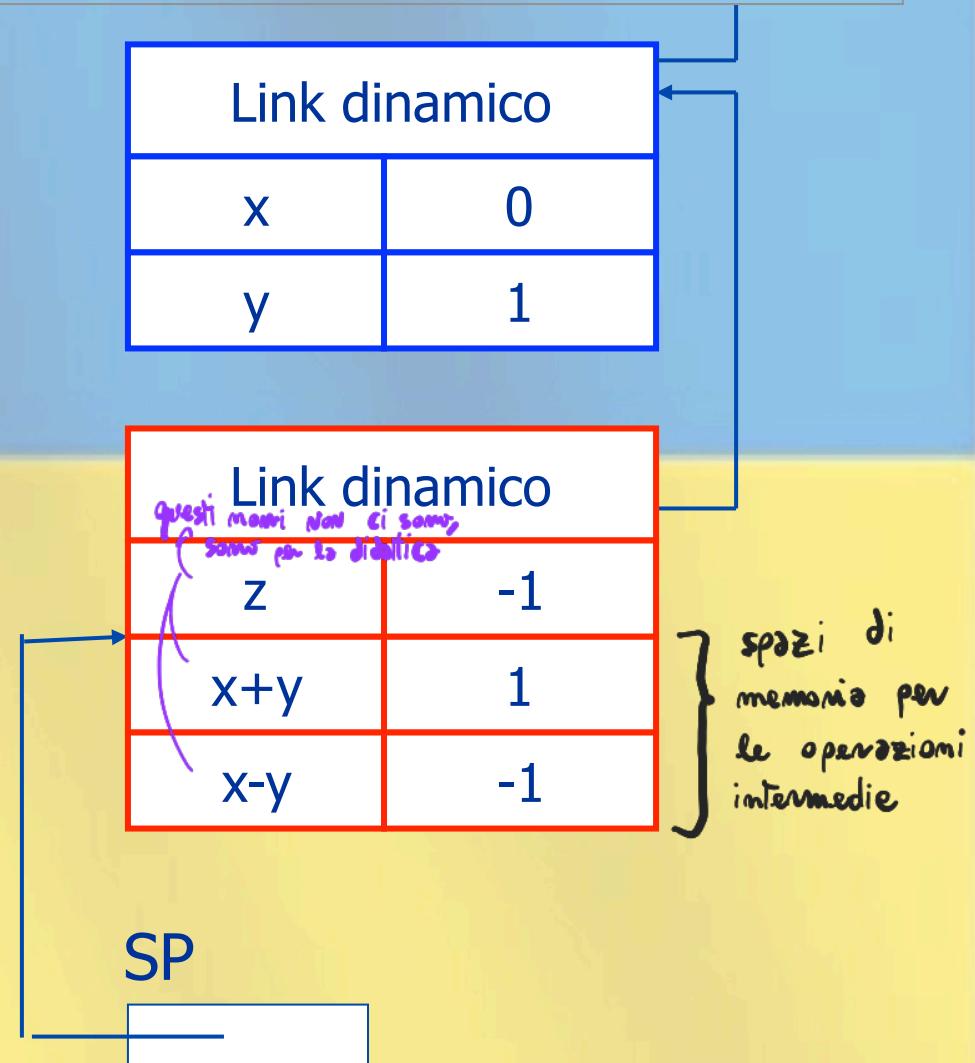
- **Link dinamico (o control link)**
  - puntatore al precedente record sullo stack
- **Ingresso nel blocco: Push**
  - link dinamico del nuovo Rda := SP
  - SP aggiornato a nuovo Rda
- **Uscita dal blocco: Pop**
  - Elimina RdA puntato da SP
  - SP := link dinamico del Rda tolto dallo stack

# Esempio

```
{ int x=0;  
    int y=x+1;  
    { int z=(x+y)*(x-y);  
    };  
};
```

Push record con spazio per x, y  
Setta valori di x, y  
Push record blocco interno  
Setta valore per z  
Pop record per blocco interno  
Pop record per blocco esterno

osserva: nel blocco **interno** l'accesso alle vars non locali x e y non può avvenire per (SP) +offset. **In prima approssimazione:** si deve risalire la catena dinamica.



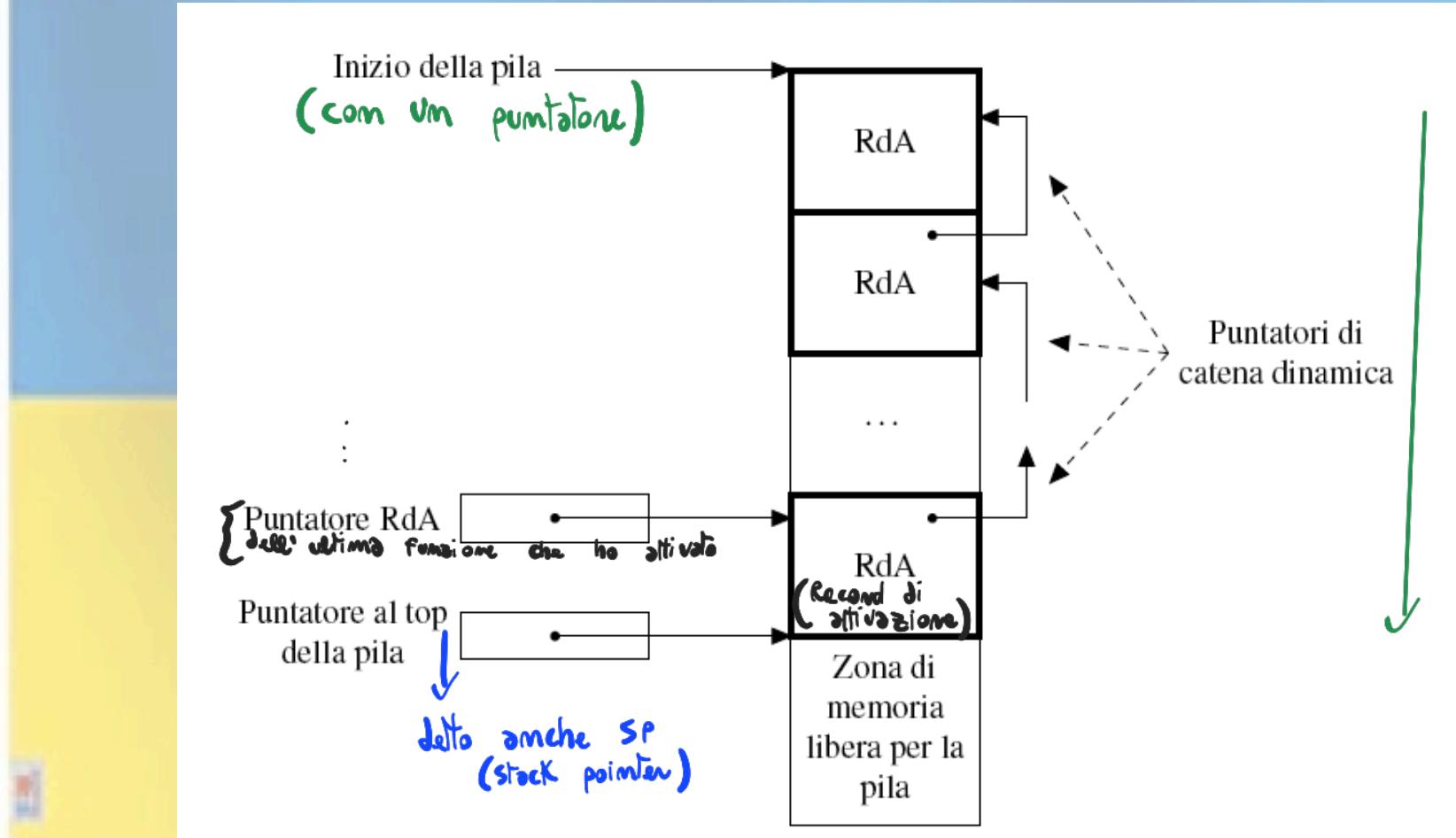
## In realtà...

- In molti linguaggi non c'è manipolazione della pila per i blocchi anonimi !
- Tutte le dichiarazioni dei blocchi annidati sono raccolte dal compilatore
- Allocazione di spazio per tutte
- Potenziale spreco di memoria, ma...
- Nessuna perdita di efficienza per la gestione della pila

# Record di attivazione per procedure (funzione)

Puntatore di Catena Dinamica
Puntatore di Catena Statica <i>gestisce lo scope statico</i>
Indirizzo di Ritorno
Indirizzo del Risultato <i>quando la funzione termina</i> → in questa zona di memoria viene salvato il valore di ritorno (se c'è)
Parametri
Variabili Locali
Risultati Intermedi

# Gestione della pila



## Perché il link dinamico e il puntatore RdA?

- Gli RdA non hanno tutti la stessa dimensione:
  - come fare pop? Occorre:
    - dimensione, oppure
    - indirizzo del RdA sotto di lui => link dinamico
- In un RdA possono esserci dati di dimensione variabile a run-time: p.e. array la cui dimensione non è nota a tempo di compilazione.
- Puntatore al top punta alla cima dell'RDA: **come ottenere offset per le var locali ?**
- Si usa invece il puntatore a RdA: punta a posizione per cui offset dei locali sempre determinabile dal compilatore (eccetto i locali di dimensione variabile ==> vedi dopo: Tipi).

# Esempio

RdA:



```
int fact (int n) {  
    if (n<= 1) return 1;  
    else return n * fact(n-1);  
}
```

- **Parametri**

- settati al valore di `n` dalla sequenza di chiamata

- **Ind. ritorno risultato**

- indirizzo della locazione dove mettere il valore finale di `fact(n)` (in RdA chiamante)

- **Risultati Intermedi**

- spazio per contenere il valore di `fact(n-1)`

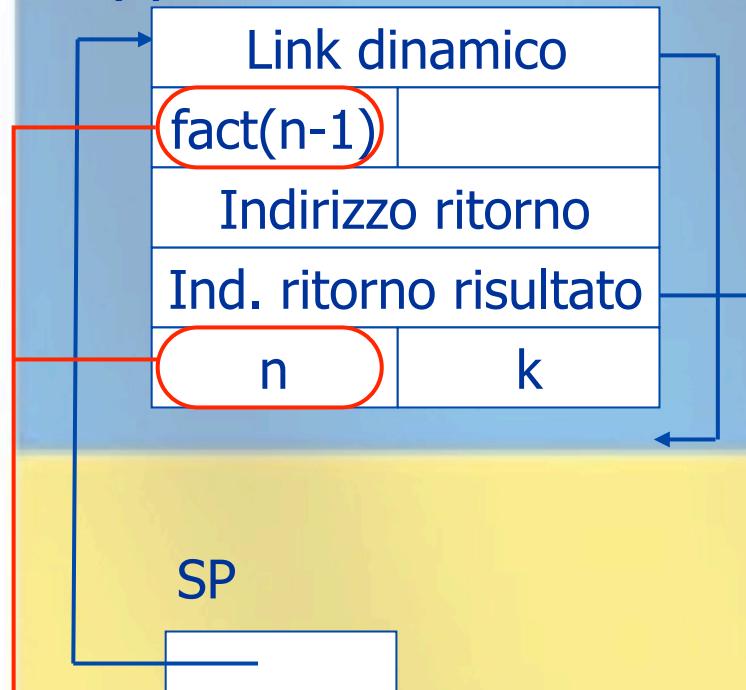
- **Variabili locali**

- non presente in questo caso

**non ci preoccupiamo oltre di punt RdA**

## Chiamata della funzione: fact (3) ;

fact(k)



```
{int fact (int n) {
    if (n<= 1) return 1;
    else return n * ... (a capo)
    fact(n-1);
```

I nomi non sono presenti: solo  
in una situazione  
non vedremo  
solo indirizzi di  
memoria

fact(3)

saiuto qua  
(puntatore)

fact(2)

fact(1)

Link dinamico

fact(n-1)

*indirizzo della cella di memoria  
dove c'è la chiamata di funzione*

Ind. ritorno risultato

n 3

Link dinamico

fact(n-1)

punt nel codice di fact

Ind. ritorno risultato

n 2

Link dinamico

fact(n-1)

punt nel codice di fact

Ind. ritorno risultato

n 1

## Ritorno dalla funzione

fact(3)

Link dinamico	
fact(n-1)	
punt nel codice del <b>main</b>	
Ind. ritorno risultato	
n	3
Link dinamico	
fact(n-1)	
punt nel codice di <b>fact</b>	
Ind. ritorno risultato	
n	2
Link dinamico	
fact(n-1)	
punt nel codice di <b>fact</b>	
Ind. ritorno risultato	
n	1

fact(2)

```
{int fact (int n) {
    if (n<= 1) return 1;
    else return n *
        fact(n-1);}
```

Link dinamico	
fact(n-1)	2
punt nel codice del <b>main</b>	
Ind. ritorno risultato	
n	3
Link dinamico	
fact(n-1)	1
punt nel codice di <b>fact</b>	
Ind. ritorno risultato	
n	2

fact(1)

## Gestione della pila: ingresso in blocco

- Sequenza di chiamata e prologo si dividono i seguenti compiti:
  - Modifica del contatore programma
  - Allocazione RdA sulla pila (modifica puntatore a top)
  - Modifica del puntatore al RdA
  - Passaggio dei parametri
  - Salvataggio dei registri
  - Eventuali inizializzazioni
  - Trasferimento del controllo

## Gestione della pila: uscita da blocco

- Sequenza di uscita ed epilogo si dividono i seguenti compiti:
  - Restituzione dei valori dal chiamato al chiamante, oppure il valore calcolato dalla funzione
  - Ripristino dei registri
    - In particolare deve essere ripristinato il vecchio valore del puntatore al RdA.
  - Eventuale finalizzazione
  - Deallocazione dello spazio sulla pila
  - Ripristino del valore del contatore programma

## Allocazione dinamica con heap

→ significa "macchia"

≠ dalla PILA

- **Heap:** regione di memoria i cui (sotto) blocchi possono essere allocati e deallocated in momenti arbitrari
- Necessario quando il linguaggio permette
  - allocazione esplicita di memoria a run-time (e.g. puntatori e strutture dati dinamiche quali liste, alberi .... )
  - oggetti di dimensioni variabili (stringhe, insiemi ...)
  - oggetti la cui vita non ha un regime definito a priori (cioè con vita non LIFO)
- La gestione dello heap non è banale
  - gestione efficiente dello spazio: frammentazione
  - velocità di accesso

Esempio:

//alloco esplicitamente la memoria per A

malloc (A)

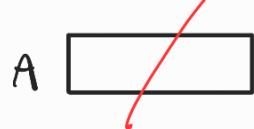


//alloco esplicitamente la memoria per B

malloc (B)



free (A)



free (B)

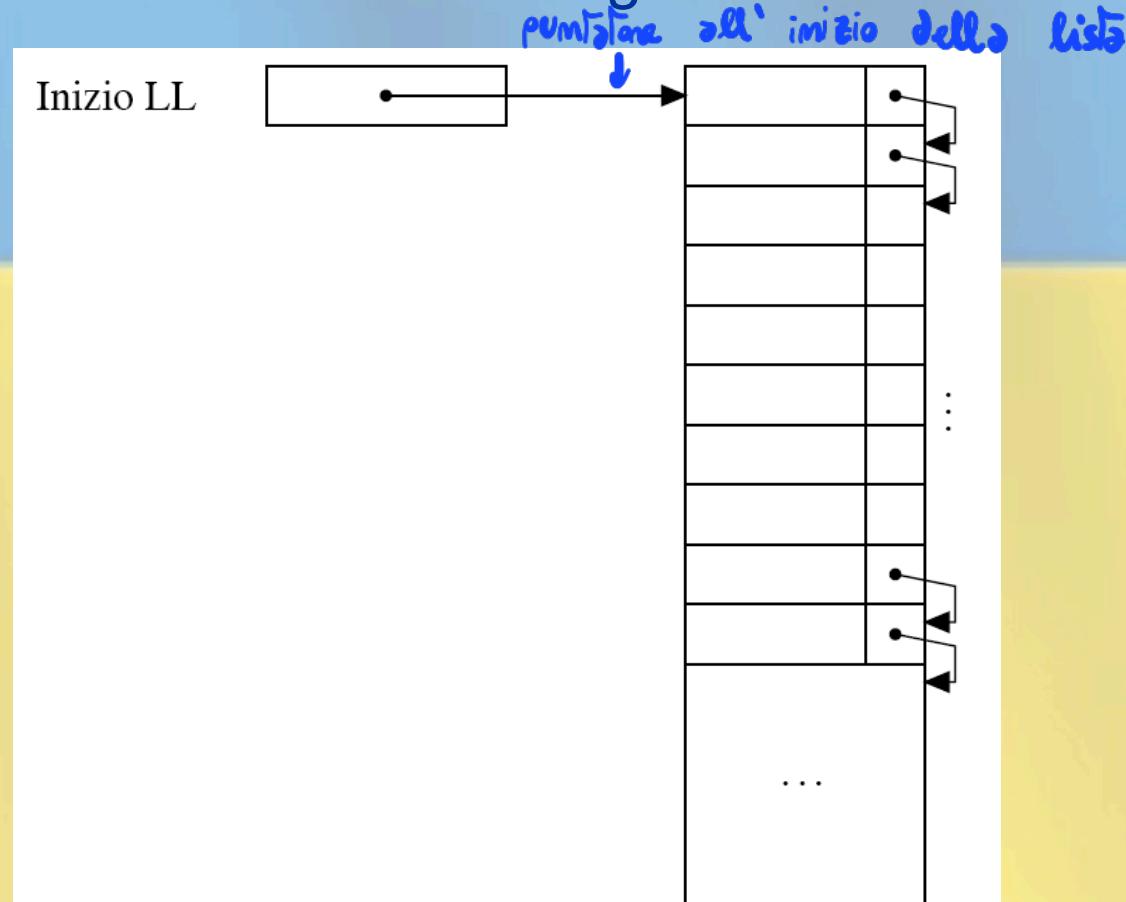
→ Con la STACK non lo posso fare, devo deallocare  
primo B e poi A (politica LIFO della PILA)

IMPLEMENTATO con la struttura dati lista libera

## Heap: blocchi di dimensione fissa

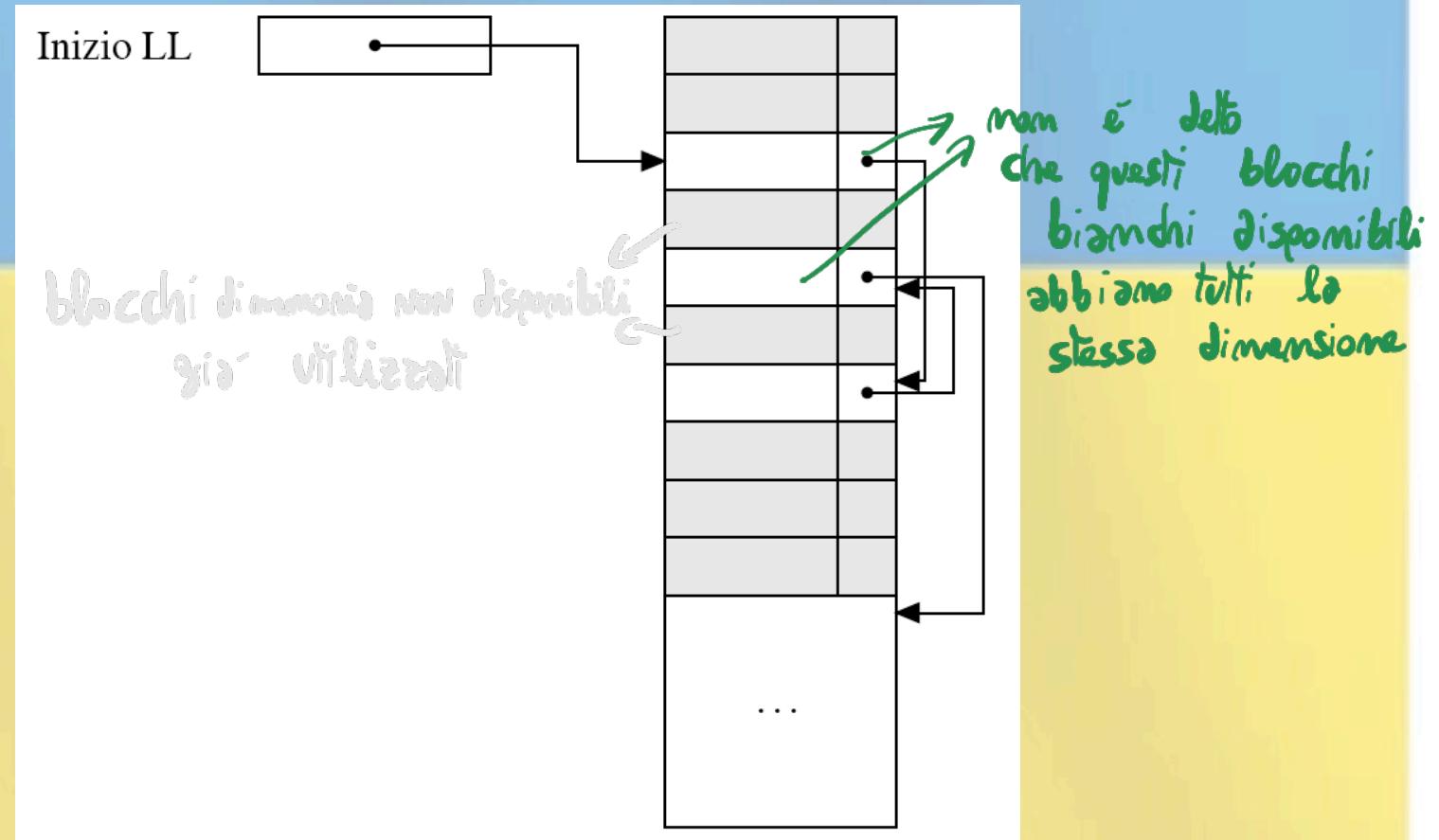
IMPLEMENTAZIONE con  
struttura dati lista libera

- Heap suddiviso in blocchi di dimensione fissa
  - e abbastanza limitata: qualche parola
- In origine: tutti i blocchi collegati nella *lista libera*



## Heap: blocchi di dimensione fissa

- Allocazione di uno o più blocchi contigui (push)
- Deallocazione: restituzione alla lista libera (pop)

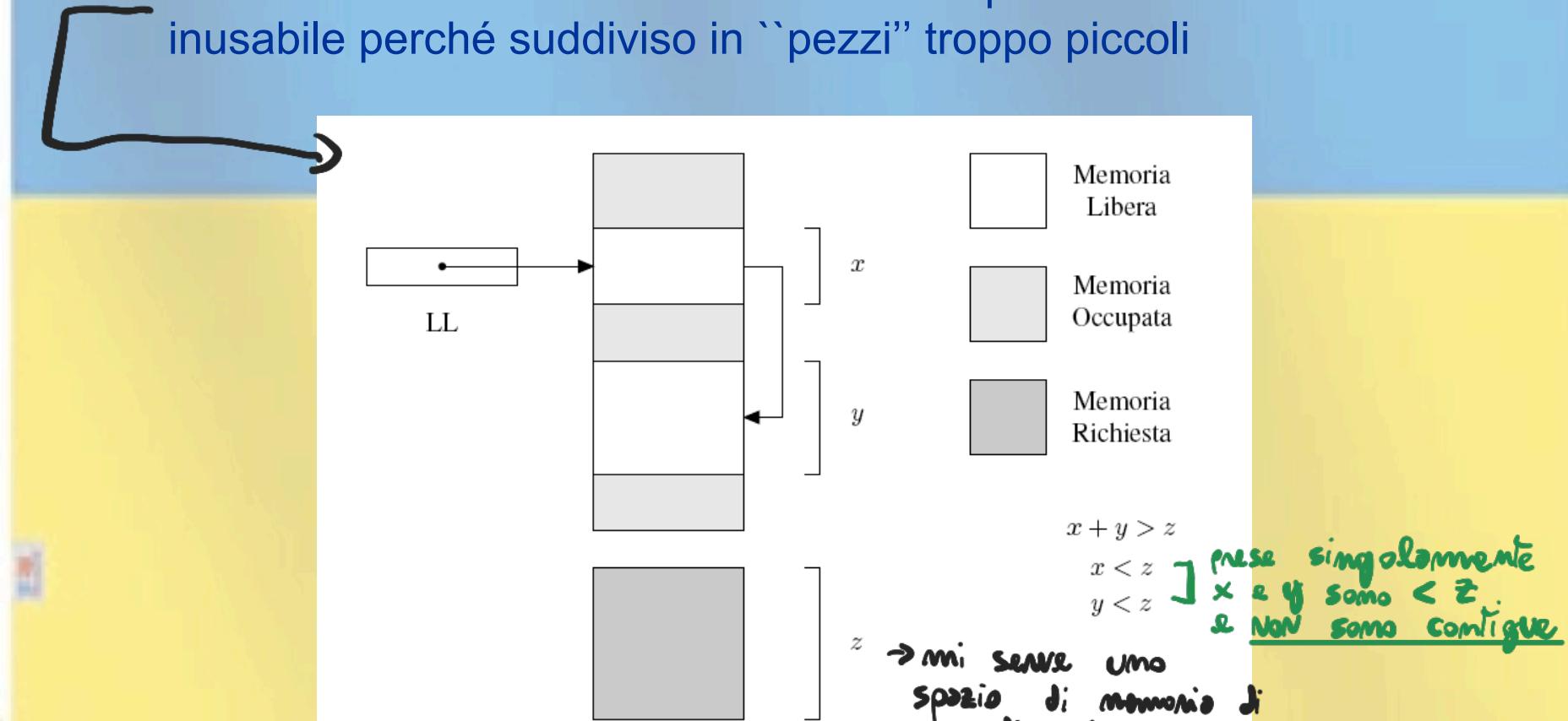


## Heap: blocchi di dimensione variabile

- Inizialmente unico blocco nello heap
- Allocazione: determinazione di un blocco libero della dimensione opportuna
- Deallocazione: restituzione alla lista libera
- Problemi:
  - le operazioni devono essere efficienti
  - evitare lo spreco di memoria
    - frammentazione interna
    - frammentazione esterna

# Frammentazione

- Frammentazione **interna**: lo spazio richiesto è  $X$ ,  
*Pendo un blocco di dimensione Y con  $Y > X$*   
– viene allocato un blocco di dimensione  $Y > X$ ,  
– lo spazio  $Y-X$  è sprecato
- Frammentazione **esterna**: ci sarebbe lo spazio necessario ma è inusabile perché suddiviso in ``pezzi'' troppo piccoli



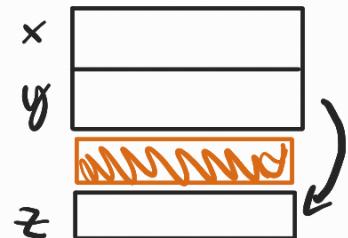
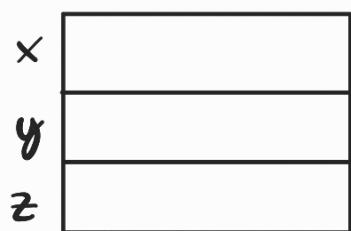
UNITÀ MINIMA di ALLOCAZIONE : è il BLOCCO di memoria

Perche le celle di memoria devono essere CONTIGUE (cioè una dopo l'altra)?

Ad una funzione serve una certa quantità di memoria per memorizzare dei dati  
(es. variabili)

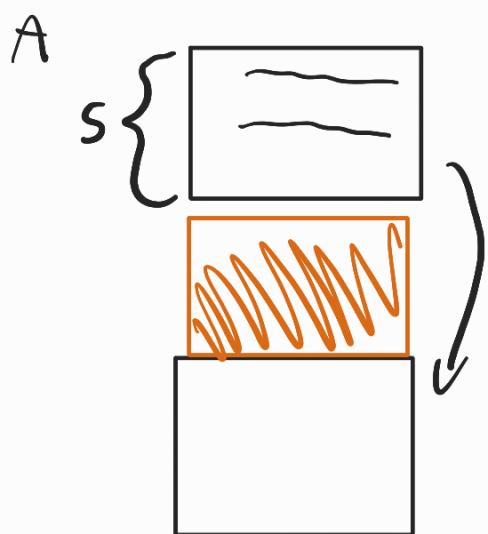
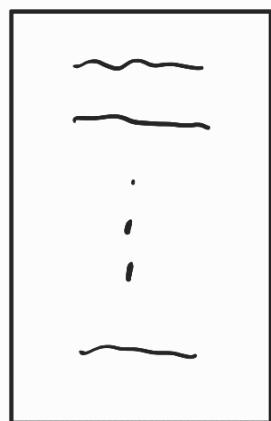
Esempio:

```
f( )  
{  
    int x, y, z;  
}
```



Altro es:

array A[10]



→ Per accedere in questo modo all'array con puntatori a celle di memoria non conviene (bisogna scomere tutti i puntatori)

## Gestione della lista libera: unica lista

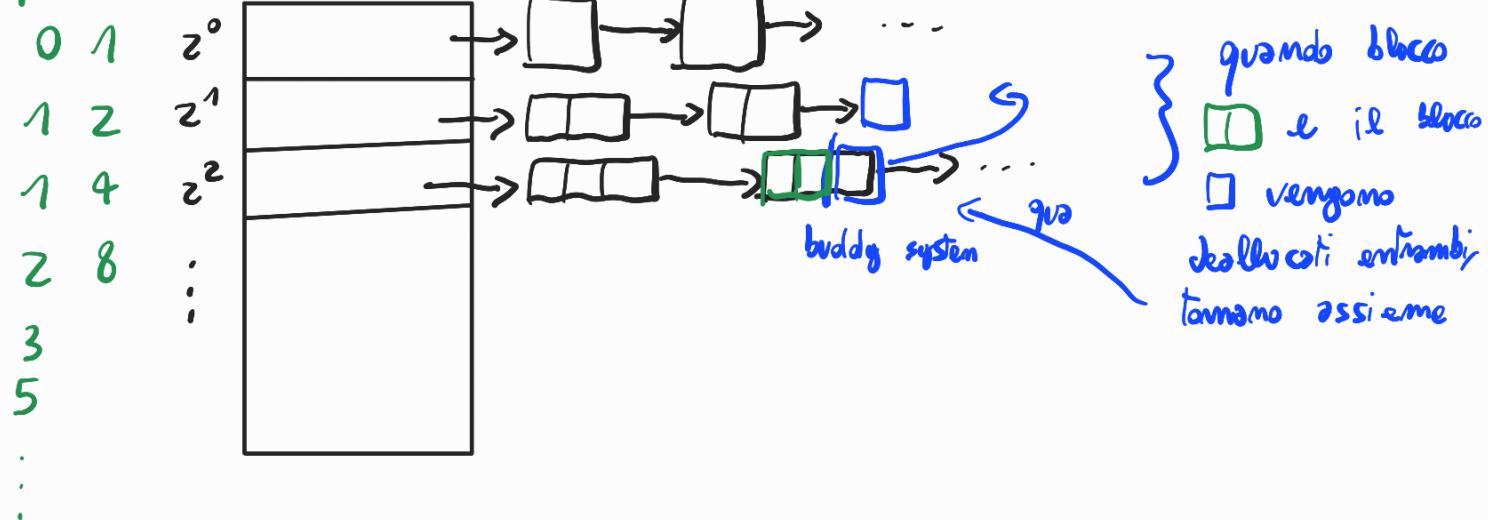
- Inizialmente contiene un solo blocco, della dimensione dello heap  
→ quando un blocco è molto grande rispetto allo spazio che ci serve tale blocco si divide in 2 modi:
  - 1) lo prendiamo
  - 2) lo restituiamo alla lista libera (lo heap)
- Ad ogni richiesta di allocazione: cerca blocco di dimensione opportuna - secondo due modalità:
  - first fit: primo blocco grande abbastanza → più veloce, ma spreca più memoria
  - best fit: quello di dimensione più piccola, grande abbastanza
    - più lenta, ma spreca meno memoria
    - sufficienti per il blocco che voglio allocare
- Se il blocco scelto è molto più grande di quello che serve, viene diviso in due e la parte inutilizzata è aggiunta alla LL → Lista Libera
- Quando un blocco è de-allocato, viene restituito alla LL (se un blocco adiacente è libero, i due blocchi sono ``fusi'' in un unico blocco) → evita problemi grossi di frammentazione

## Gestione heap

- First fit o Best Fit ? Solita situazione conflittuale:
  - First fit: più veloce, occupazione memoria peggiore
  - Best fit: più lento, occupazione memoria migliore
- Con unica LL costo allocazione comunque lineare nel numero di blocchi liberi. Per migliorare:
  - mantieni liste libere multiple

Posso avere UU multiple, con blocchi di dimensioni diverse

seq. fibonaci



# Liste libere multiple

- liste libere multiple, per blocchi di dimensione diversa
  - la ripartizione dei blocchi fra le varie liste può essere
    - statica
    - dinamica: Buddy system o Fibonacci system
  - Buddy system: k liste; la lista k ha blocchi di dimensione  $2^k$ 
    - se richiesta allocazione per blocco di  $2^k$  è tale dimensione non è disponibile, blocco di  $2^{k+1}$  diviso in 2
    - se un blocco di  $2^k$  e' de-allocato è riunito alla sua altra metà (buddy, amico)
  - Fibonacci simile, ma si usano numeri di Fibonacci invece di potenze di 2 (crescono più lentamente)

## Implementazione delle regole di scope

- Scope statico
  - catena statica
  - display
- Scope dinamico
  - A-list
  - Tabella centrale dell'ambiente (CRT)

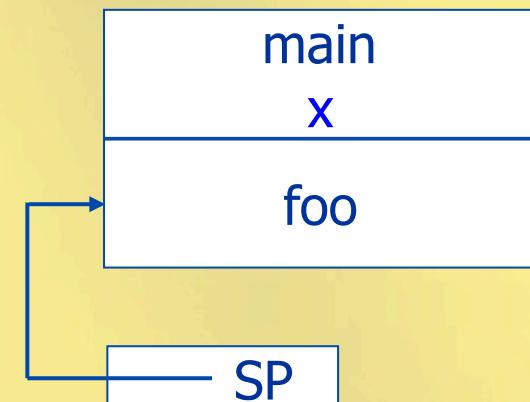
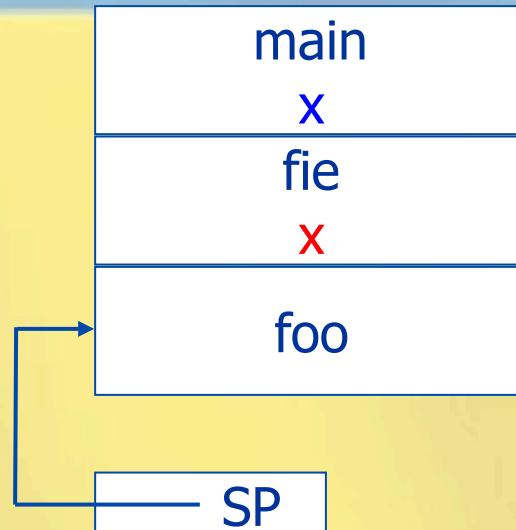
# Come si determina il legame corretto?

- Il codice di `foo` deve accedere sempre alla stessa variabile `x`
- Tale `x` è memorizzato in un certo RdA (in questo caso in quello del `main`)
- In cima alla pila abbiamo il RdA di `foo` (perché `foo` è in esecuzione)

primo caso:  
foo chiamato dentro fie

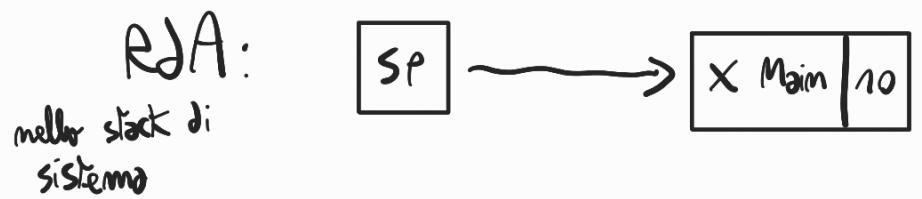
secondo caso:  
foo chiamato dal main

```
main {int x=10;  
void foo () {  
    x++;  
}  
void fie (){  
    int x=0;  
    foo();  
}  
fie();  
foo();  
}
```



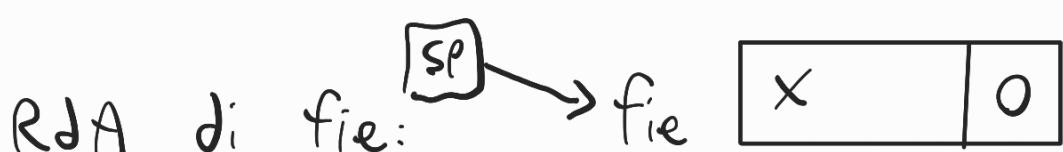
- Determina prima il corretto RdA dove trovare `x`
- Accedi a `x` tramite offset relativo a tale RdA (e non relativo a SP)

Simuliamo l'esecuzione del programma (Utile per gli es. d'esame)

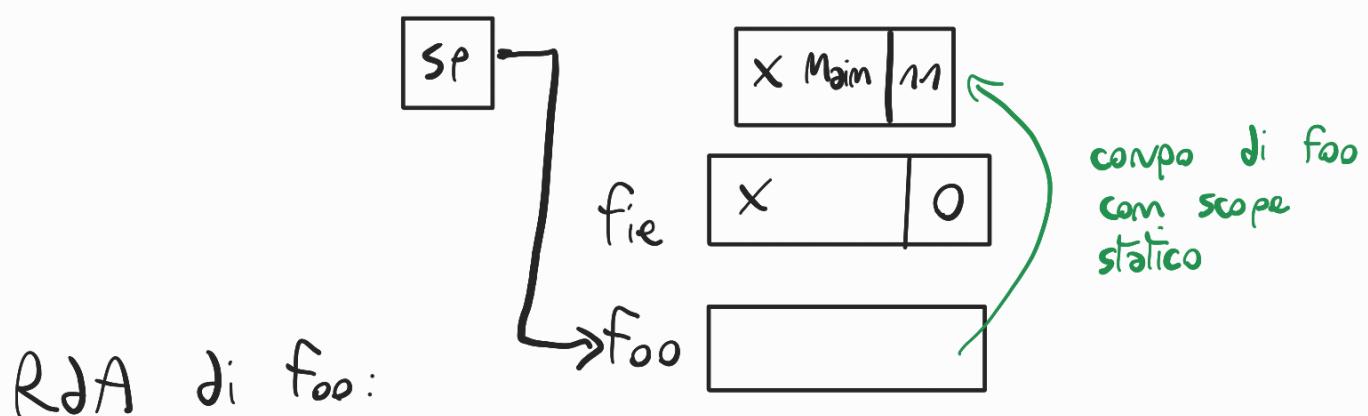


Trovo due dichiarazioni di funzione:  
(NON è una chiamata di funzione → Non alloco)  
(la memoria)

Trovo la chiamata di fie()



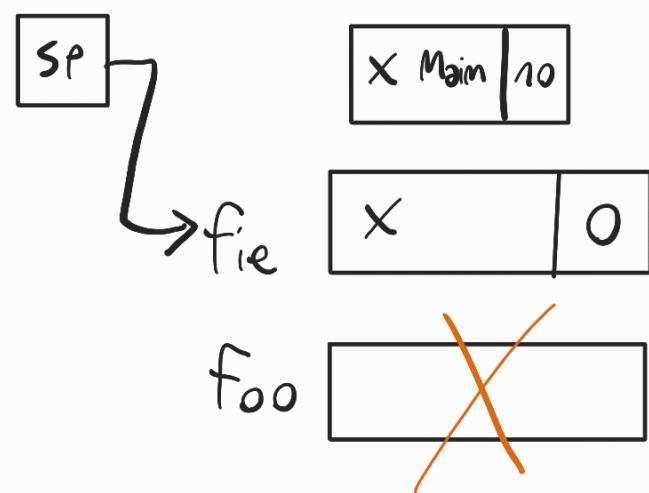
Trovo la chiamata di foo dentro fie



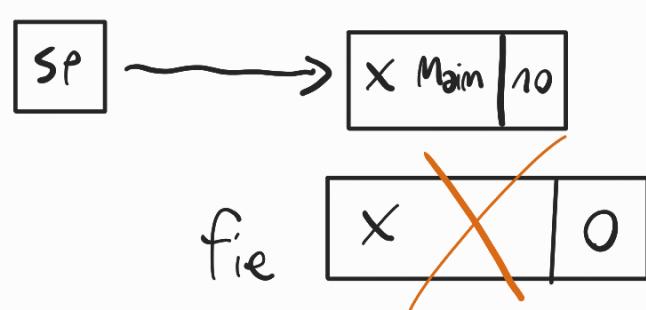
con la regola di scope statico:

→ la  $x++$  di foo è il valore di  $x$   
nel blocco che la racchiude è  
 $x = 10 \rightarrow x++ = 11$

foo termina



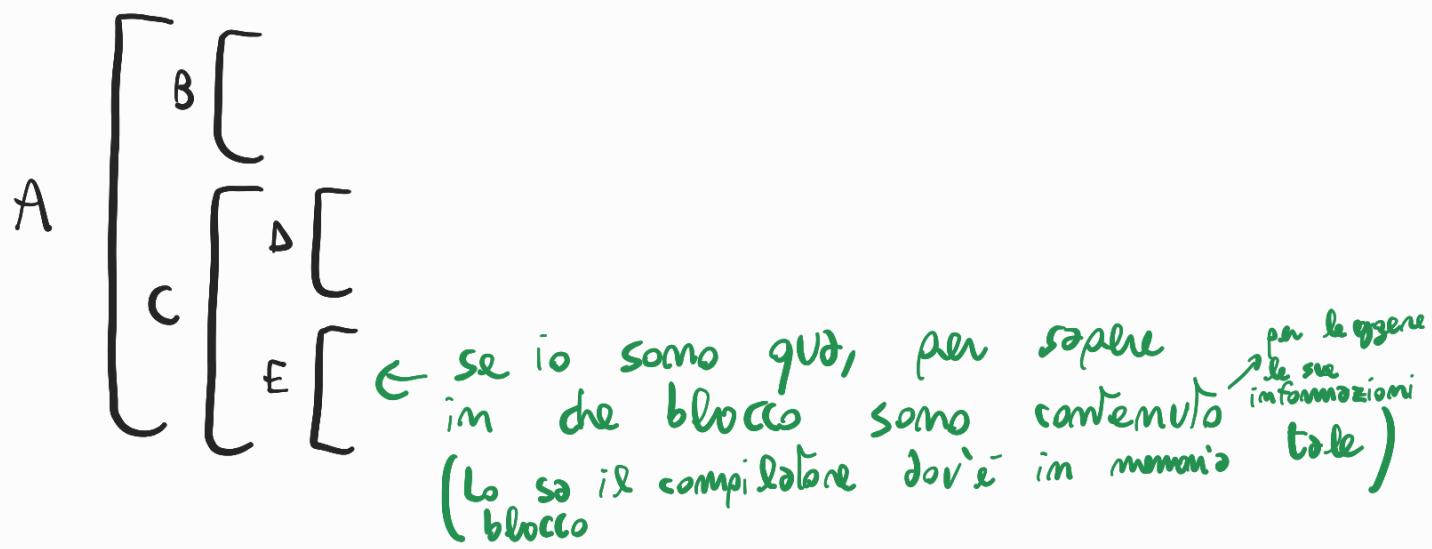
e anche fie termina



Domanda per lo scope statico?

Devo sapere qual'è il blocco che contiene quello in cui mi trovo  
Come faccio?

Siamo A, B, C, D, E blocchi:

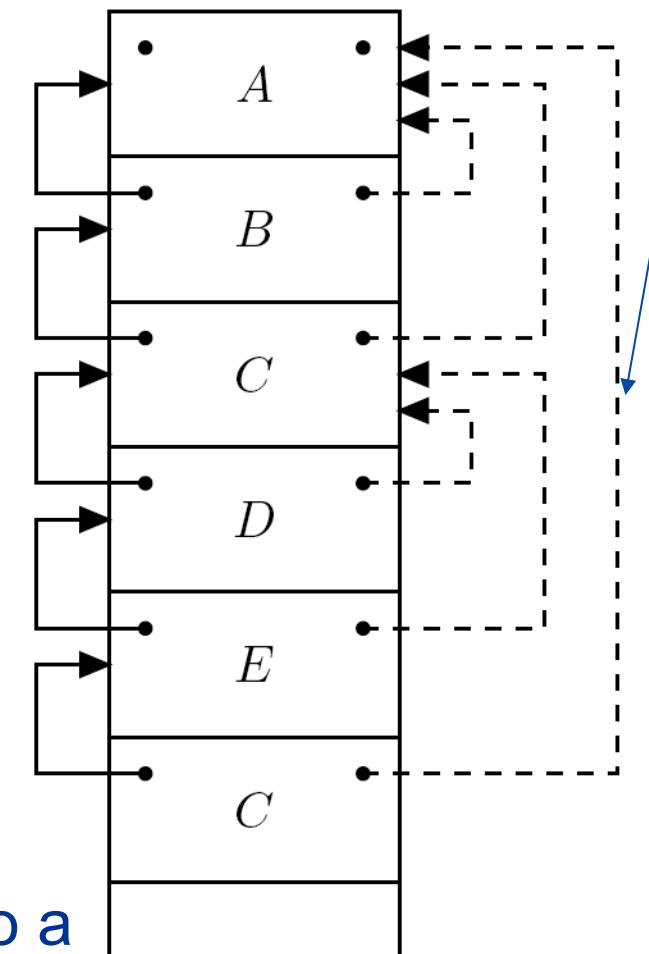
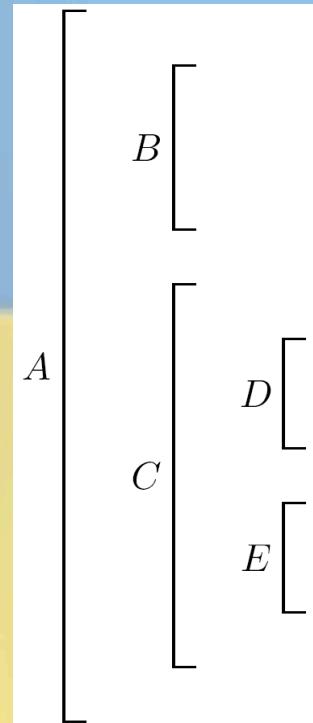


Ho un puntatore di catena statica che indica il Record di attivazione del blocco che contiene quello in cui sono.

(Invece il puntatore di catena dinamica punta all' RDA temporalmente precedente )

## Catena Statica: esempio

- Sequenza di chiamate a run time  
A, B, C, D, E, C



Se un sottoprogramma è annidato a livello  $k$ , allora la catena è lunga  $k$

Suppongo di essere in  $E$  e  $x$  è

dichiarato in  $A$

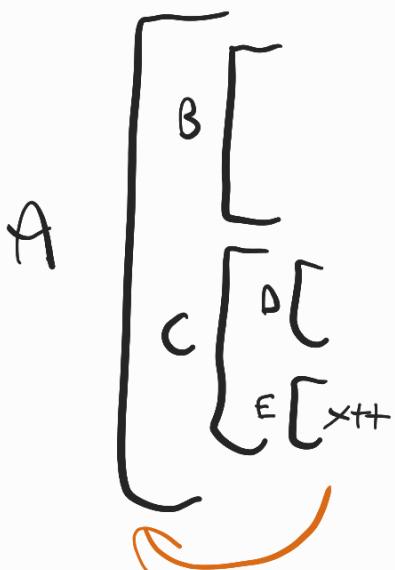
$E$



$x++$

Devo andare  
a prendere  
la  $x$  che sta  
in  $A$

Come faccio a sapere il valore  $x$   
che sta in  $A$ ?  $\rightarrow$  Uso i puntatori a catena statica



il compilatore sa che dopo 2 livelli  
c'è il valore di  $x$   
(in  $A$ )

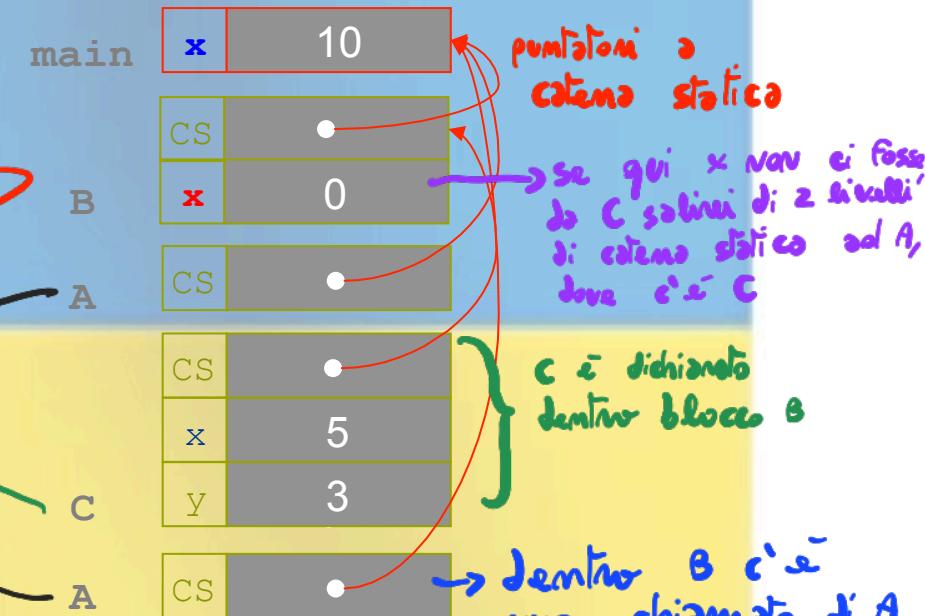
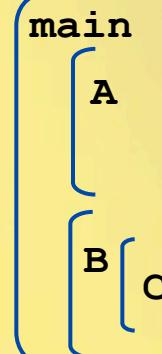
# Record di attivazione per scoping statico



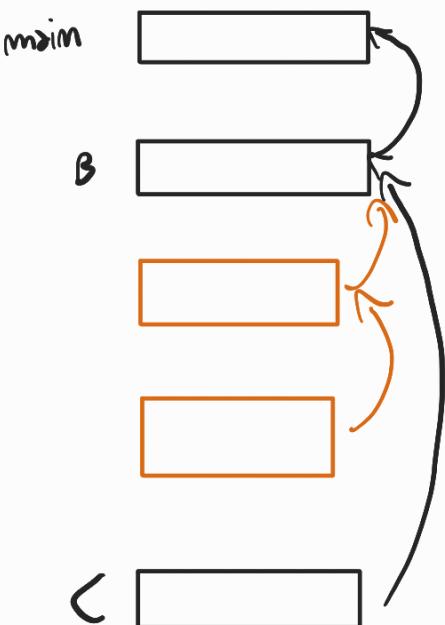
# Esempio

```

main {int x;
      void A(){
          x=x+1;
      }
      void B() {→ N.B. Se dichiam. si funz. non creano RDA, se chiamate sì
          int x;// dichiamaz. → NO RDA
          void C (int y) {
              int x; // x=0
              x=y+2; A() ↗
          }
          x=0; A(); C(3) ↘
      }
      x=10;
      B();
}
    
```



Perche' devo seguire la catena statica a run-time e non posso mettere un indirizzo corrispondente al RDA sullo stack staticamente?



Essendo allocati a run-time  
potrei avere più blocchi in un  
certo momento dell'esecuzione.  
Quindi staticamente NON posso  
stabilire il valore del  
puntatore di catena statica

# Dal punto di vista del supporto a run-time

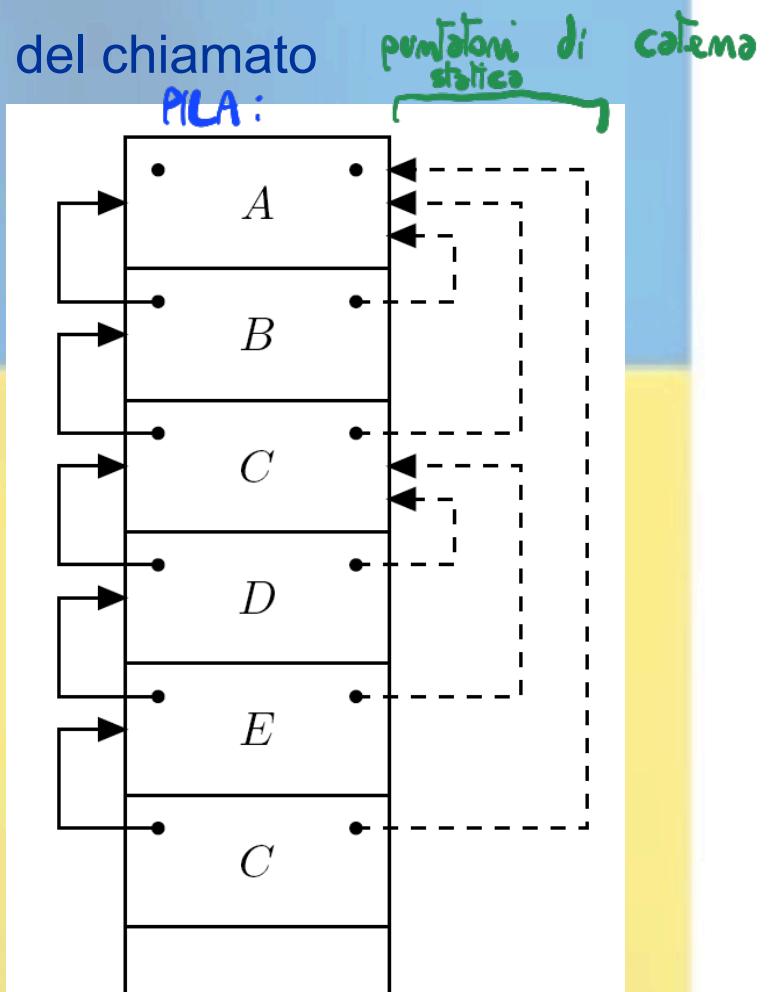
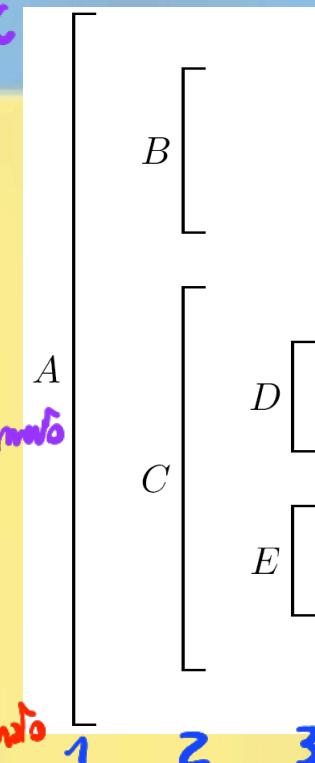
- Come viene determinato il link statico del chiamato?
- È il chiamante a determinare il link statico del chiamato
- Info a disposizione del chiamante:
  - annidamento statico dei blocchi (determinata dal compilatore)
  - proprio RdA

Siamo chiamate A, B, C, D, E, C

Come trovo da E il puntatore a cattedra statica di C?

→ Se io sono E chiamante,  
C chiamato, E sa il  
suo livello di annidamento  
rispetto a C

(Sono informazioni che  
fornisce il compilatore)  
→ distanza fra chiamante e chiamato



## Esercizio: PUNTATORI CATENA STATICA

7. Si consideri il seguente frammento di programma scritto in uno pseudo-linguaggio che usa scope statico.

```
{  
void f() {  
    void g() {  
        corpo_di_g;  
    }  
  
    void h() {  
        void l(){  
            corpo_di_l;  
        }  
        corpo_di_h;  
    }  
corpo_di_f;  
}
```

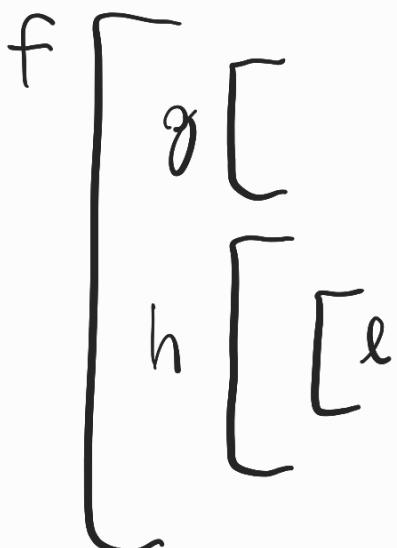
della catena statica

Si descriva graficamente l'evoluzione del display nella sequenza di chiamate f, g, h, l, g, supponendo che tutte le chiamate rimangano attive (ossia nessuna funzione ha restituito il controllo).

## SVOLGIMENTO:

Sequenza di chiamate: f, g, h, l, g

Codice:

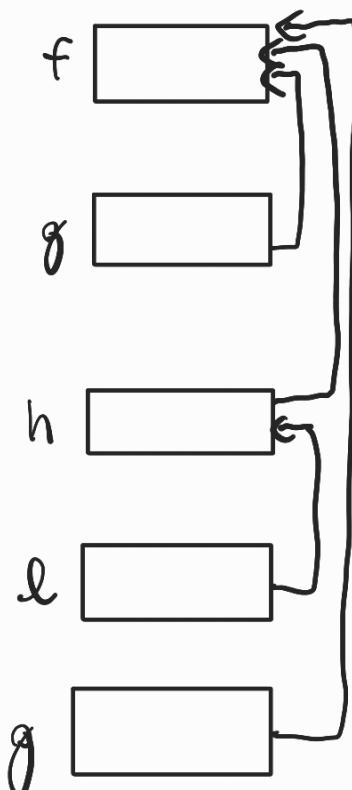


Si puo' fare (Aggiunto dal prof A CASO)



BLOCCHI:

PUNTATORI CATENA  
STATICA :



## Esercizio: DISPLAY

7. Si consideri il seguente frammento di programma scritto in uno pseudo-linguaggio che usa scope statico.

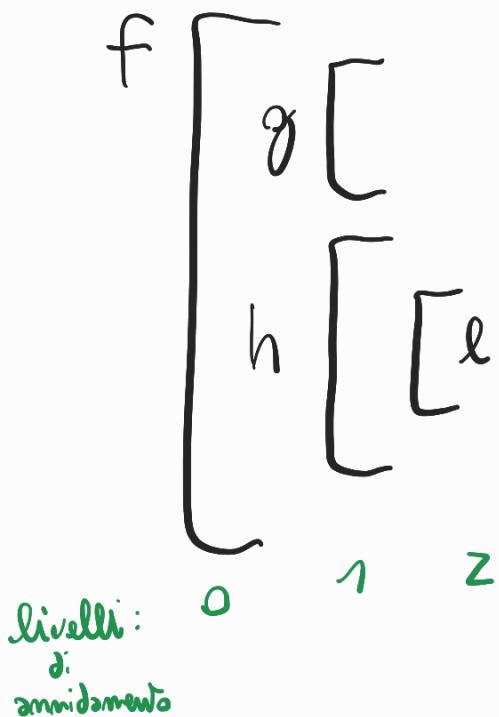
```
{  
void f() {  
    void g() {  
        corpo_di_g;  
    }  
  
    void h() {  
        void l(){  
            corpo_di_l;  
        }  
        corpo_di_h;  
    }  
corpo_di_f;  
}
```

Si descriva graficamente l'evoluzione del display nella sequenza di chiamate f, g, h, l, g, supponendo che tutte le chiamate rimangano attive (ossia nessuna funzione ha restituito il controllo).

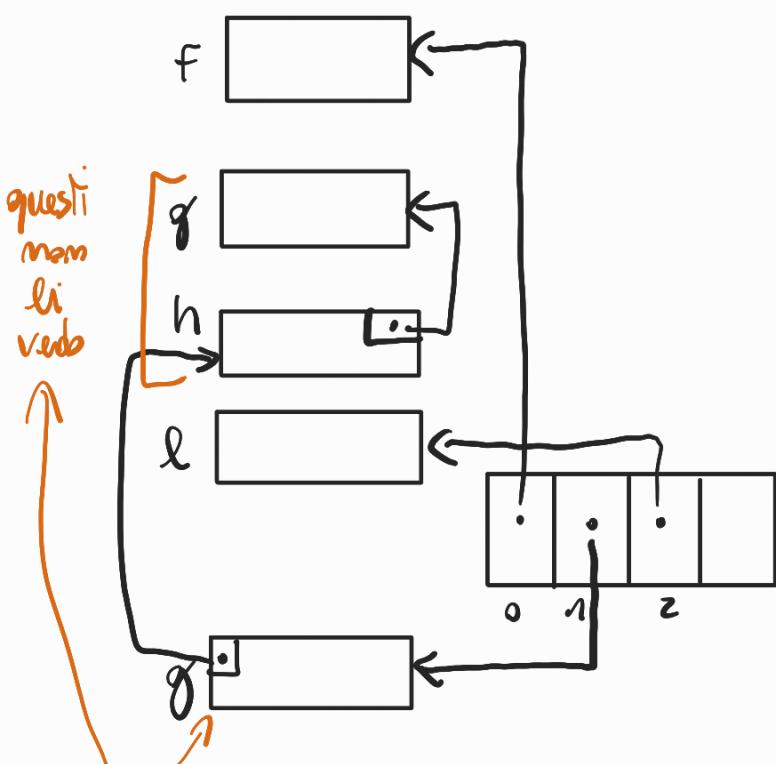
## SVOLGIMENTO:

Sequenza di chiamate: f, g, h, l, g

### Codice:



### DISPLAY:



Se sono in g  
→ l'evoluzione del display segue  
l'esecuzione

(valgono le regole di visibilità)

# Come determinare il puntatore di CS

Il chiamante Ch “conosce” l’annidamento dei blocchi:

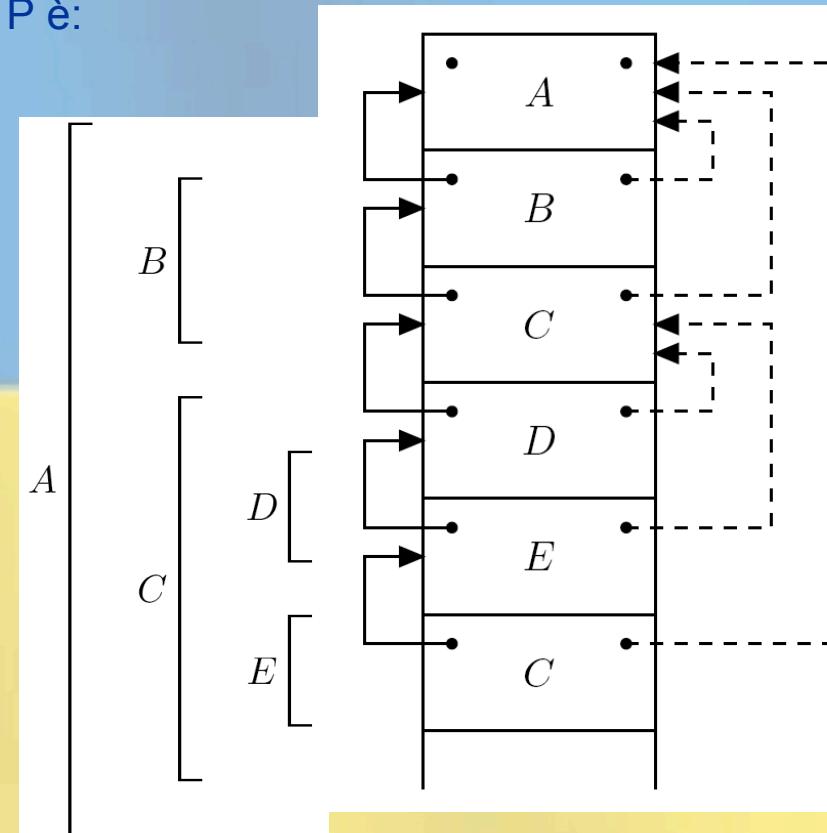
- quando Ch chiama P sa se la definizione di P è:
  - immediatamente inclusa in Ch ( $k=0$ );
  - in un blocco  $k$  passi fuori Ch
- nessun altro caso possibile:
  - perché P deve essere in scope!
- nel caso a destra:
  - chiamate: A, B, C, D, E, C
- con i dati di catena statica:
  - A; (B,0); (C,1); (D,0); (E,1); (C,2)

• Se  $k=0$ :

- Ch passa a P il proprio SP

• Se  $k>0$ :

- Ch risale la propria catena statica di  $k$  passi e passa il puntatore così determinato



# Ripartizione dei compiti

- Compilatore:
  - associa l'informazione **k** ad ogni chiamata
  - associa ad ogni nome un indice **h**:
    - $h=0$ : nome locale
    - $h \neq 0$ : nome non locale definito  $h$  blocchi sopra
- Sequenza chiamata/prologo
  - risale la catena statica
  - inizializza il puntatore di catena statica
- Costi
  - per ogni chiamata
    - $k$  passi di catena statica
  - ad *ogni* accesso ad una variabile non locale
    - $h$  passi di catena statica in più rispetto all'accesso ad un locale

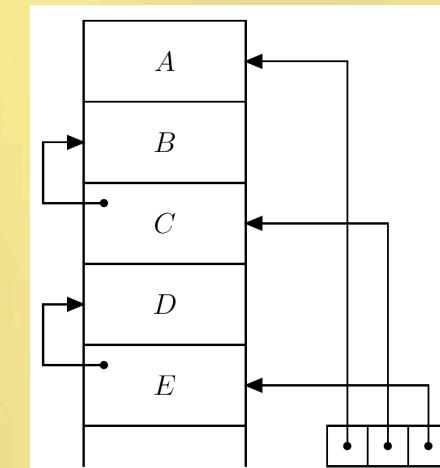
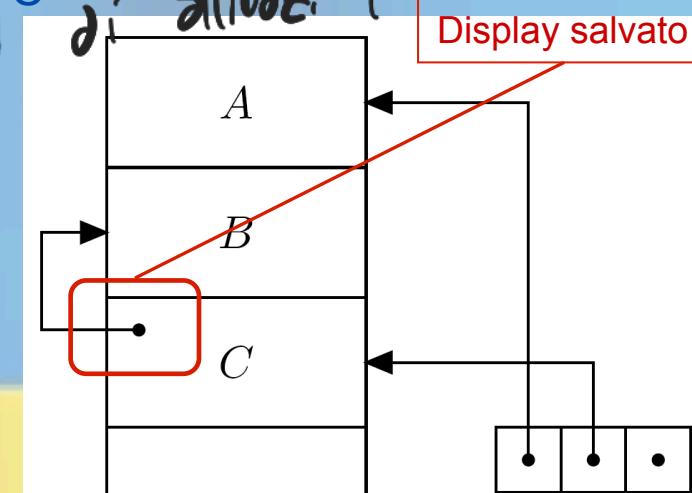
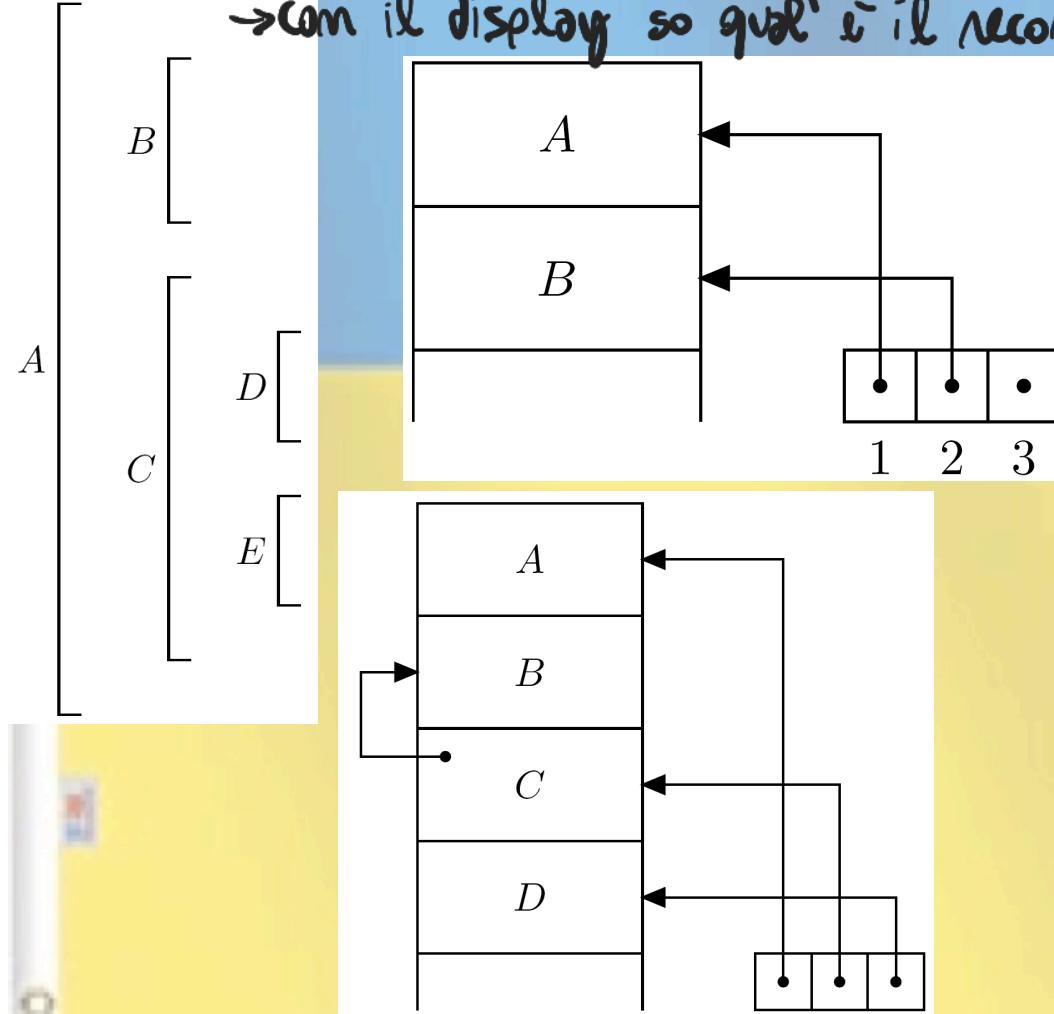
## Tentiamo di ridurre i costi: il *display*

- Si può ridurre il costo  $h$  ad una costante usando la tecnica del *display*:
  - la catena statica viene rappresentata mediante un array:
    - $i$ -esimo elemento dell'array = punt all'RdA del sottoprogramma di livello di annidamento  $i$ , attivo per ultimo
  - Dunque:
    - $Display[1]$ =RdA di una proc P di top level
    - $Display[2]$ =RdA di una proc Q dichiarata in P
    - ...
    - $Display[i]$ =RdA della proc attiva in questo momento (dichiarata dentro quella che si trova in  $Display[i-1]$ )
- Se il sottoprogramma corrente è annidato a livello  $i$ , un oggetto che è in uno scope esterno di  $h$  livelli può essere trovato guardando il punt a RdA nel *display* alla posizione  $j = i - h$

**Display** → vettore che in posizione  $i$  ha il puntatore al record di attivazione di livello  $i$  che è stato creato più recentemente ed è attivo

- $\text{Display}[i] = \text{Punt RdA della proc di livello } i, \text{ attiva per ultimo}$
- Sequenza di chiamate: A, B, C, D, E, C

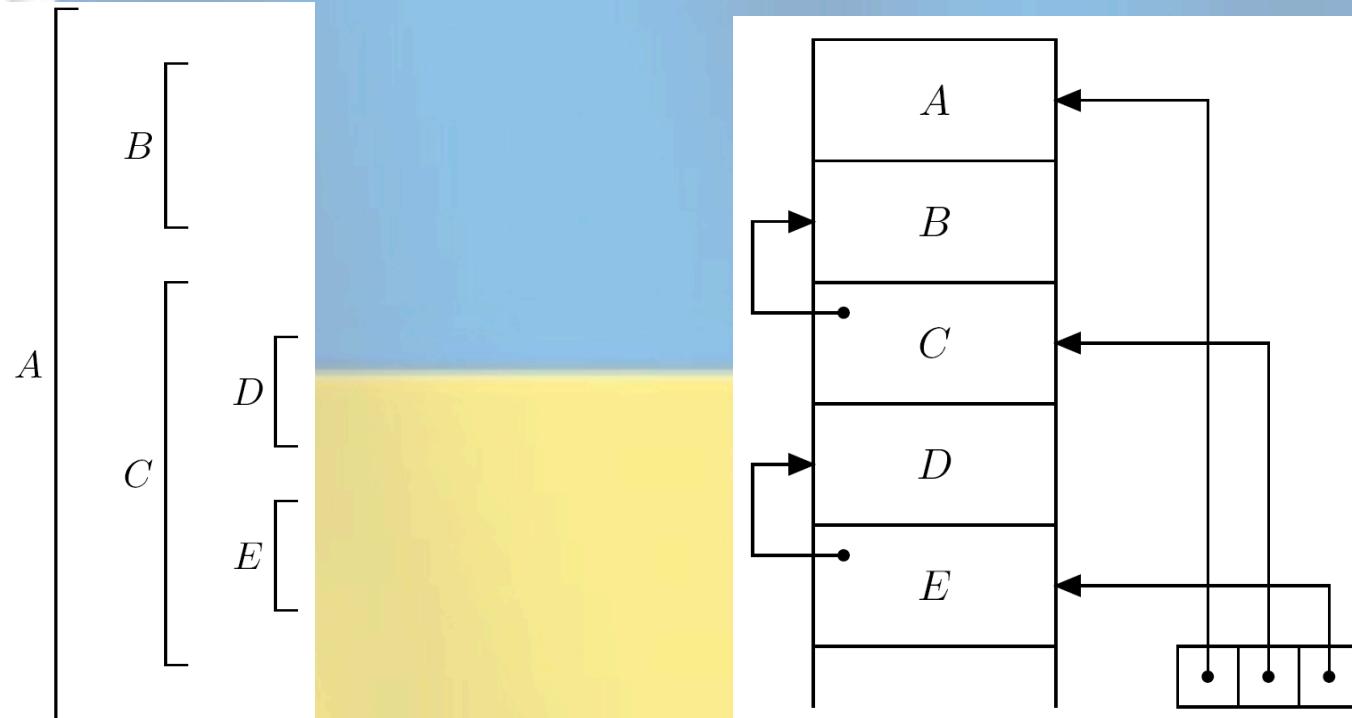
→ con il display so qual' è il record



di attivaz. più recente ed attivo  
ad un certo livello  
l'evito di seguire i puntatori di catena statica

# Display

- $\text{Display}[i]$  = Punt RdA della proc di livello  $i$ , attiva per ultimo
- Sequenza di chiamate: A, B, C, D, E, C



Se proc corrente annidata a livello  $i$ , lo scope esterno di  $h$  livelli si ottiene in  $\text{Display}[i - h]$

Con  $\text{Display}$  in memoria un oggetto è trovato con due accessi, uno per il display e uno per l'oggetto

## Come si determina il display

- È il chiamato a maneggiare il display.
  - Quando Ch chiama P a livello di annidamento  $j$ , P salva il valore di  $\text{Display}[j]$  nel proprio RdA e vi mette una copia del proprio (nuovo) punt a RdA.
- Funziona. Ragioniamo coi soliti due casi:
  - P dichiarata immediatamente in Ch ( $k=0$ );
  - P dichiarata in un blocco  $k$  passi fuori Ch
- Se  $k=0$ :
  - Ch e P condividono Display fino al livello corrente (che è  $j-1$ ). Mettendo il nuovo punt a RdA in  $\text{Display}[j]$ , il livello corrente viene esteso di 1 (il salvataggio potrebbe essere inutile, ma il chiamato non ha modo di saperlo: Ch potrebbe essere chiamato da Q, a sua volta a livello  $> j$  ).
- Se  $k>0$ :
  - Ch e P condividono Display fino al livello  $j-1$ .  $\text{Display}[j]$  deve essere modificato (dopo il salvataggio).

poco usato

## Display o catena statica ?

- Rari annidamenti di profondità > 3, quindi lunghezza max di catena statica = 3  
→ Se ho un programma con 100 livelli di annidamento, coi puntatori scendendo il display è comodo, ma poco usato perché in situazioni reali difficili avere più di 3 livelli di annidam.
  - Tecniche di ottimizzazione possono migliorare gli accessi alle catene usate più frequentemente (tenendo nei registri i puntatori)
- 
- Il display è più costoso da mantenere della catena statica nella sequenza di chiamata ...
  - Conclusione: display poco usato nelle implementazioni moderne...

## Scope dinamico

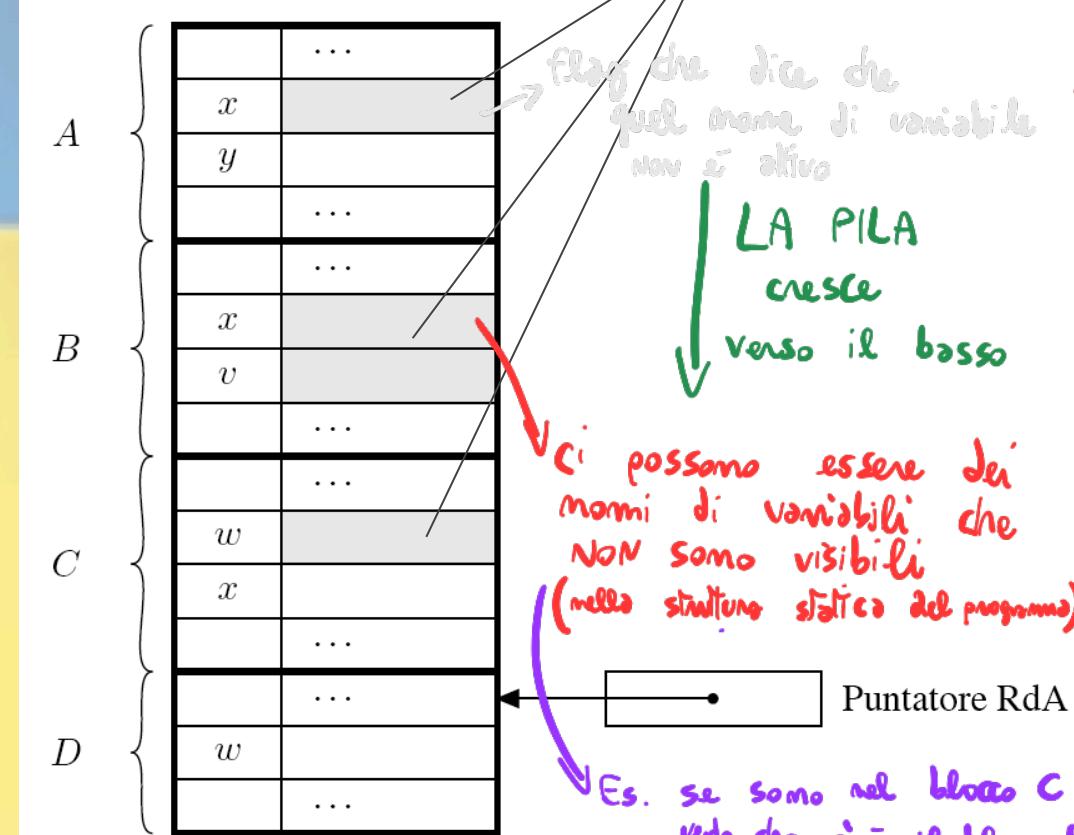
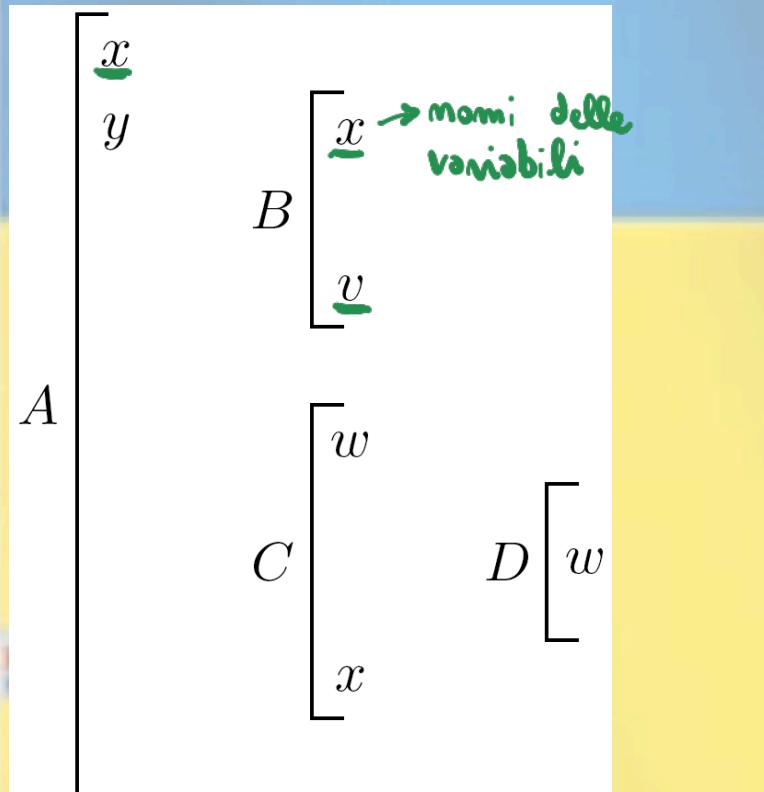
- Con scope dinamico l'associazione nomi-oggetti denotabili dipende
  - dal flusso del controllo a run-time
  - dall'ordine con cui i sottoprogrammi sono chiamati
- La regola generale è semplice: l'associazione corrente per un nome è quella determinata per ultima nell'esecuzione (non ancora distrutta).

# maif Implementazione ovvia

- Memorizzare i nomi negli RdA *della variabile*
- Ricerca per nome risalendo la pila
- Esempio: chiamate A,B,C,D

*La regola dice che  
dovrò trovare il primo nome di variabile attivo*

*in grigio associazioni  
non attive*

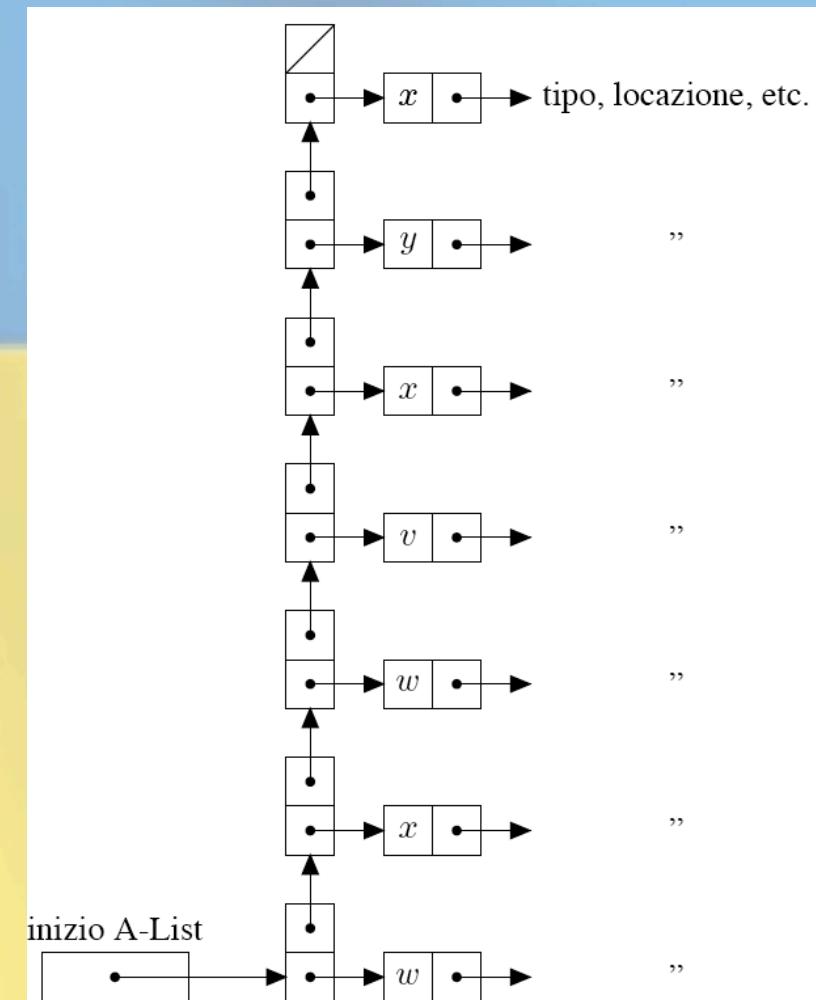
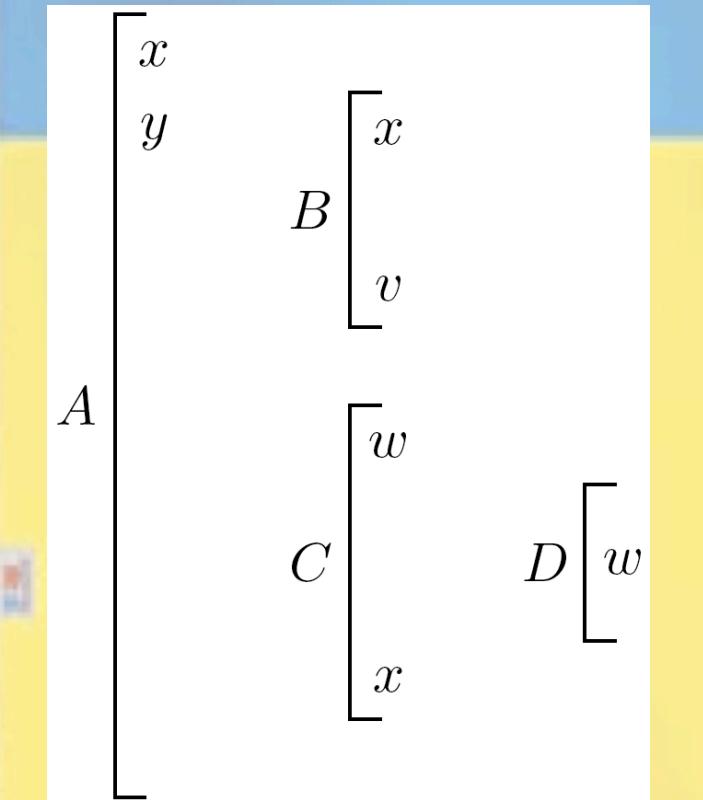


sta per "associazioni"

## Variante: A-list

struttura dati: miglioramento: seguono lo scope dinamico delle informazioni dello stack

- Le associazioni sono memorizzate in una struttura apposita, manipolata come una pila
- Esempio: chiamate A,B,C,D



## Costi delle A-list

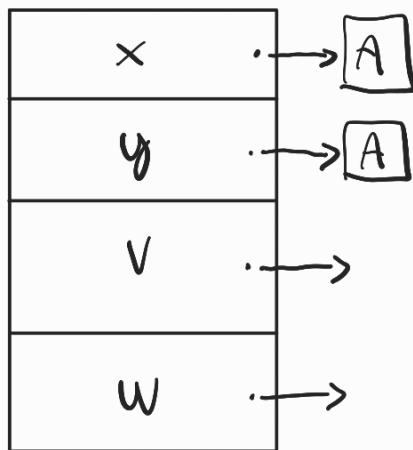
- Molto semplice da implementare
- Occupazione memoria:
  - nomi presenti esplicitamente
- Costo di gestione
  - ingresso/uscita da blocco
    - inserzione/rimozione di blocchi sulla pila
- Tempo di accesso
  - sempre lineare nella profondità della A-list  $O(n)$  com  
*→ devo cercare il nome della variabile che mi interessa nella A-list*
- Possiamo ridurre il tempo d'accesso medio, aumentando il tempo di ingresso/uscita da blocco...

## Tabella centrale dei riferimenti, CRT

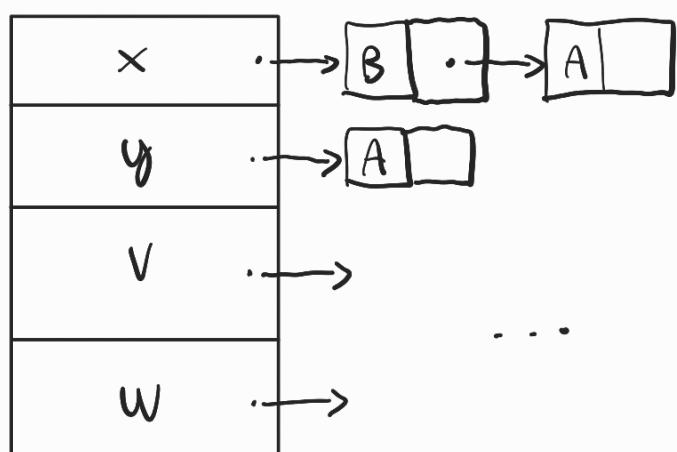
Idea: Memorizzare i nomi di variabili in cui è l'ultimo nome più recente

- Evita le lunghe scansioni delle A-list
- Una tabella mantiene tutti i nomi distinti del programma
  - se i nomi son noti staticamente, si può accedere all'elemento della tabella in tempo costante (accesso diretto)
  - altrimenti, accesso hash → *Se ci sono nomi di variabile dichiarati a run-time ci sono funzioni hash per accedervi*
- Ad ogni nome è associata la lista delle associazioni di quel nome
  - la più recente è la prima
  - le altre (disattivate) seguono
- Tempo di accesso costante

Inizialmente è vuota la CRT:



Quando trovo la dichiarazione di x in B, devo mascherare lo x dichiarato in A, quindi:



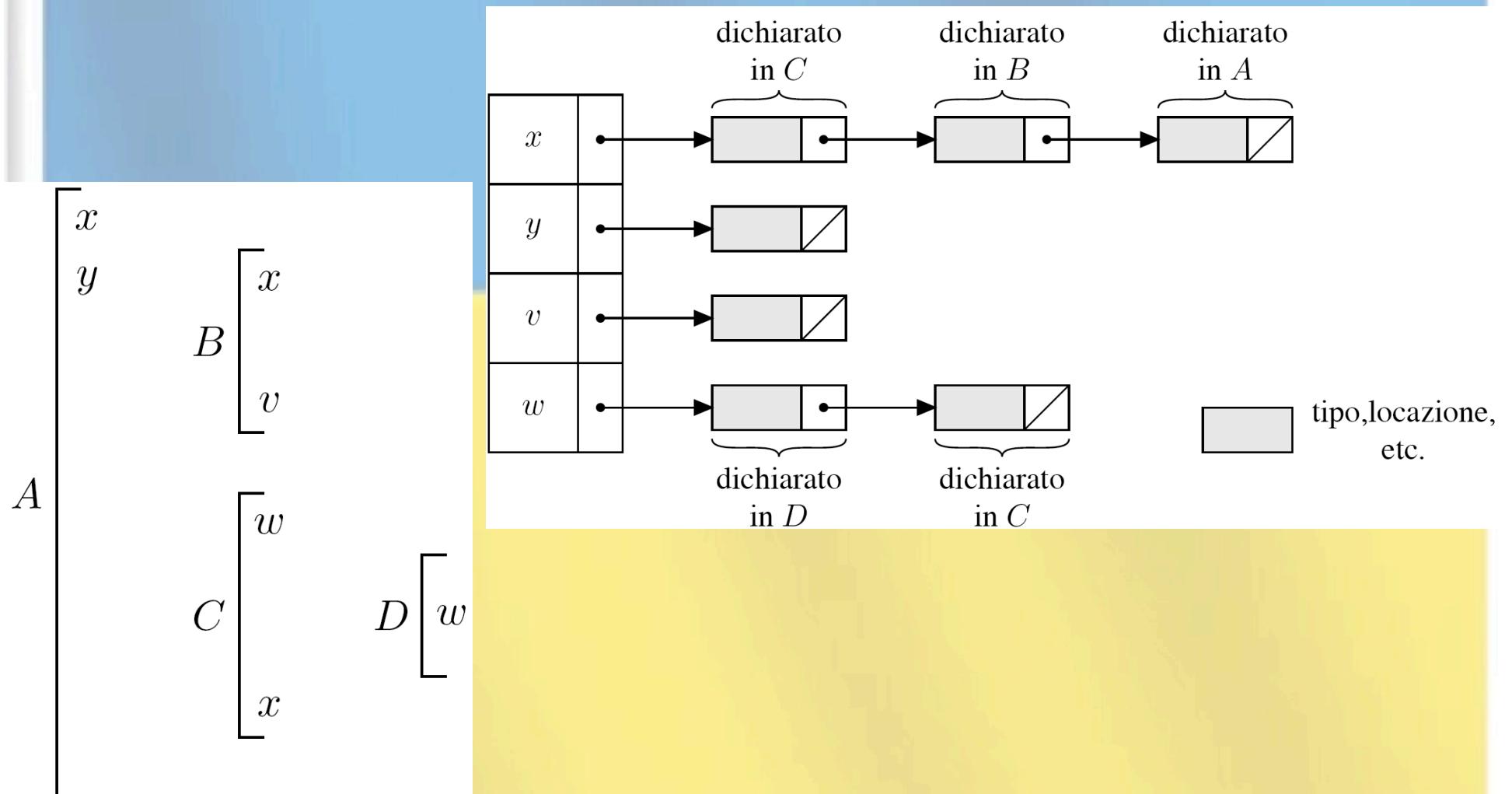
ecc...

→ Vantaggio: tempo di accesso costante  
(non devo più scorrere la lista)

→ Svantaggio: per ogni elemento aggiunto / tolto la gestione di questo tabella si complica

## Esempio (CRT)

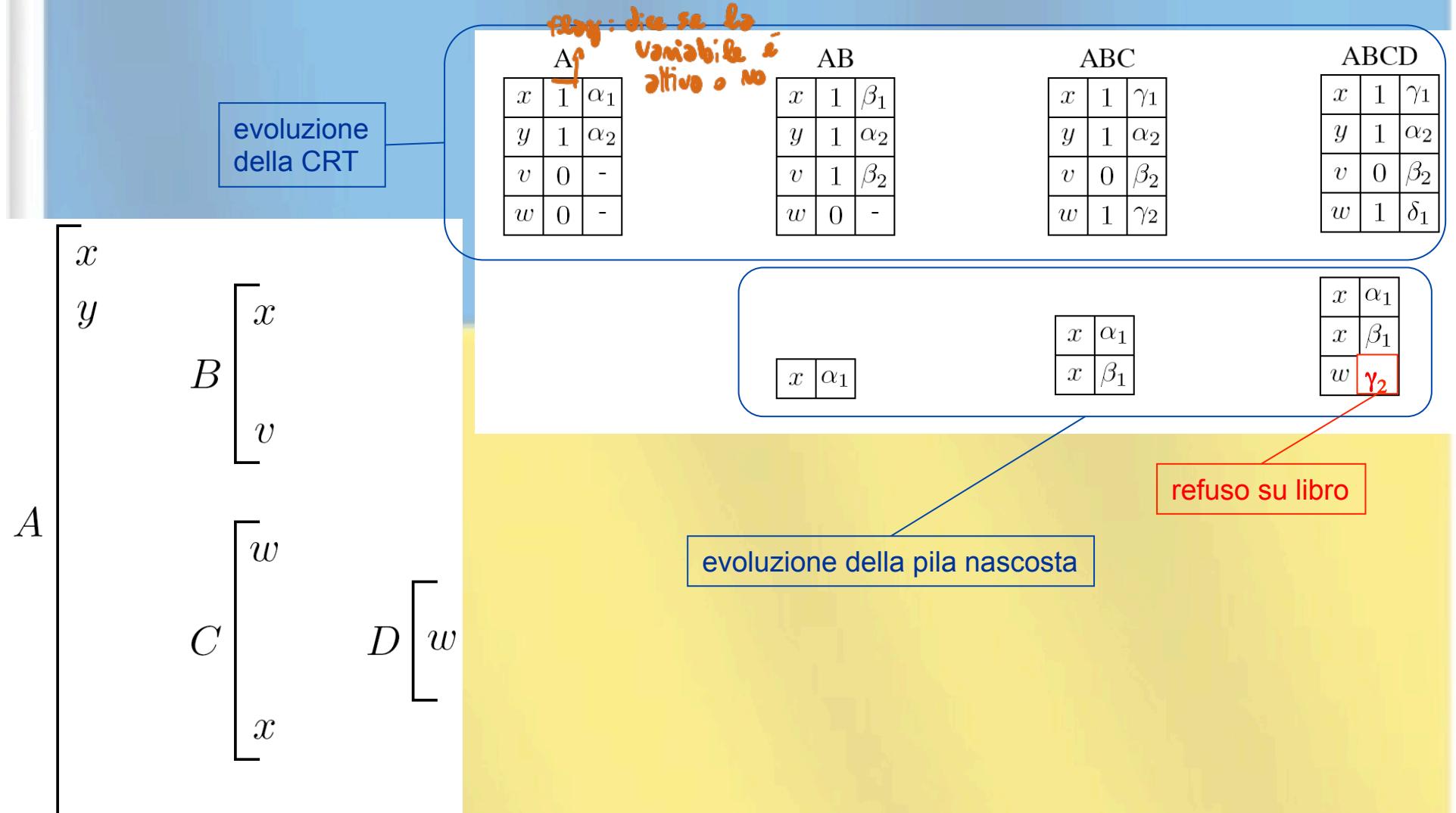
- Esempio: chiamate A,B,C,D



## CRT con pila nascosta

→ vantaggio implementativo: non ho più tante pile per ogni mem, ma una unica per tutti i nomi

- Esempio: chiamate A,B,C,D



## Costi della CRT

- Gestione più complessa di A-list
- Meno occupazione di memoria:
  - se nomi noti staticamente, i nomi non sono necessari
  - in ogni caso, ogni nome memorizzato una sola volta
- Costo di gestione
  - ingresso/uscita da blocco
    - manipolazione di tutte le liste dei nomi presenti nel blocco
- Tempo di accesso
  - costante (due accessi indiretti)
- Possiamo ridurre il tempo d' accesso medio, aumentando il tempo di ingresso/uscita da blocco...