Capitolo 7

La gestione della memoria

Tipi di allocazione della memoria

- Tre meccanismi di allocazione della memoria:
  - **statica**: memoria allocata a tempo di compilazione
  - **dinamica**: memoria allocata a tempo d'esecuzione
    - pila (stack):
      - oggetti allocati con politica LIFO
    - heap:
      - oggetti allocati e deallocati in qualsiasi momento

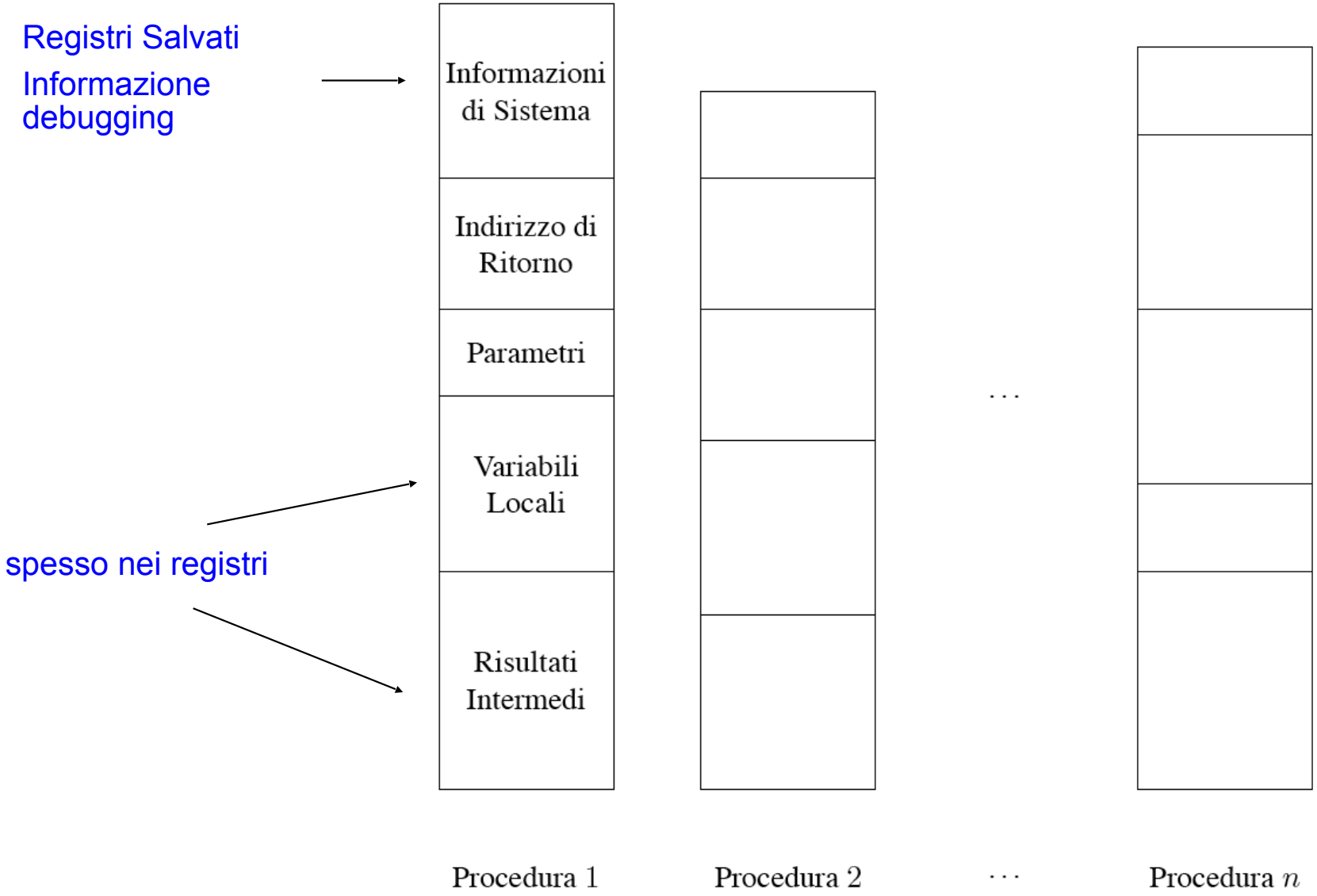
Allocazione statica

- 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

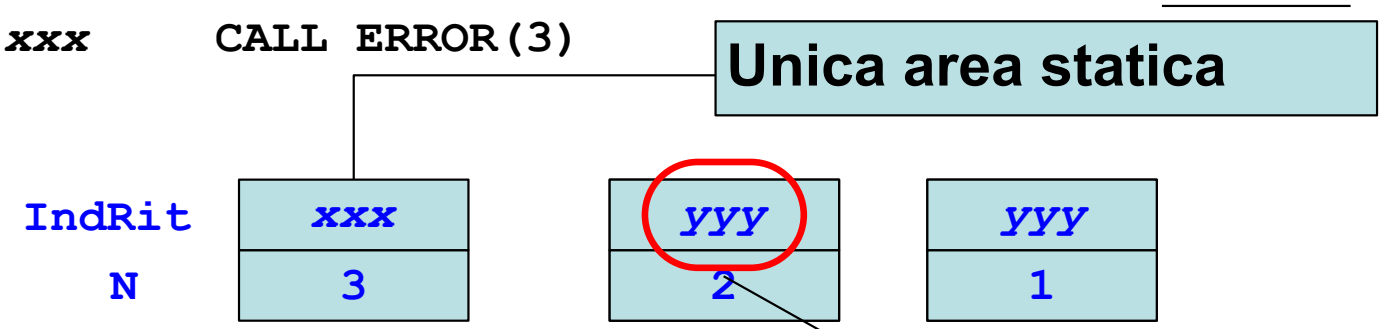


L'allocazione statica non permette ricorsione

FORTRAN: Programma sintatticamente illegale: ricorsione non ammessa

```
SUBROUTINE ERROR(N)
           IF (N.LE.1) RETURN
yyy        CALL ERROR(N-1)
           PRINT N
           END
           …
xxx        CALL ERROR(3)
```

Supponiamolo legale: eseguiamolo nel modello di memoria statica



OUTPUT

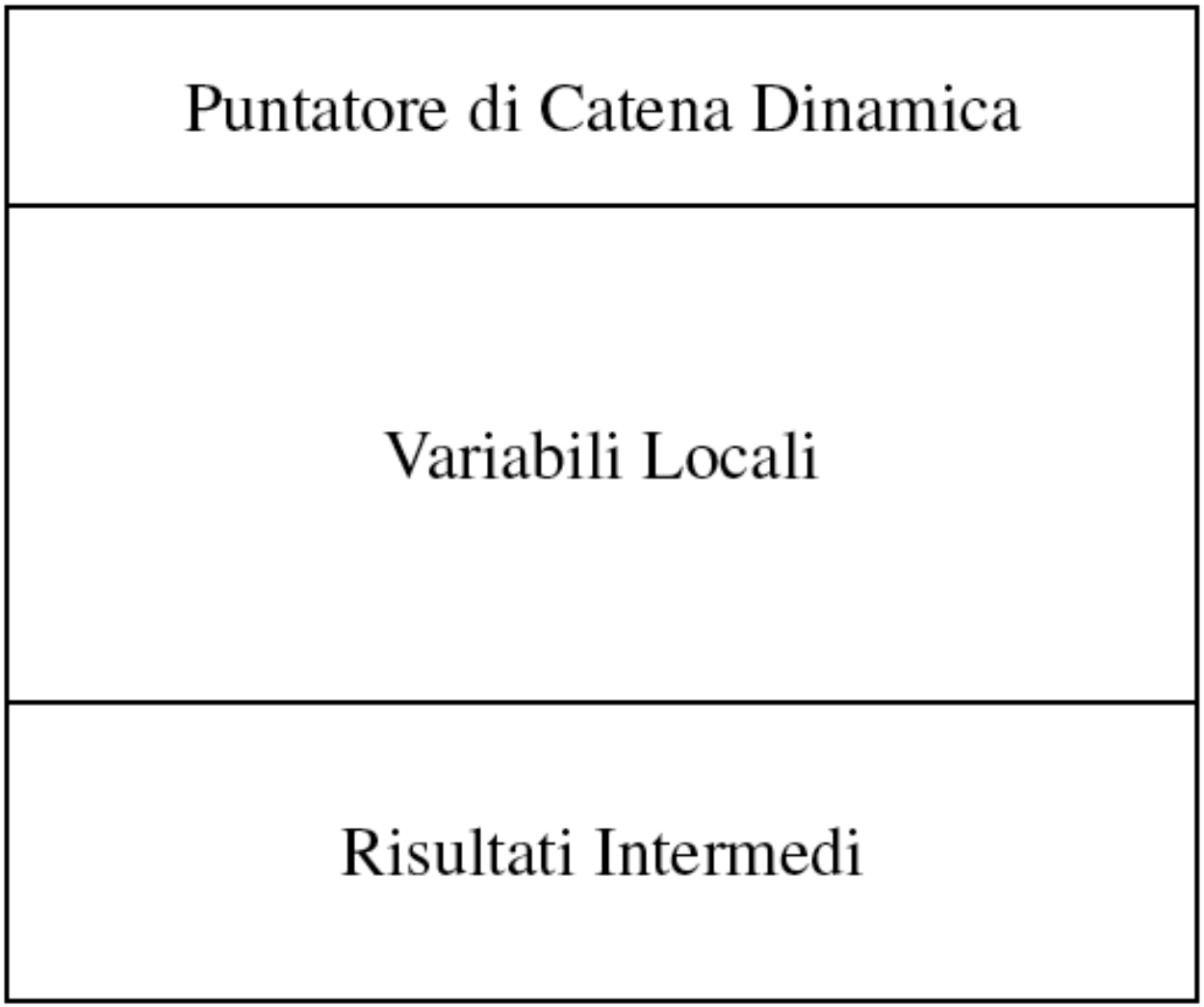
L'indirizzo di ritorno originale è perduto

1 1 1 1 1 1 …

Allocazione dinamica: pila

- Per ogni istanza di un sottoprogramma a run-time abbiamo un record di attivazione (RdA o frame) contente le informazioni relative a tale istanza
- Analogamente, ogni blocco ha un suo record di attivazione (più semplice)
- La Pila (LIFO) è la struttura dati naturale per gestire i RdA. Perché ?
- Anche in un linguaggio senza ricorsione può essere utile usare la pila per risparmiare memoria

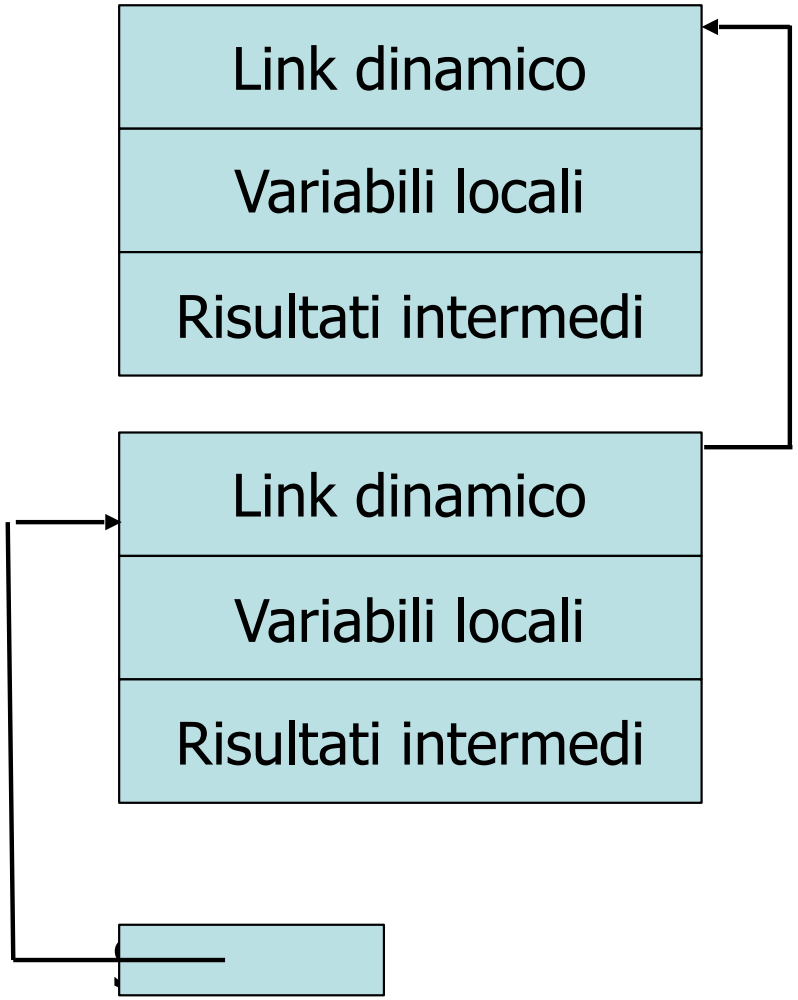
Record di attivazione per blocchi anonimi



Allocazione dinamica con pila

- La gestione della pila è compiuta mediante:
  - sequenza di chiamata
  - prologo
  - epilogo
  - sequenza di ritorno
- Indirizzo di un RdA non è noto a compile-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:
  - indirizzo-info = contenuto(SP)+offset
  - offset determinabile staticamente
  - Somma eseguita con unica istruzione macchina **load** o **store**

Record di attivazione per blocchi in-line



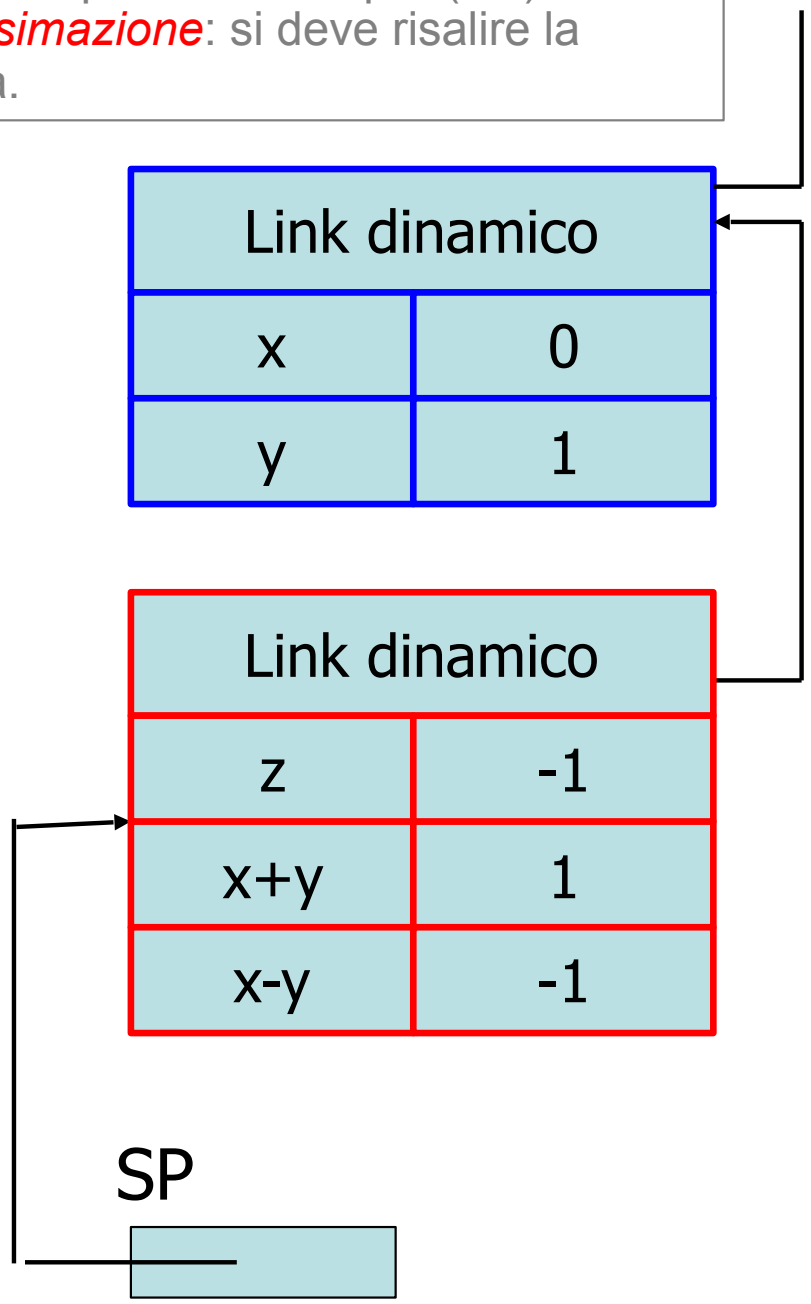
- Link dinamico (o control link)
  - puntatore al precedente record sullo stack
- Ingresso nel blocco: Push
  - link dinamico del nuovo Rda := SP
  - SP aggiornato a nuovo RdA
- Uscita dal blocco: Pop
  - Elimina RdA puntato da SP
  - SP := link dinamico del Rda tolto dallo stack

Esempio

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.

```
{ 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

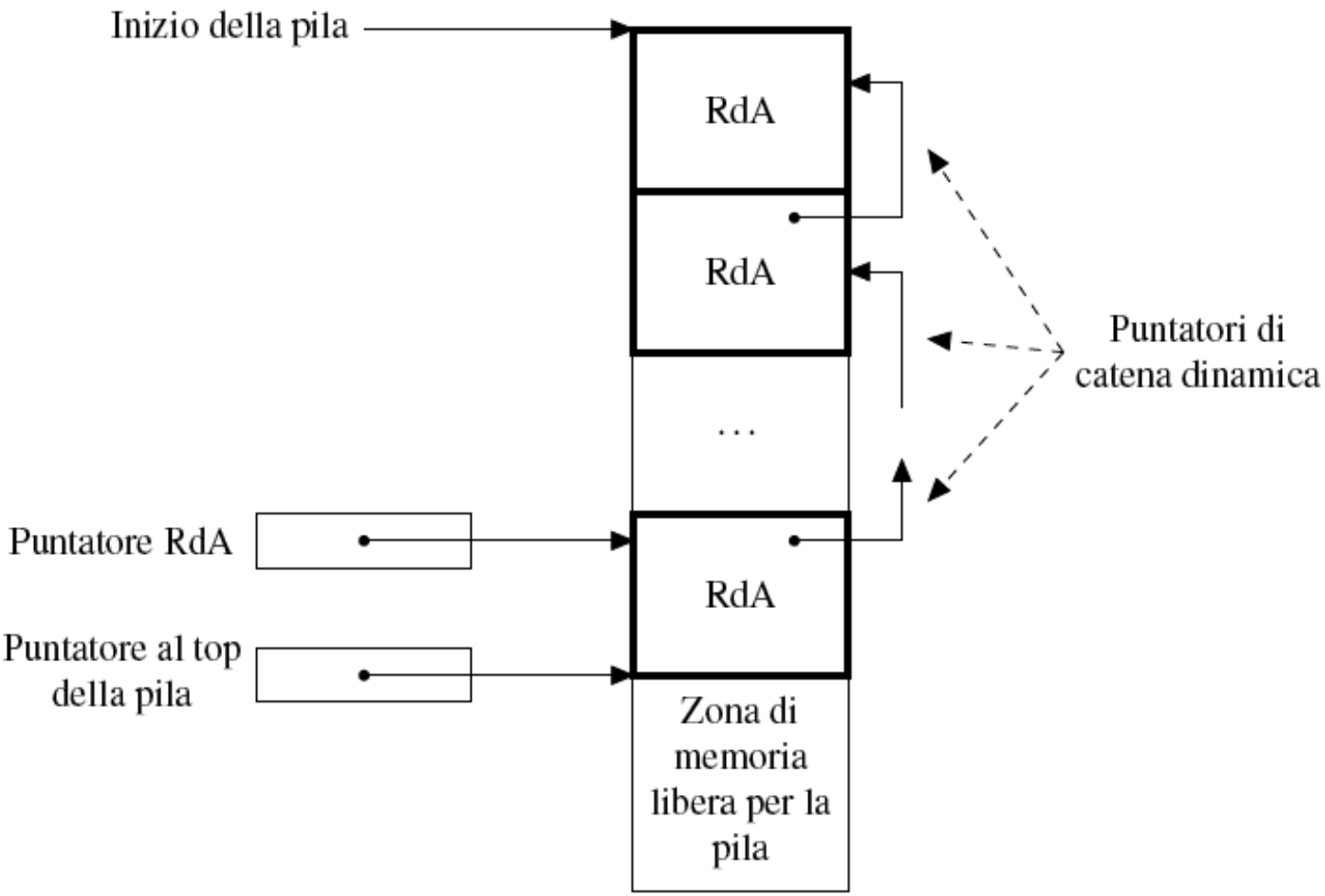


In realtà…

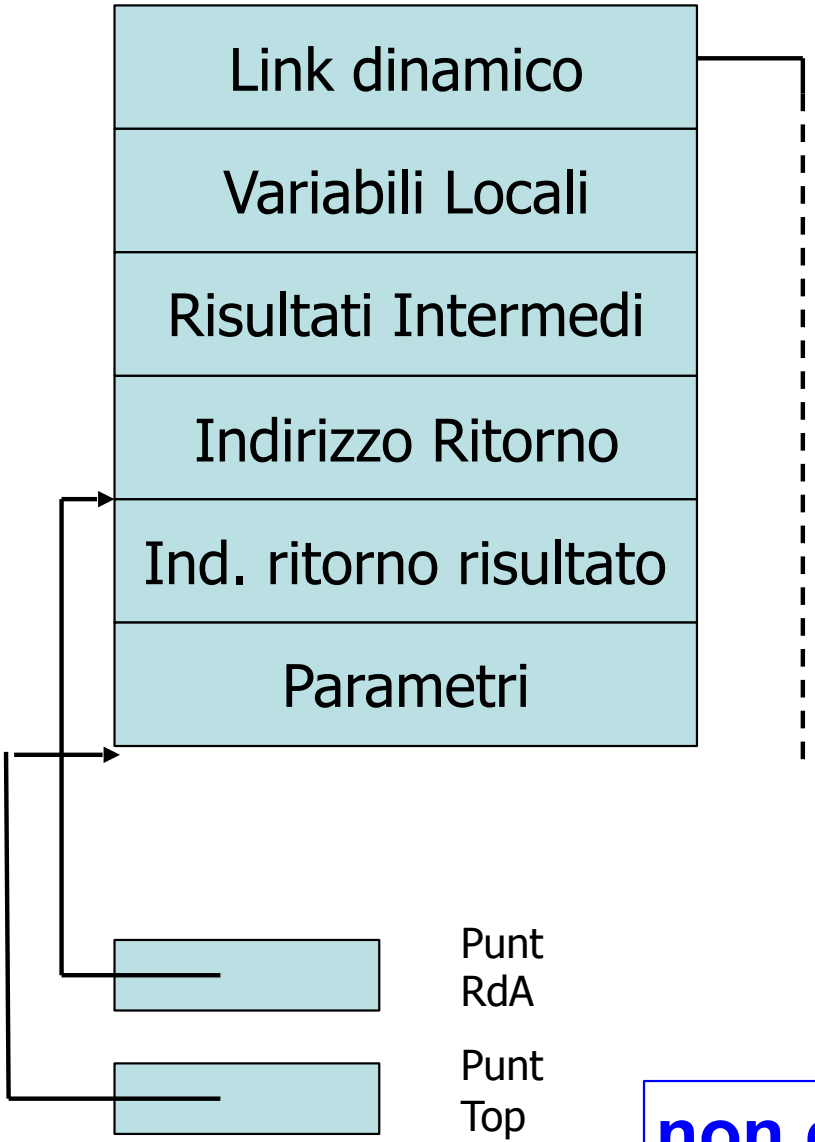
- In molti linguaggi non c'è manipolazione della pila per i blocchi anonimi
- Tutte le dichiarazioni dei blocchi annidati sono raccolte dal compilatore
- Allocazione di spazio per tutte
- Potenziale spreco di memoria, ma…
- Nessuna perdita di efficienza per la gestione della pila

Record di attivazione per procedure

Gestione della pila



Esempio

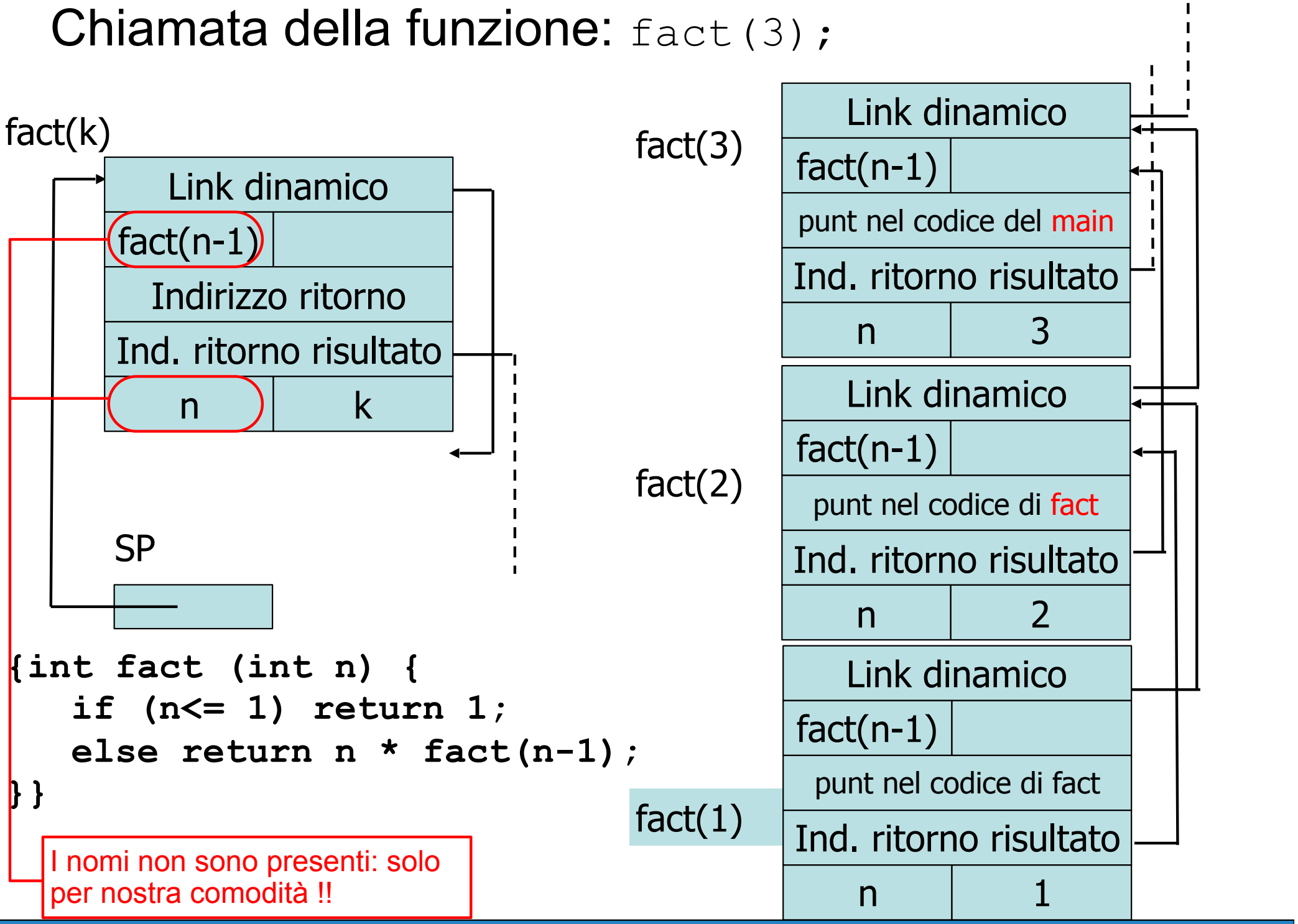



```
{int fact (int n) {
    if (n<= 1) return 1;
    else return n * fact(n-1);
 }}
```

- 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);`

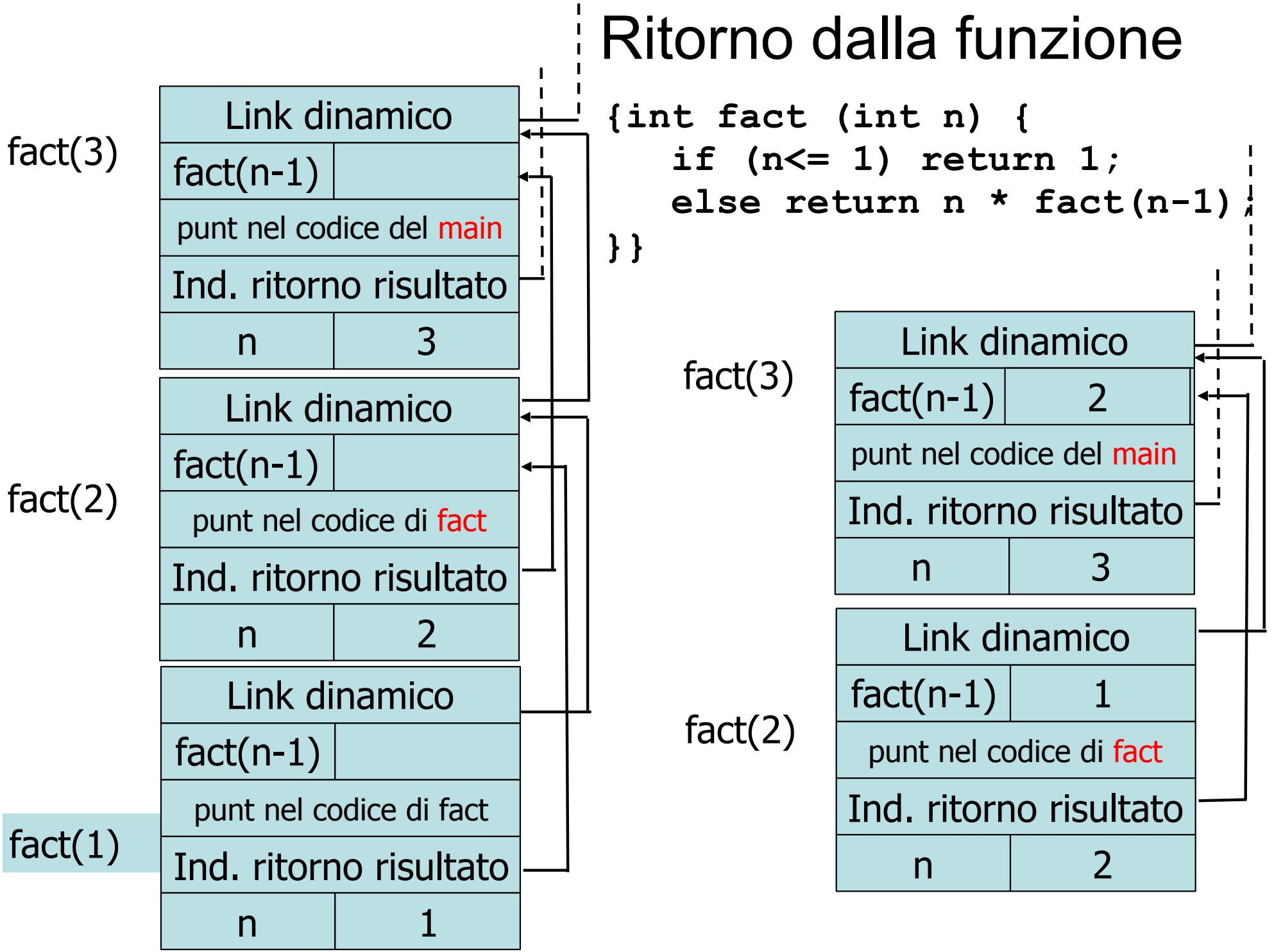



```
{int fact (int n) {
   if (n<= 1) return 1;
   else return n * fact(n-1);
}}
```

I nomi non sono presenti: solo per nostra comodità !!

Ritorno dalla funzione

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



Allocazione dinamica con heap

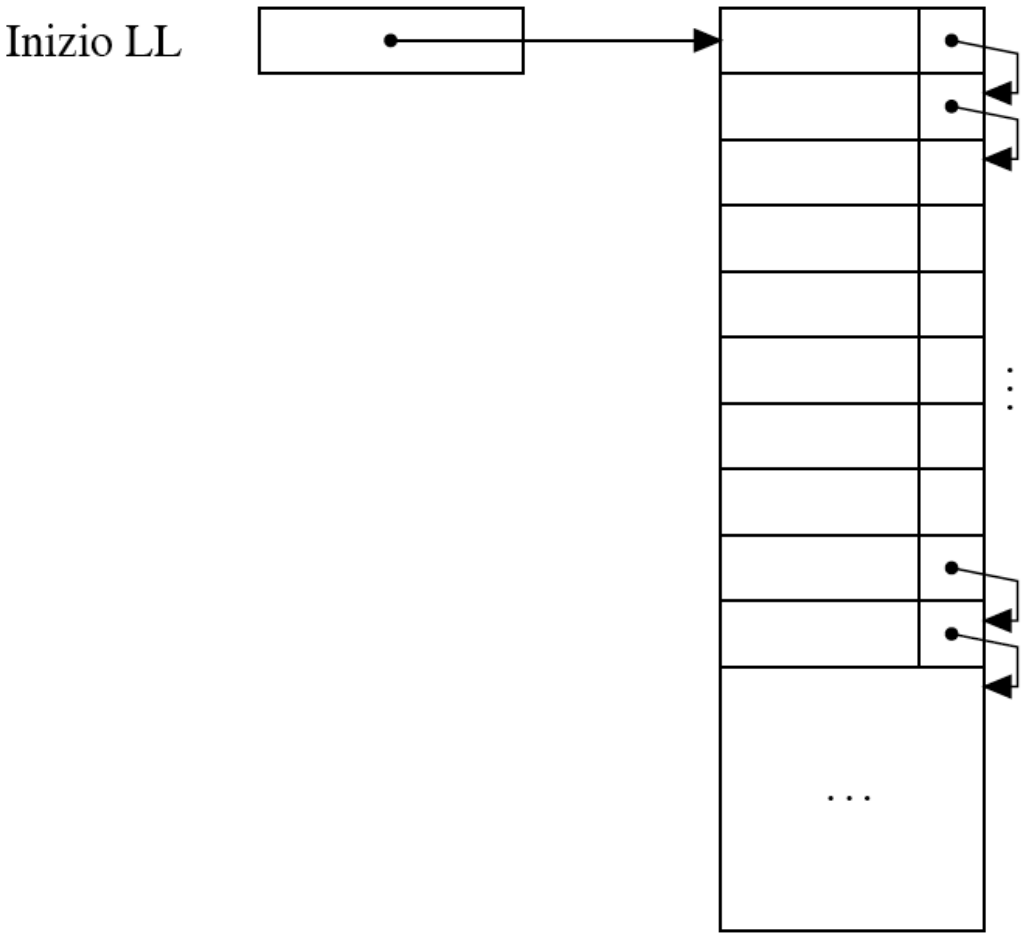
- Heap: regione di memoria i cui (sotto) blocchi possono essere allocati e deallocati in momenti arbitrari

- Necessario quando il linguaggio permette
  - allocazione esplicita di memoria a run-time
  - oggetti di dimensioni variabili
  - oggetti con vita non LIFO

- La gestione dello heap non è banale
  - gestione efficiente dello spazio: frammentazione
  - velocità di accesso

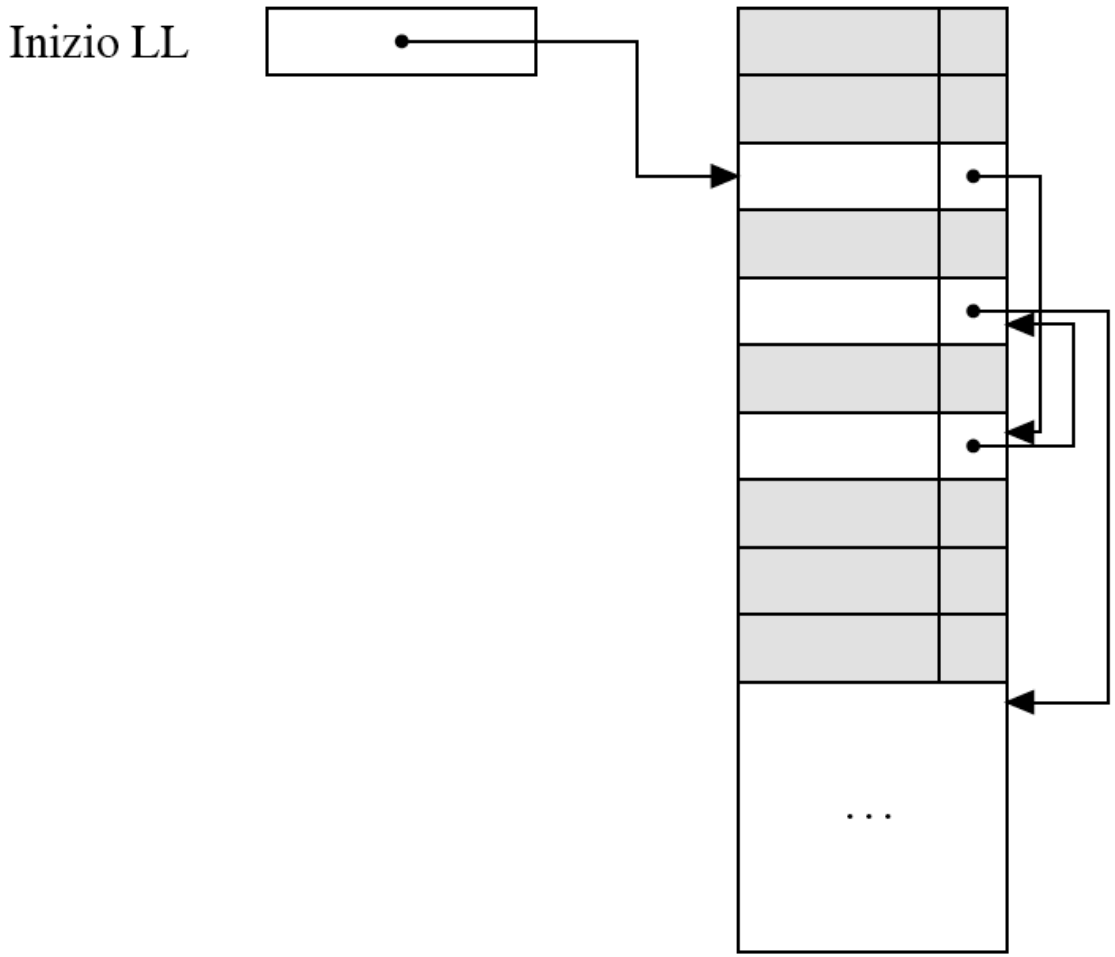
Heap: blocchi di dimensione fissa

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



Heap: blocchi di dimensione fissa

- Allocazione di uno o più blocchi contigui
- Deallocazione: restituzione alla lista libera



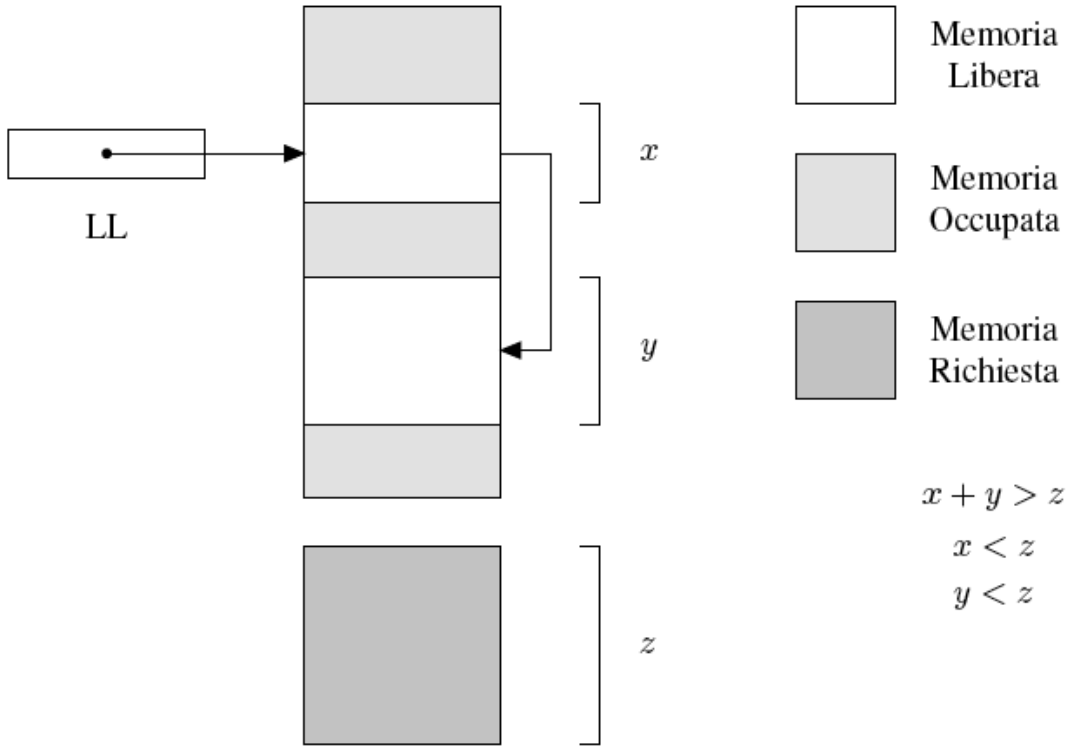
Heap: blocchi di dimensione variabile

- 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,
  - 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



Gestione della lista libera: unica lista

- Inizialmente un solo blocco, della dimensione dello heap
- Ad ogni richiesta di allocazione: cerca blocco di dimensione opportuna
  - first fit: primo blocco grande abbastanza
  - best fit: quello di dimensione più piccola, grande abbastanza
- Se il blocco scelto è molto più grande di quello che serve viene diviso in due e la parte inutilizzata è aggiunta alla LL
- 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).

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 lineare nel numero di blocchi liberi. Per migliorare liste libere multiple: La ripartizione dei blocchi fra le varie liste può essere
  - statica
  - dinamica:
    - Buddy system: k liste; la lista k ha blocchi di dimensione 2^k
      - » se richiesta allocazione per blocco di 2^k e 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*), se disponibile
      - »
    - Fibonacci simile, ma si usano numeri di Fibonacci

Implementazione delle regole di scope

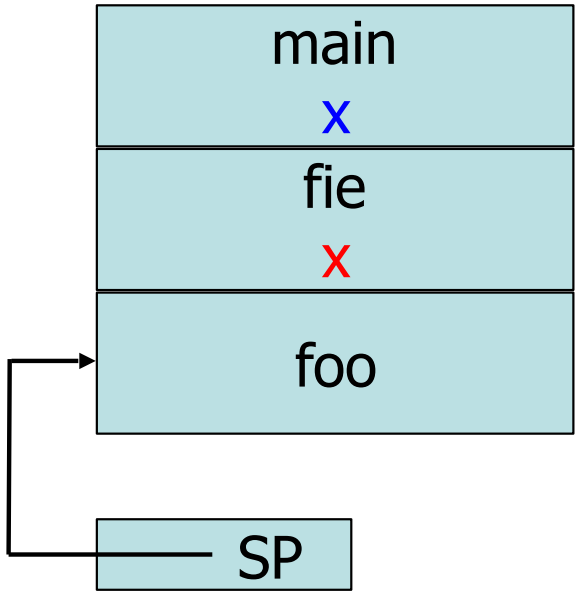
- Scope statico
  - catena statica
  - display
- Scope dinamico
  - A-list
  - Tabella centrale dell’ambiente (CRT)

Scope statico: come si determina il legame corretto?

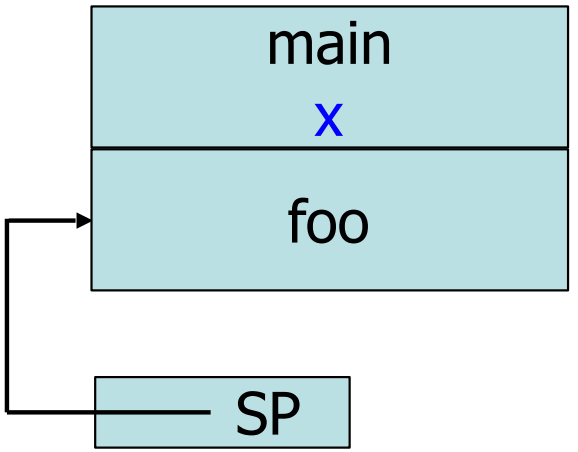
- `foo` deve accedere sempre alla stessa variabile x
- x è memorizzato in un certo RdA (qui *main*)
- In cima alla pila abbiamo il RdA di `foo`

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

primo caso:
foo chiamato dentro fie

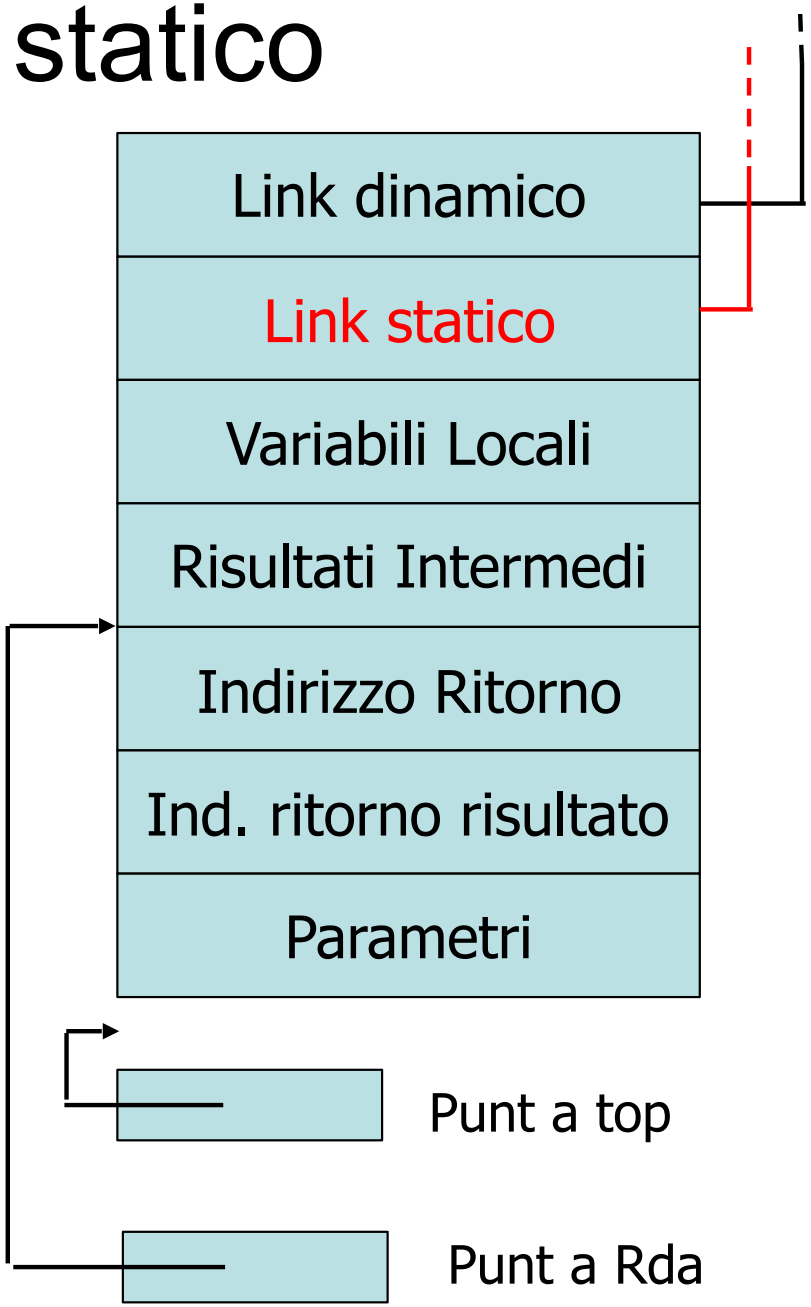


secondo caso:
foo chiamato dal main



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

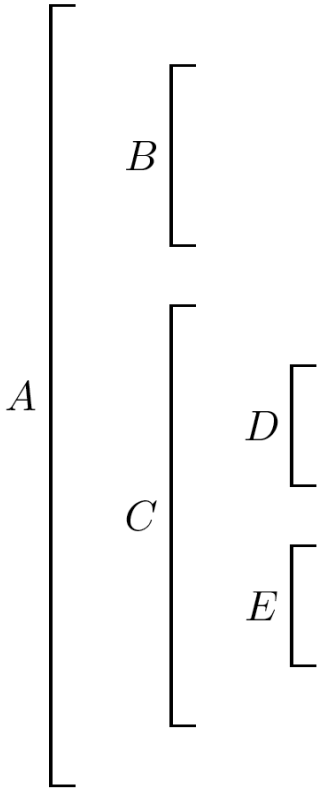
Record di attivazione per scoping statico

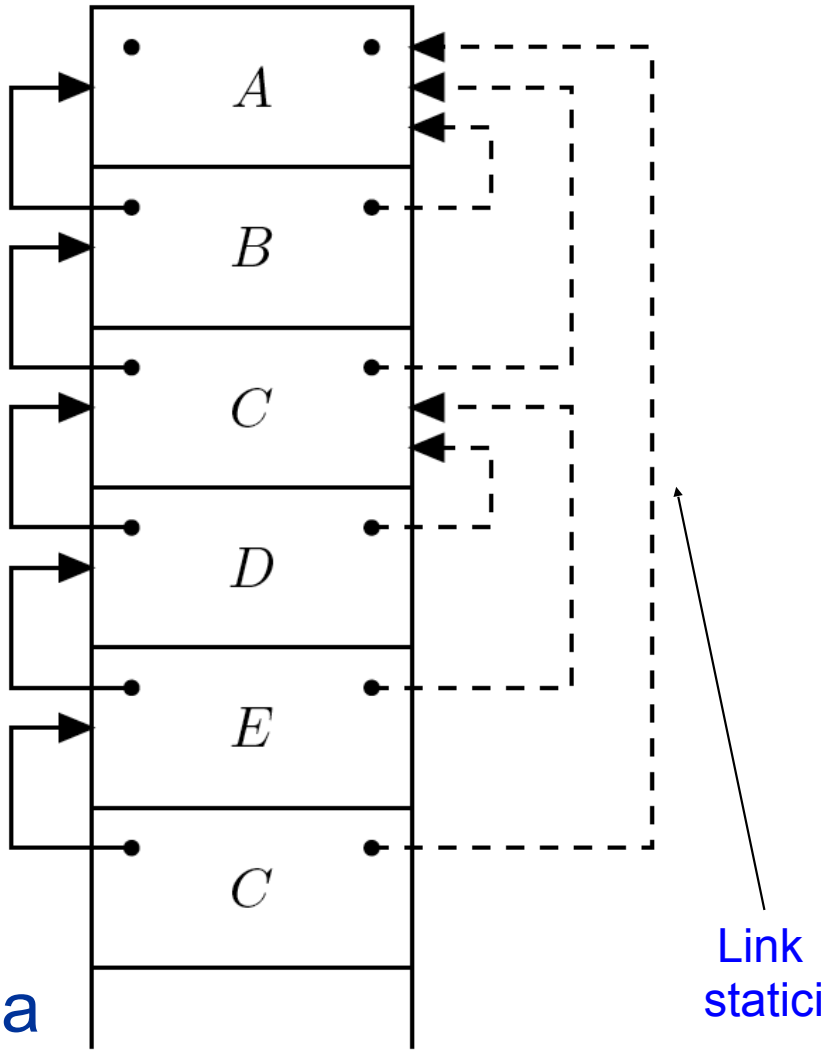


- Link dinamico:
  - (già visto)
- Link statico:
  - puntatore all'RdA del blocco che contiene immediatamente il testo del blocco in esecuzione
- Osserva:
  - link dinamico dipende dalla sequenza di esecuzione del programma
  - link statico dipende dall'annidamento statico (nel testo) delle dichiarazioni delle procedure

Catena Statica: esempio

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

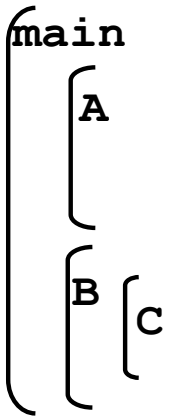


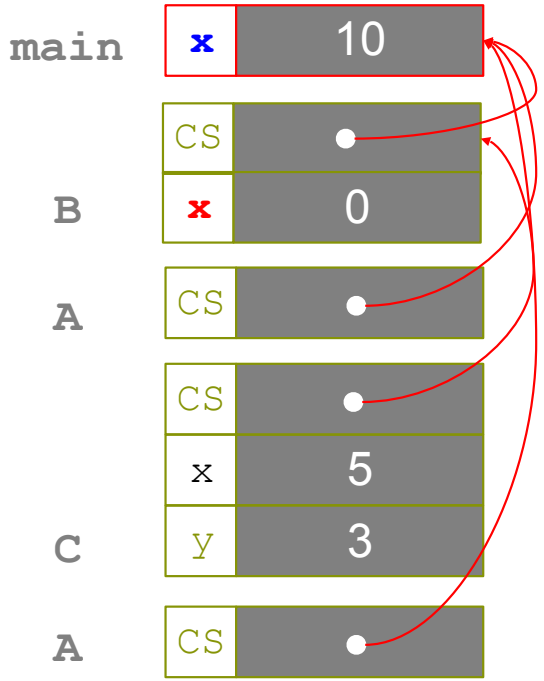


Se un sottoprogramma è annidato a livello *k*, allora la catena è lunga *k*

Esempio

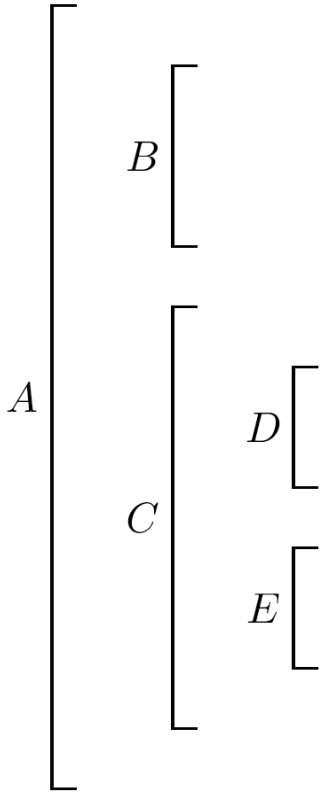
```
{int x;
 void A(){
    x=x+1;}
 void B(){
   int x;
   void C (int y){
      int x;
      x=y+2; A();
   }
   x=0; A(); C(3);
 }
 x=10;
 B();
}
```

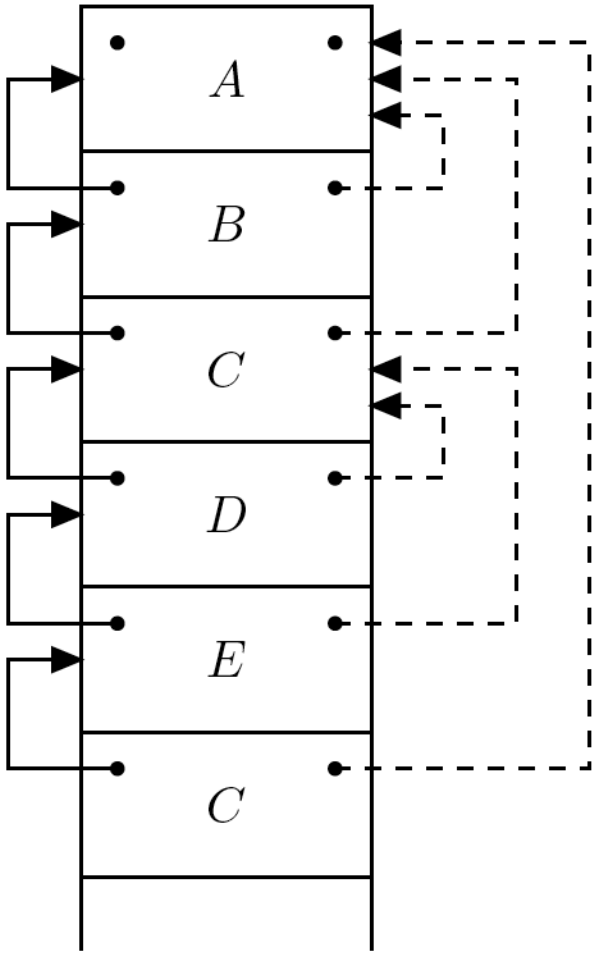




Dal punto di vista del supporto a run-time

- 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

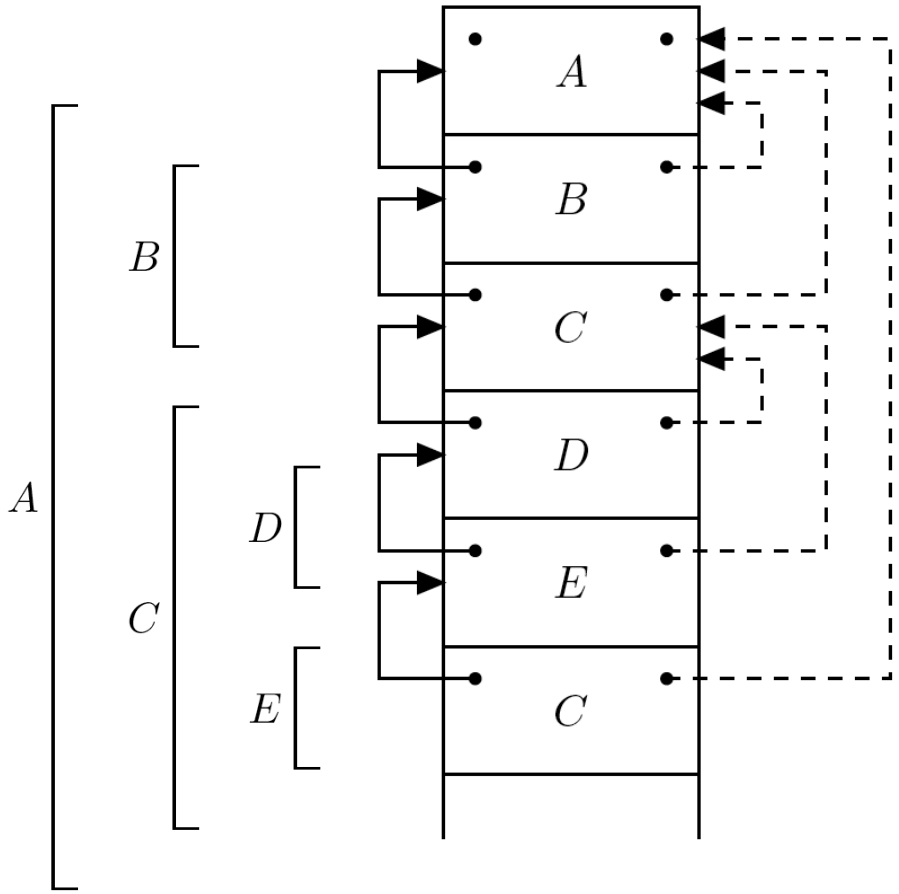




Come determinare il puntatore di CS

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)?
- 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 pass determinato

Tentiamo di ridurre i costi: il *display*

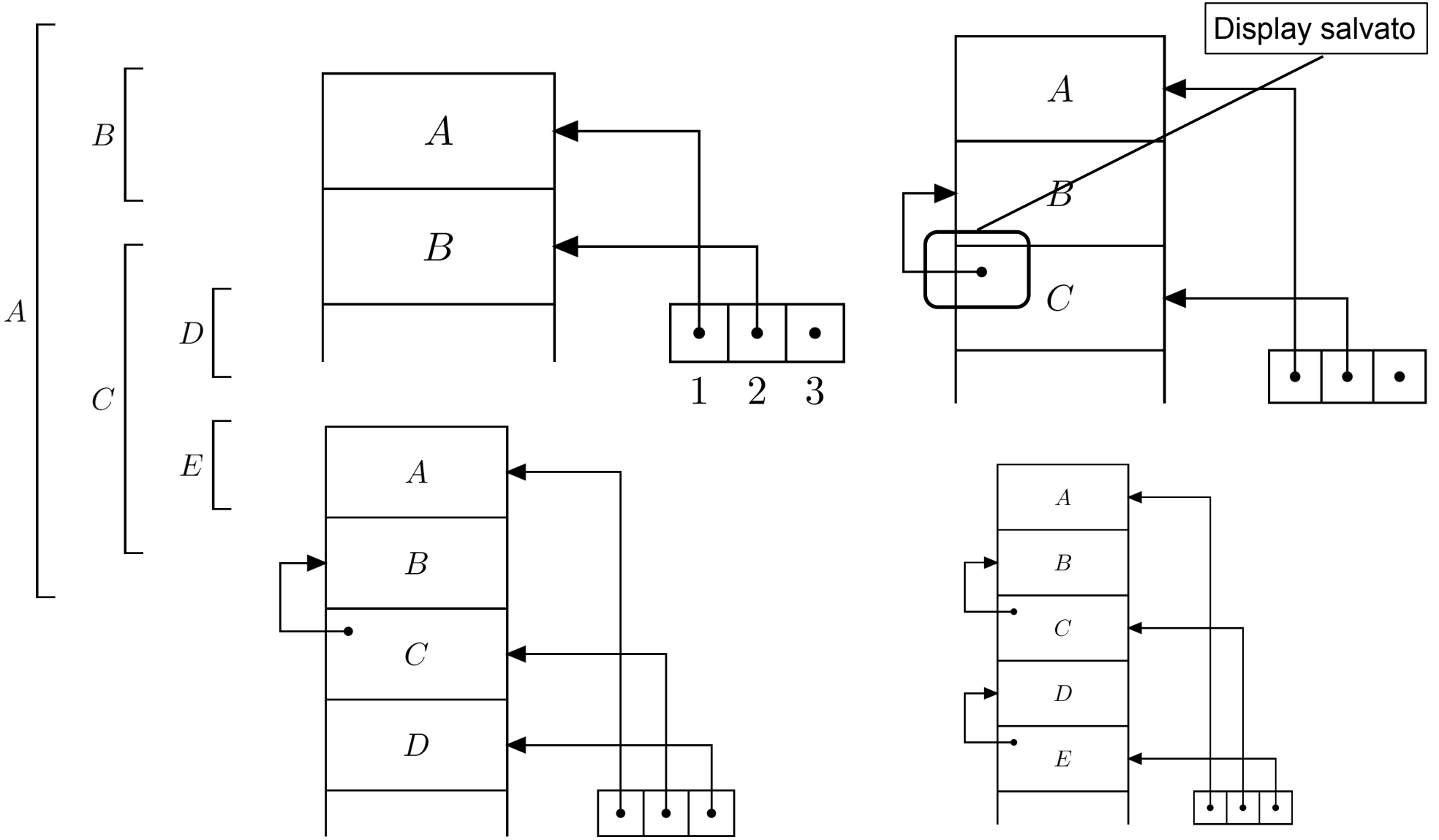
- Si può ridurre il costo derivante dalla scansione della CS ad una costante usando il *display*:

- La catena statica viene rappresentata mediante un array (detto *display*):
  - i-esimo elemento dell'array = puntatore all'RdA del sottoprogramma di livello di annidamento i, attivo per ultimo

- 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

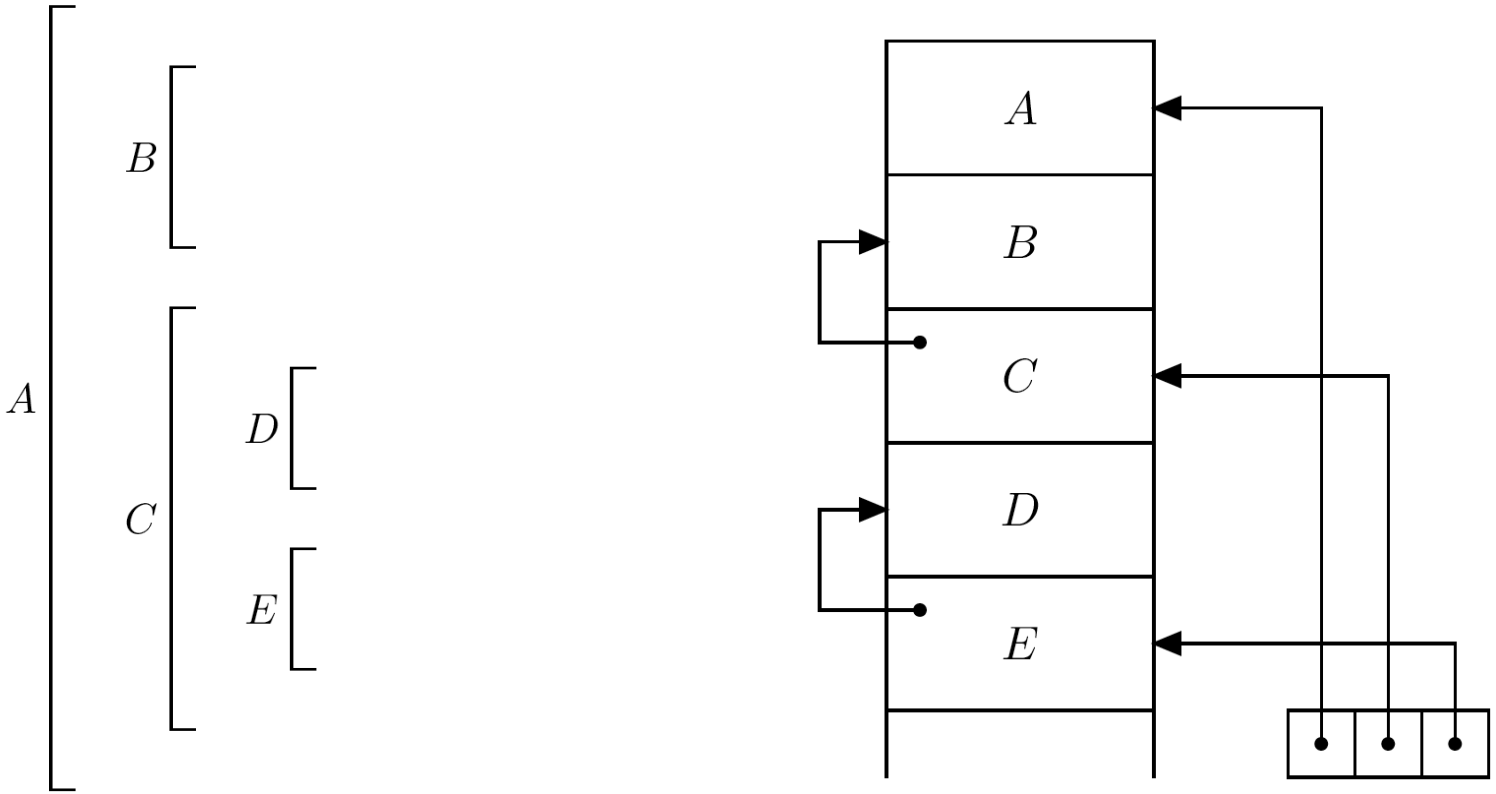
Display

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



Display

- *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 *Display*[i - h]
Con *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 Display[j] nel proprio RdA e vi mette una copia del proprio (nuovo) punt a RdA.
- Funziona: soliti due casi
  - P dichiarata immediatamente in Ch (k=0); Ch e P condividono Display fino al livello corrente (che è esteso di 1)
  - P dichiarata in un blocco k passi fuori Ch; Ch e P condividono Display fino al livello j-1.
- Comunque rara lunghezza catena statica >3, display poco usato nelle implementazioni moderne…

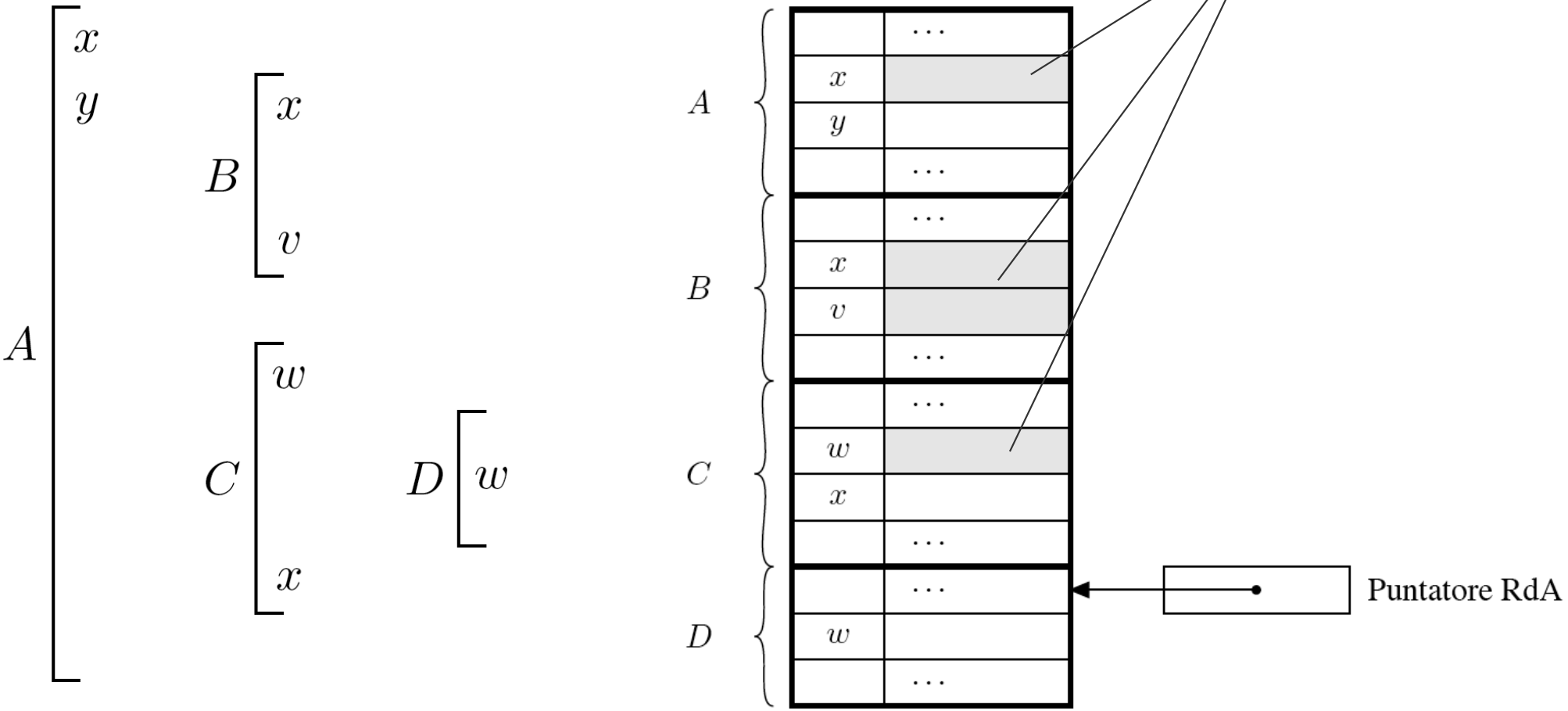
Scope dinamico

- 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).

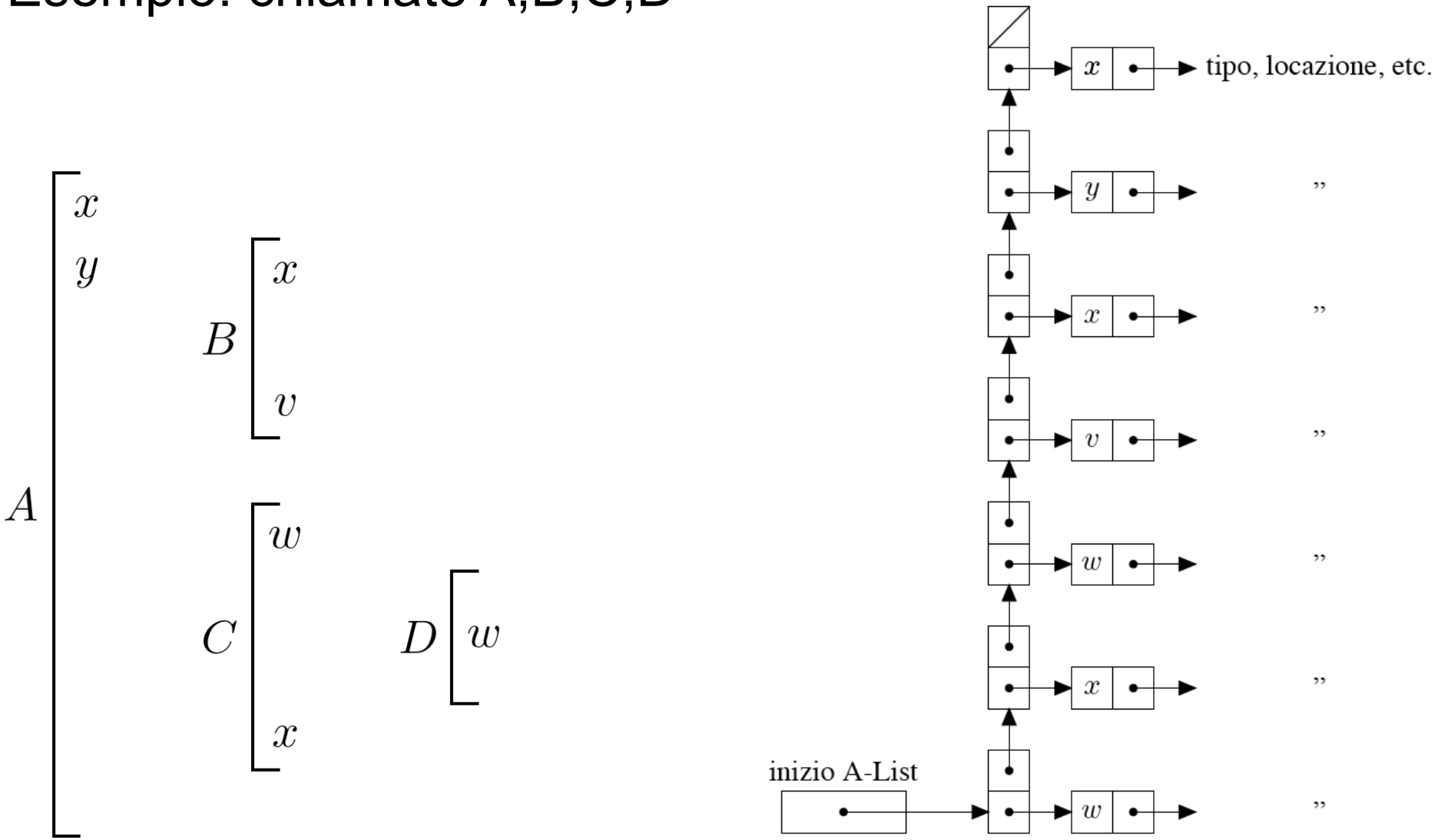
Implementazione ovvia

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



Variante: A-list

- 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

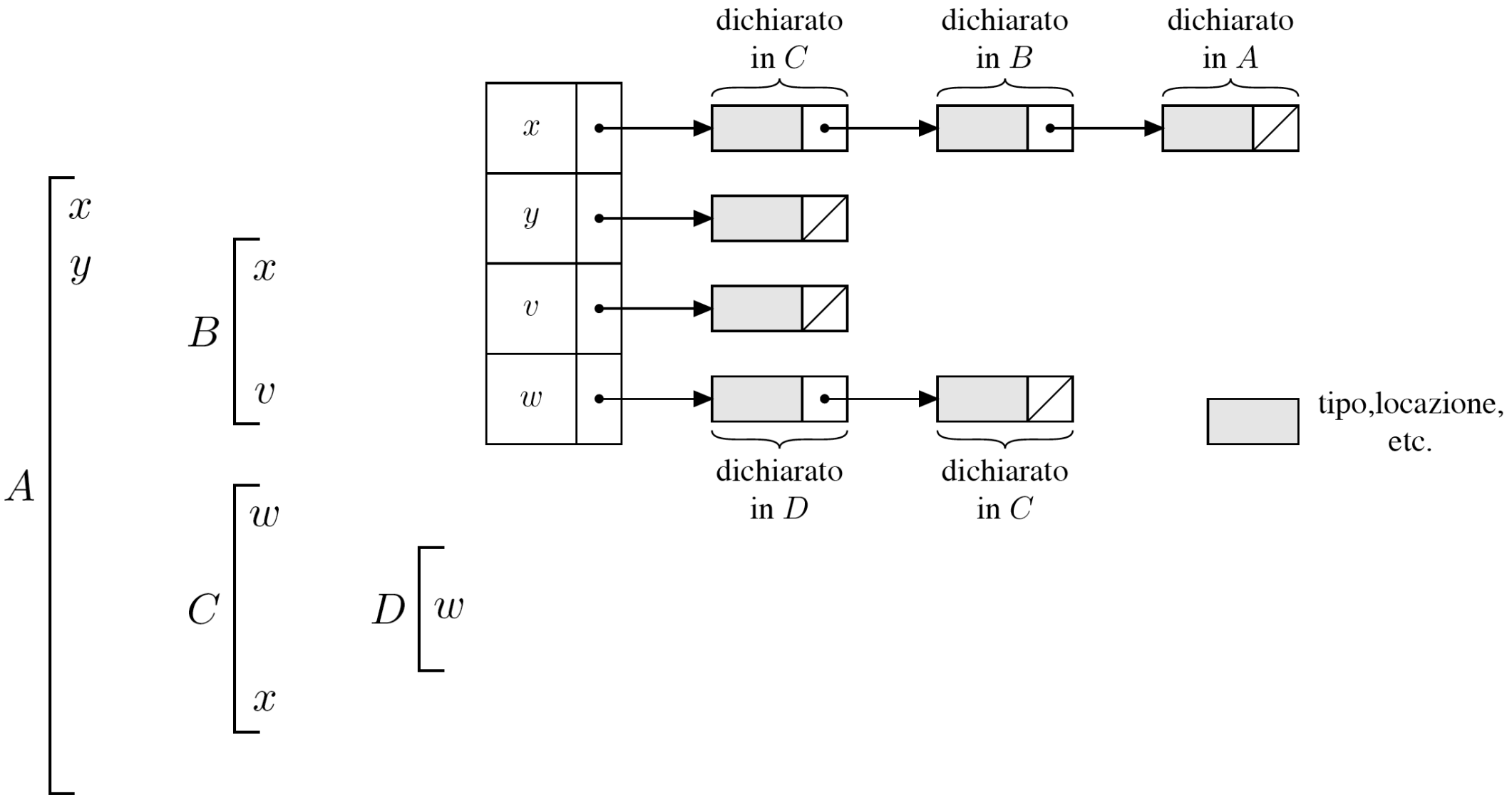
- Possiamo ridurre il tempo d'accesso medio, aumentando il tempo di ingresso/uscita da blocco...

Tabella centrale dei riferimenti, CRT

- Evita le lunghe scansioni delle A-list
- Una tabella mantiene tutti i nomi distinti del programma
  - se nomi noti staticamente accesso in tempo costante
  - altrimenti, accesso hash
- 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

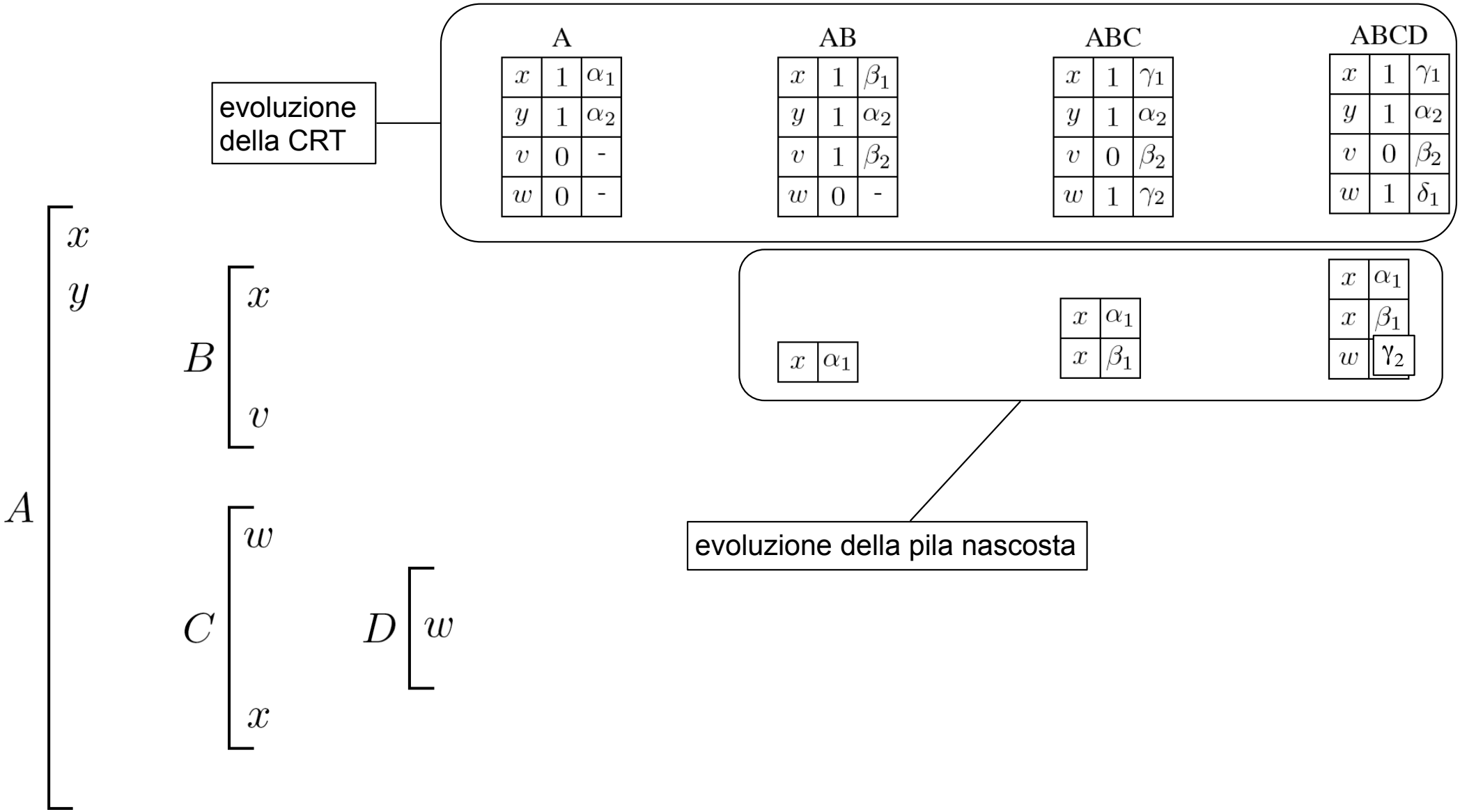
Esempio (CRT)

- Esempio: chiamate A,B,C,D



CRT con pila nascosta

- 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...