Gestione della memoria

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

M. Gabbrielli, S. Martini

Linguaggi di programmazione: principi e paradigmi

McGraw-Hill Italia, 2005

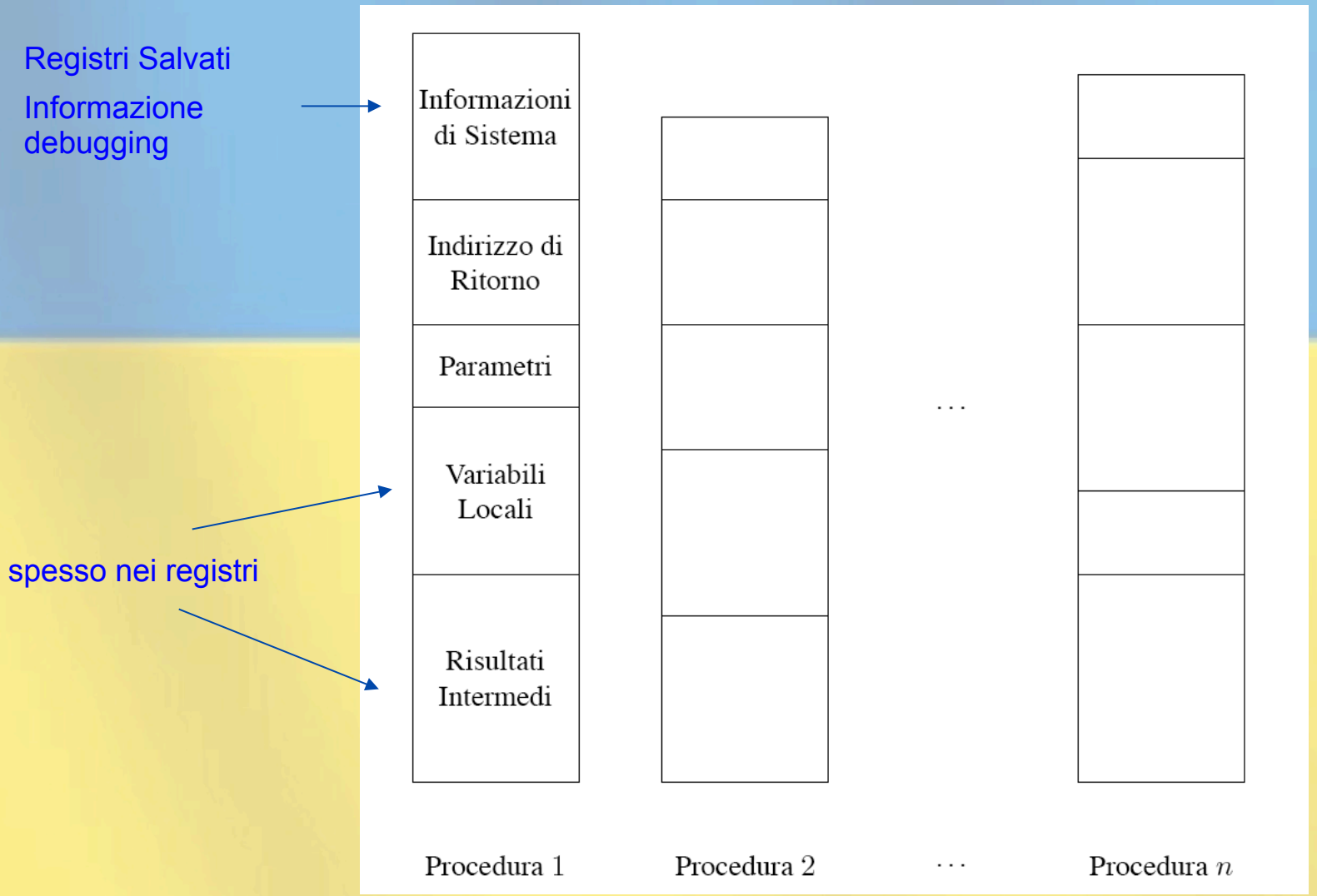
Tipi di allocazione della memoria

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

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



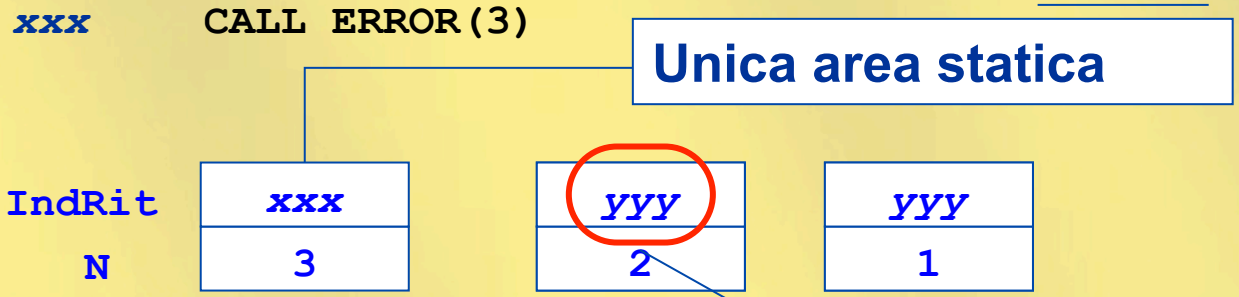
L'allocazione statica non permette ricorsione

FORTRAN.

Programma sintatticamente illegale: non ammessa chiamata ricorsiva

```
        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



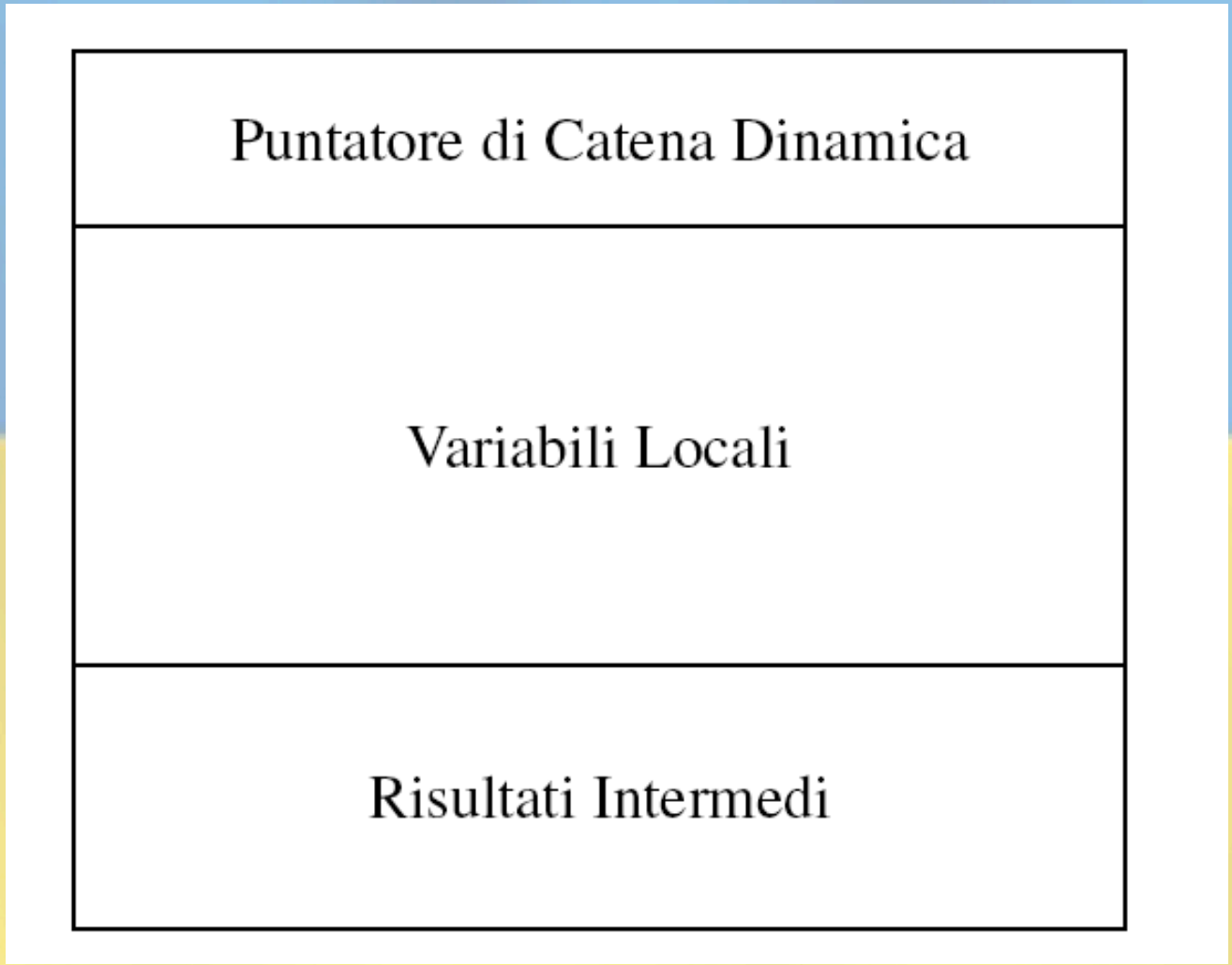
OUTPUT 1 1 1 1 1 1 …

L'indirizzo di ritorno originale è perduto

Allocazione dinamica: pila

- Con ricorsione l'allocazione statica non basta:
  - a run time possono esistere *più istanze* della stessa variabile locale di una procedura
- Ogni istanza di un sottoprogramma a run-time ha una porzione di memoria detta record di attivazione (o frame) contente 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

Record di attivazione per blocchi anonimi



Allocazione dinamica con pila

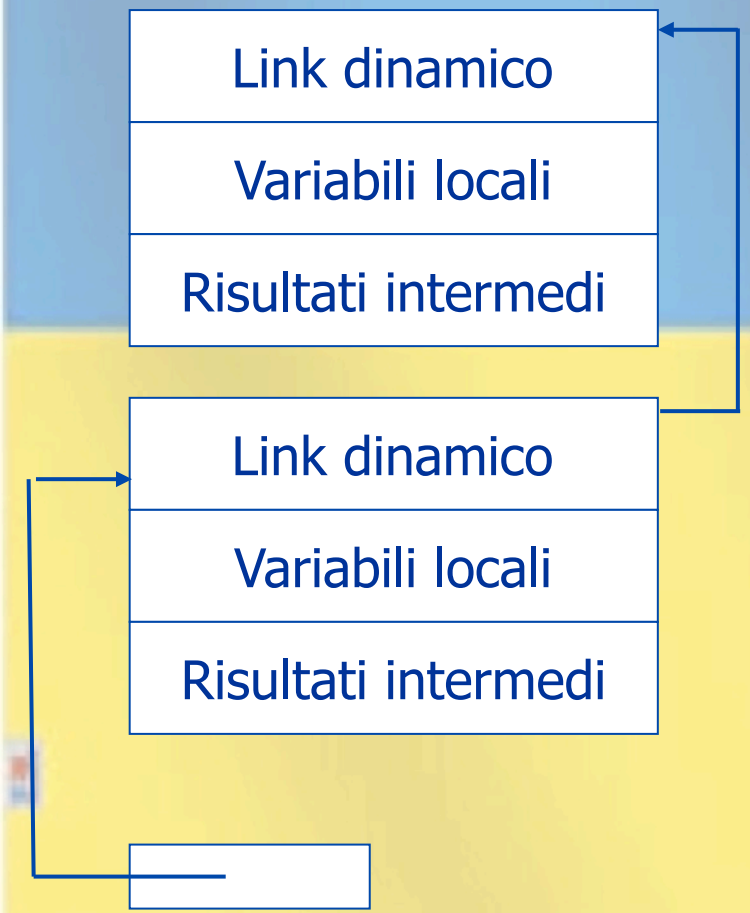
- La gestione della pila è compiuta mediante:
  - sequenza di chiamata (il codice eseguito dal chiamante immediatamente prima della chiamata)
  - prologo (codice eseguito all'inizio del blocco)
  - epilogo (codice eseguito alla fine del blocco)
  - sequenza di ritorno (il codice eseguito dal chiamante immediatamente dopo la chiamata)

- Indirizzo di un RdA non è noto a compile-time.
- Il Puntatore RdA (o SP) punta al RdA del blocco attiva
- Le info contenute in un RdA sono accessibili per offset rispetto allo SP:

  indirizzo-info = contenuto(SP)+offset

- Offset determinabile staticamente dal compilatore
- Somma SP+offset eseguita con unica istruzione macchina **load** o **store**

Record di attivazione per blocchi in-line



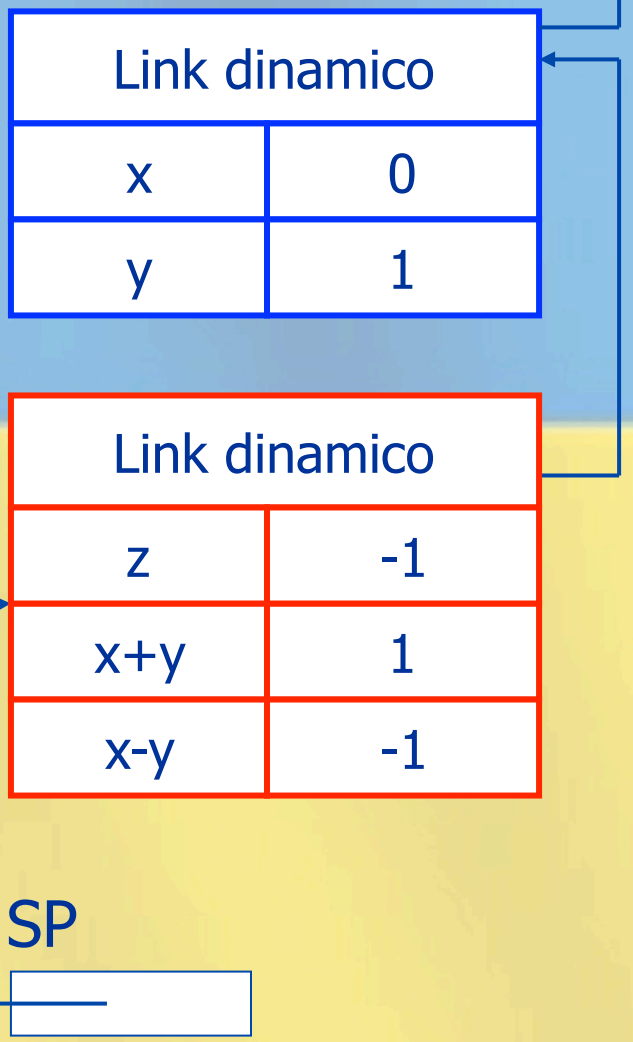
- Link dinamico (o control link)
  - puntatore al precedente record sullo stack
- Ingresso nel blocco: Push
  - link dinamico del nuovo Rda := SP
  - SP aggiornato a nuovo RdA
- Uscita dal blocco: Pop
  - Elimina RdA puntato da SP
  - SP := link dinamico del Rda tolto dallo stack

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

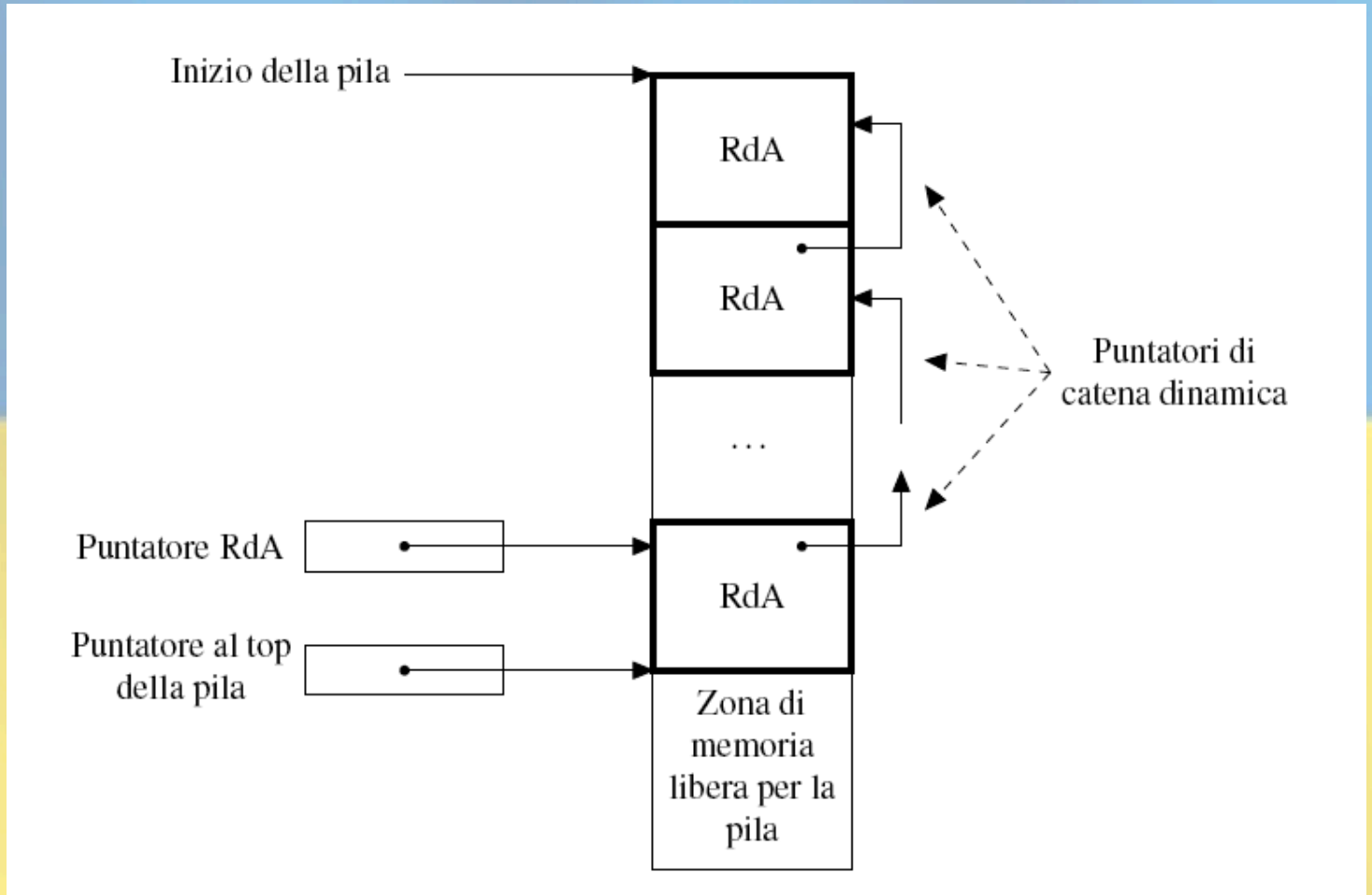


In realtà…

- **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



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).

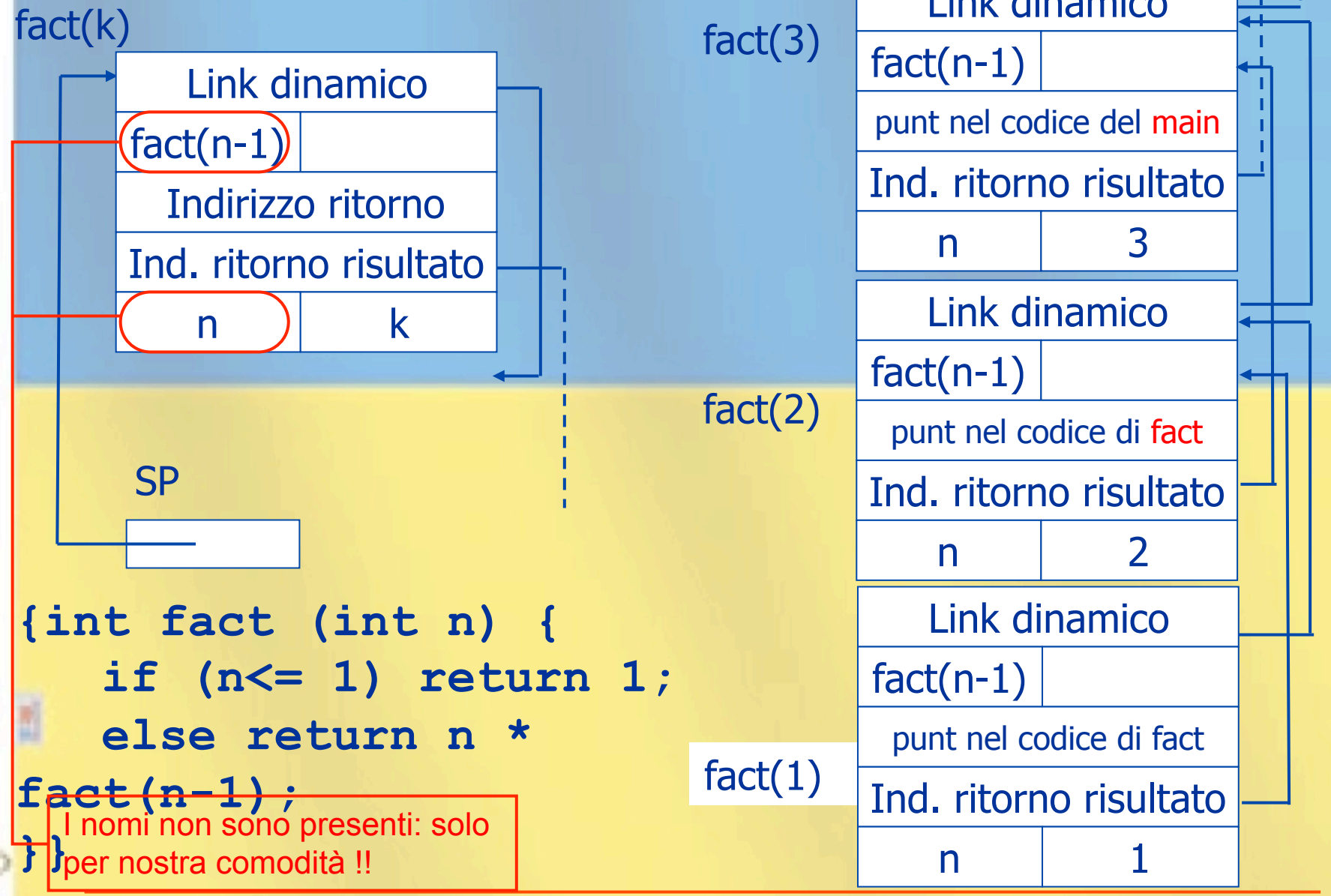
Esempio


```
{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

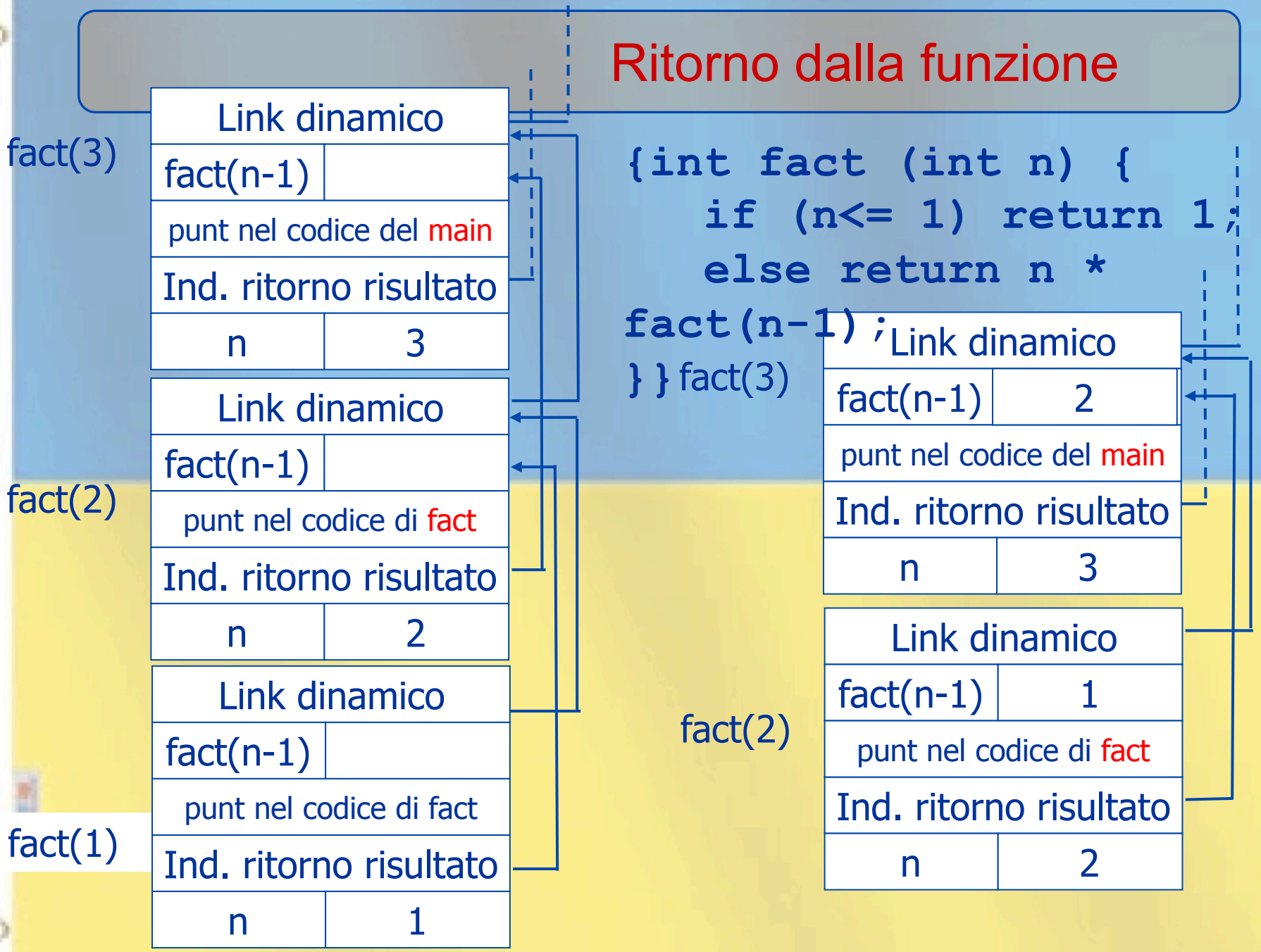
Chiamata della funzione: `fact(3);`




```
{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



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

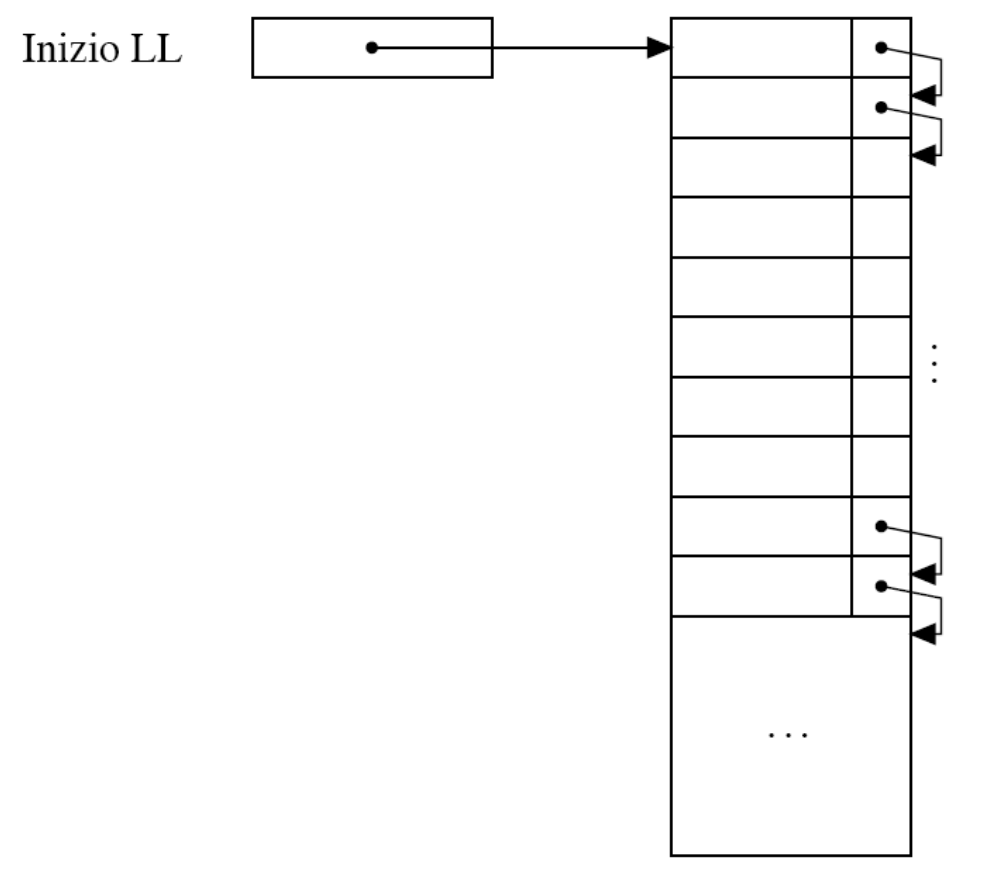
- Sequenza di uscita ed epilogo si dividono i seguenti compiti:
  - Restituzione dei valori dal chiamato al chiamante, oppure il valore calcolato dalla funzione
  - Ripristino dei registri
    - In particolare deve essere ripristinato il vecchio valore del puntatore al RdA.
  - Eventuale finalizzazione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

Allocazione dinamica con heap

- 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 (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

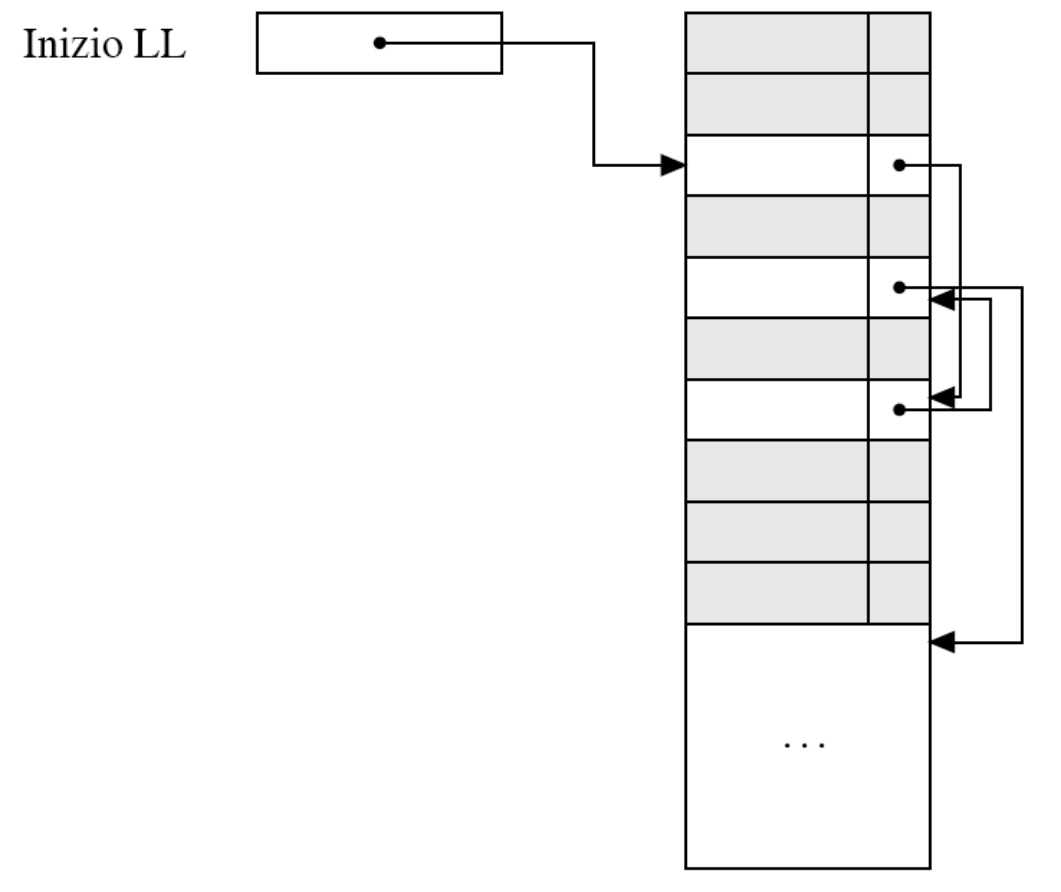
Heap: blocchi di dimensione fissa

- 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
- Deallocazione: restituzione alla lista libera



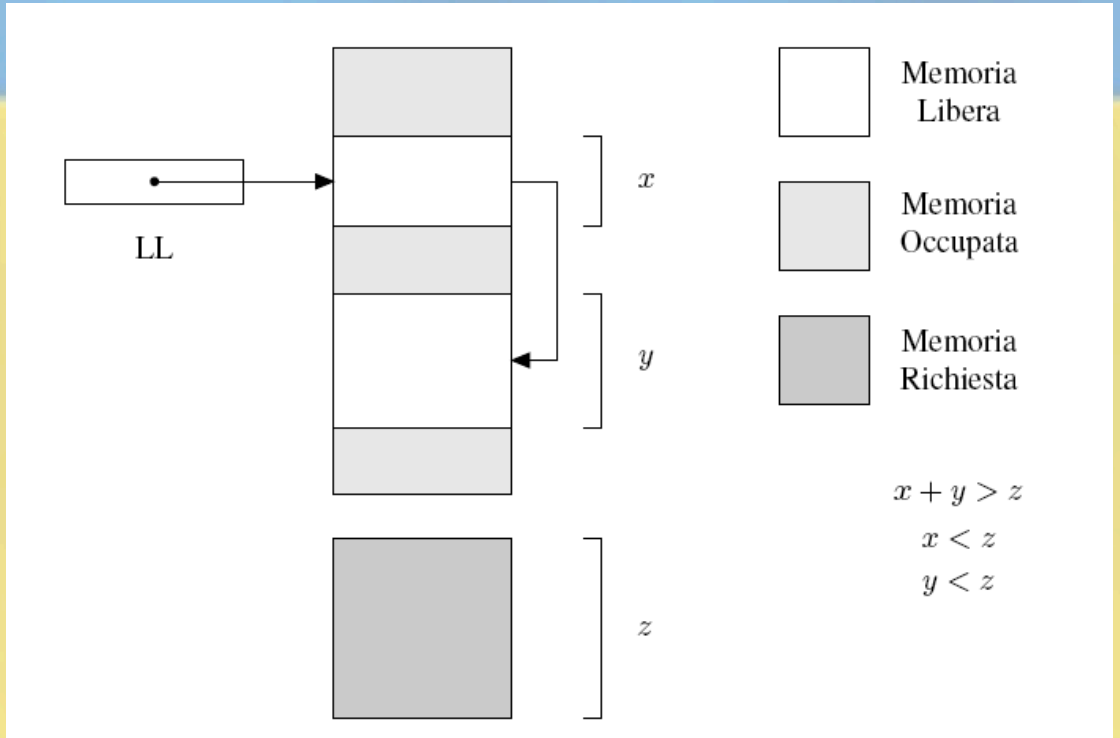
Heap: blocchi di dimensione variabile

- 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 contiene 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 restitutito 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 comunque lineare nel numero di blocchi liberi. Per migliorare:

  - mantieni liste libere multiple

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*), se disponibile
  - Fibonacci simile, ma si usano numeri di Fibonacci invece di potenze di 2 (crescono più lentamente)

Implementazione delle regole di scope

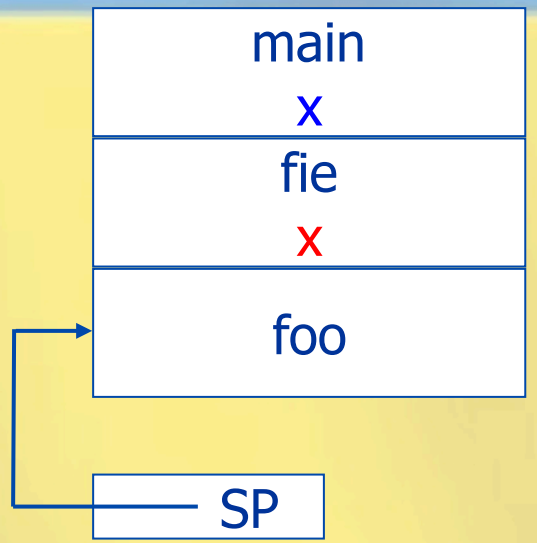
- Scope statico
  - catena statica
  - display
- Scope dinamico
  - A-list
  - Tabella centrale dell'ambiente (CRT)

Come si determina il legame corretto?

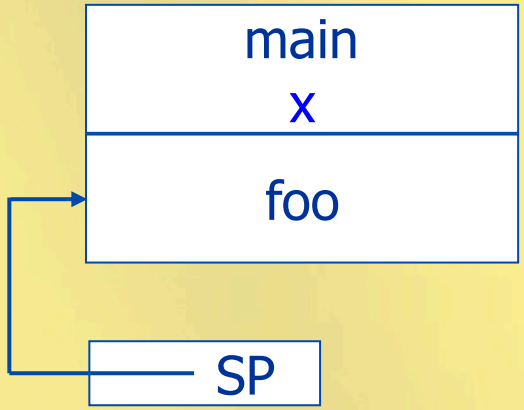
- Il codice di `foo` deve accedere sempre alla stessa variabile x
- Tale x è memorizzato in un certo RdA (in questo caso in quello del *main*)
- In cima alla pila abbiamo il RdA di `foo` (perché `foo` è in esecuzione)

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

primo caso:
foo chiamato dentro fie

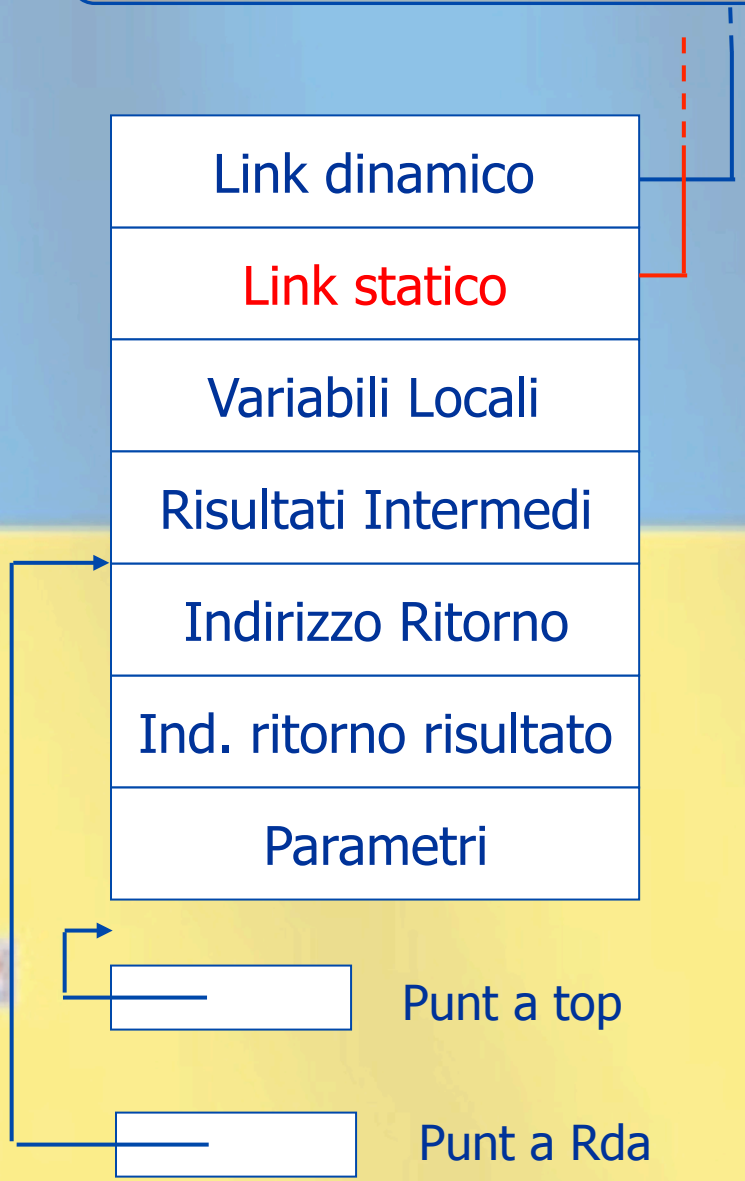


secondo caso:
foo chiamato dal main



- 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

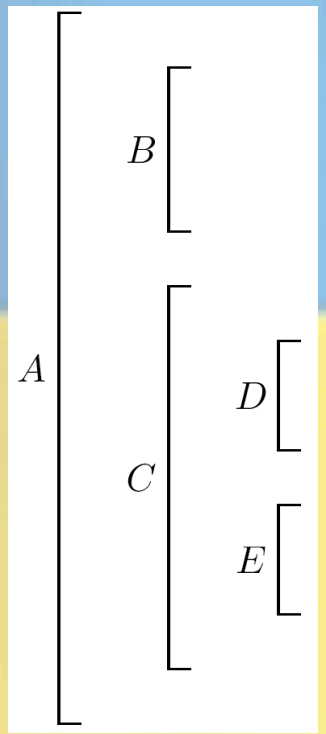


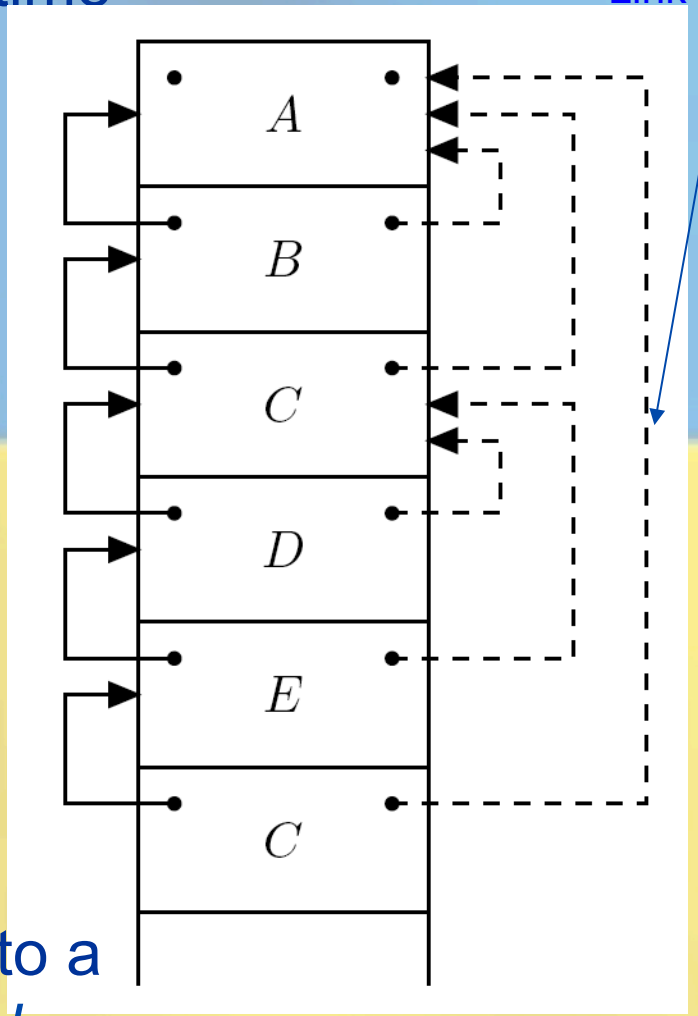
- Link dinamico:
  - puntatore all'RdA precedente sulla pila (RdA del chiamante)
- Link statico:
  - puntatore all'RdA del blocco che contiene immediatamente il testo del blocco in esecuzione

- 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

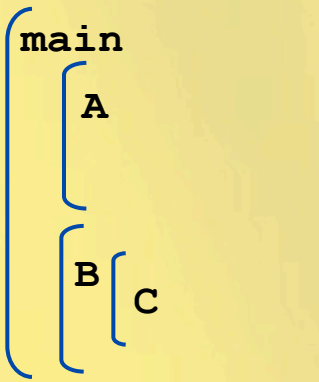


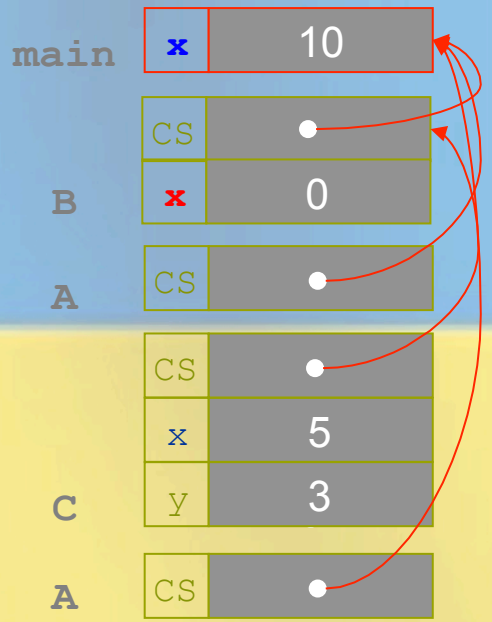


Se un sottoprogramma è annidato a livello *k*, allora la catena è lunga *k*

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();
}
```

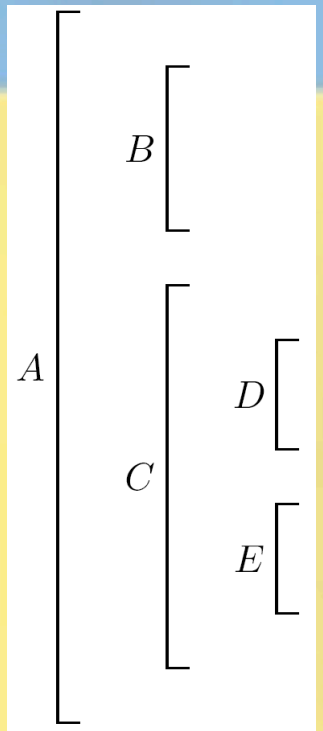


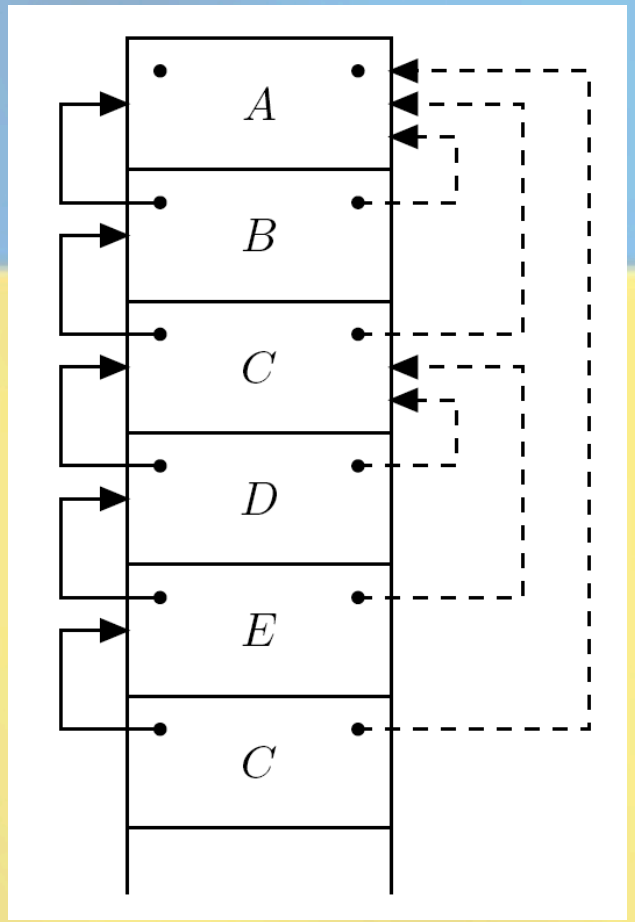


Dal punto di vista del supporto a run-time

- 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

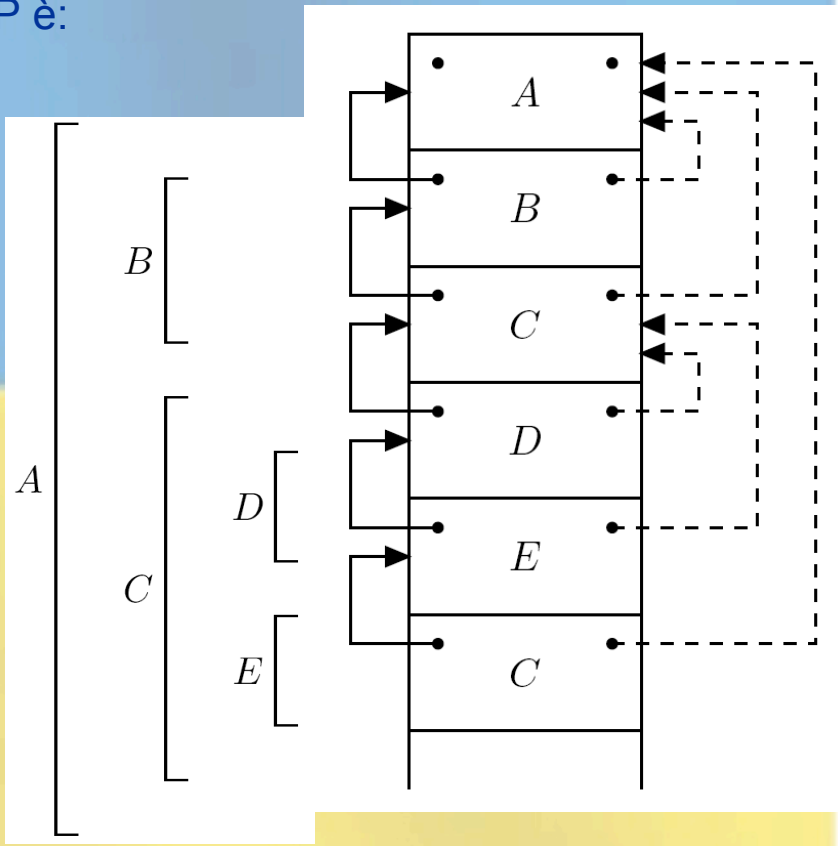




Come determinare il puntatore di CS

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 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≠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

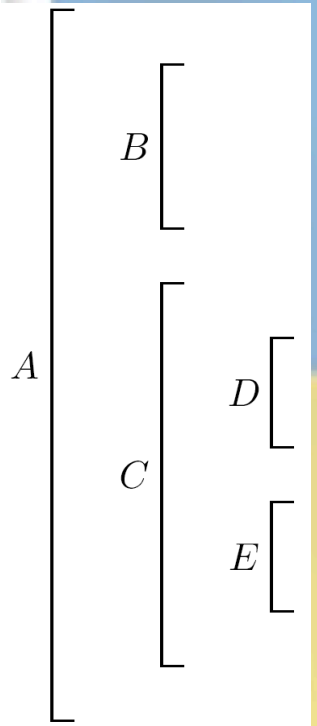
Tentiamo di ridurre i costi: il *display*

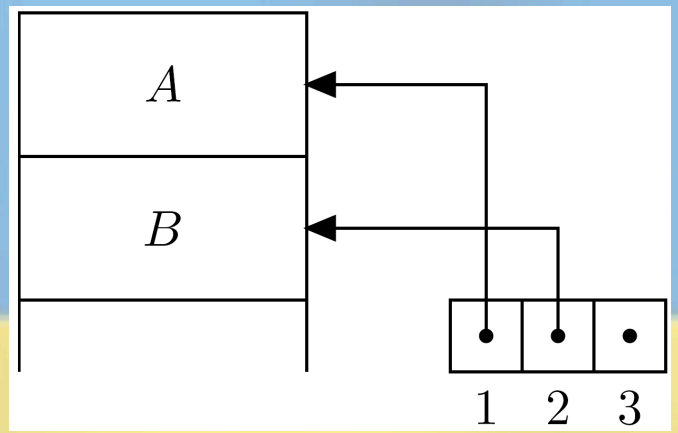
- Si può ridurre il costo h ad una costante usando la tecnica del *display*:
  - la catena statica viene rappresentata mediante un array:
    - i-esimo elemento dell'array = punt all'RdA del sottoprogramma di livello di annidamento i, attivo per ultimo
  - Dunque:
    - *Display*[1]=RdA di una proc P 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]

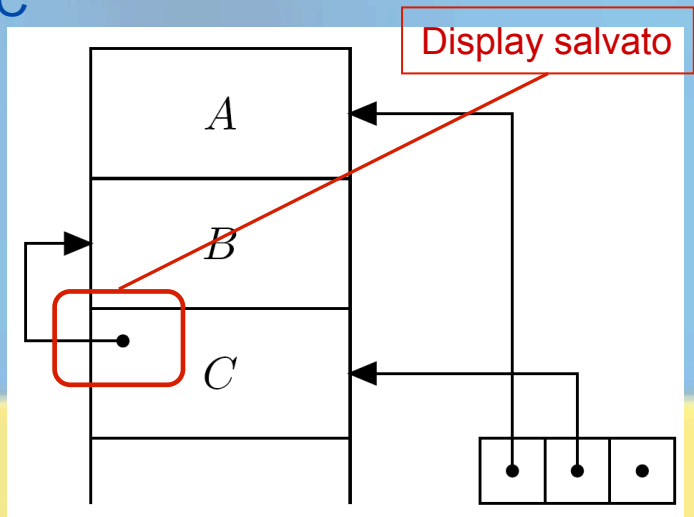
- Se il sottoprogramma corrente è annidato a livello i, un oggetto che è in uno scope esterno di h livelli può essere trovato guardando il punt a RdA nel *display* alla posizione j = i - h

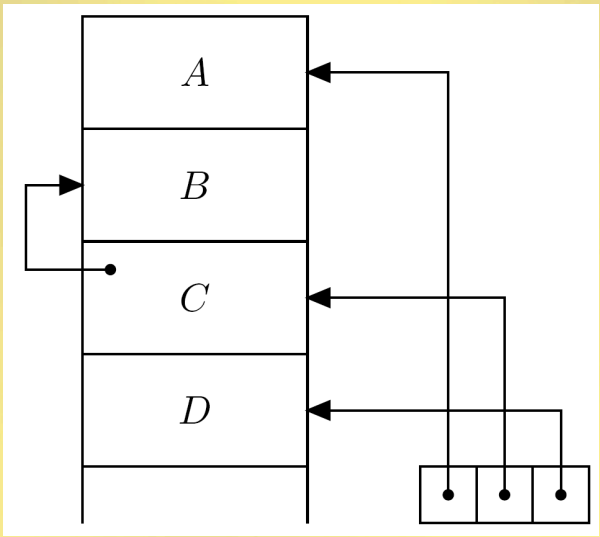
Display

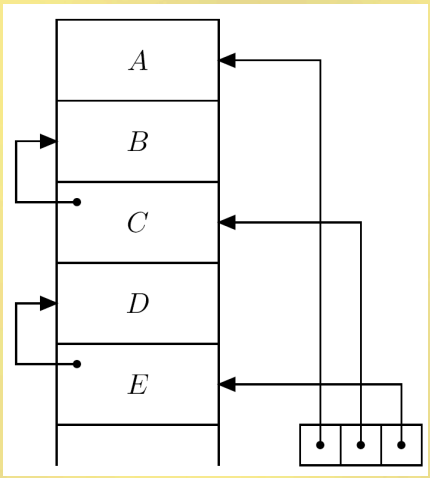
- *Display*[i] = Punt RdA della proc di livello i, attiva per ultimo
- Sequenza di chiamate: A, B, C, D, E, C





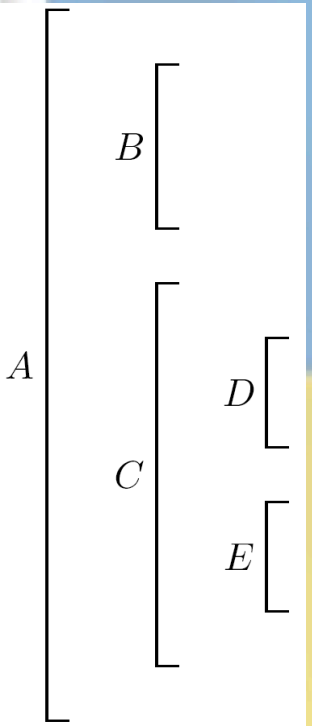


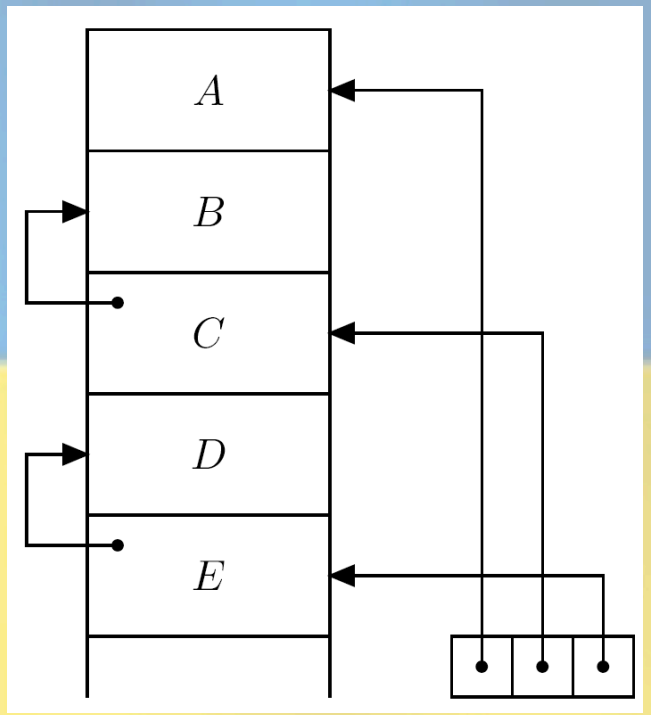




Display

- *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. 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 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. Display[j] deve essere modificato (dopo il salvataggio).

Display o catena statica ?

- Rari annidamenti di profondità > 3, quindi lunghezza max di catena statica = 3
- 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…

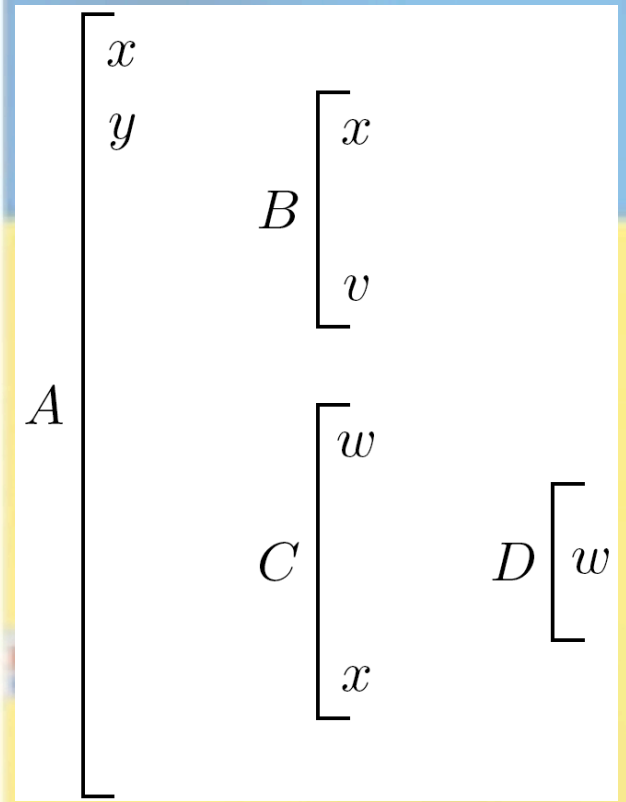
Scope dinamico

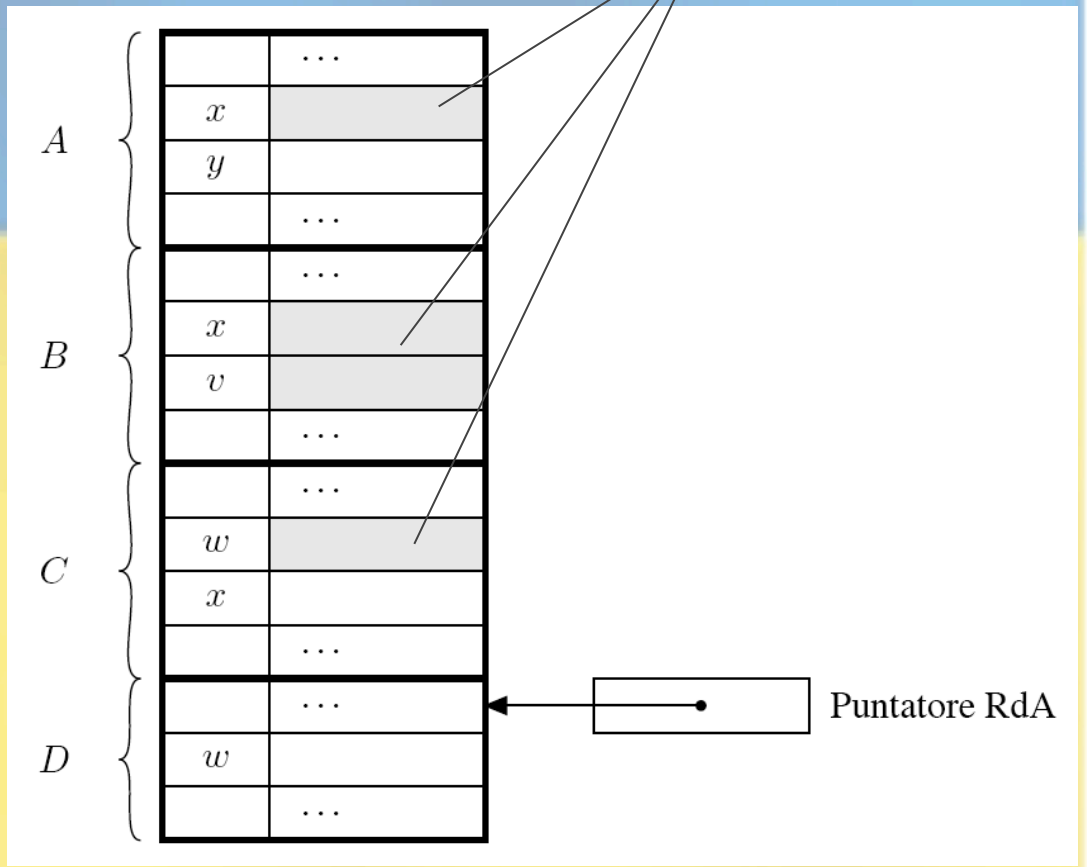
- Con scope dinamico l'associazione nomi-oggetti denotabili dipende
  - dal flusso del controllo a run-time
  - dall'ordine con cui i sottoprogrammi sono chiamati

- La regola generale è semplice: l'associazione corrente per un nome è quella determinata per ultima nell'esecuzione (non ancora distrutta).

Implementazione ovvia

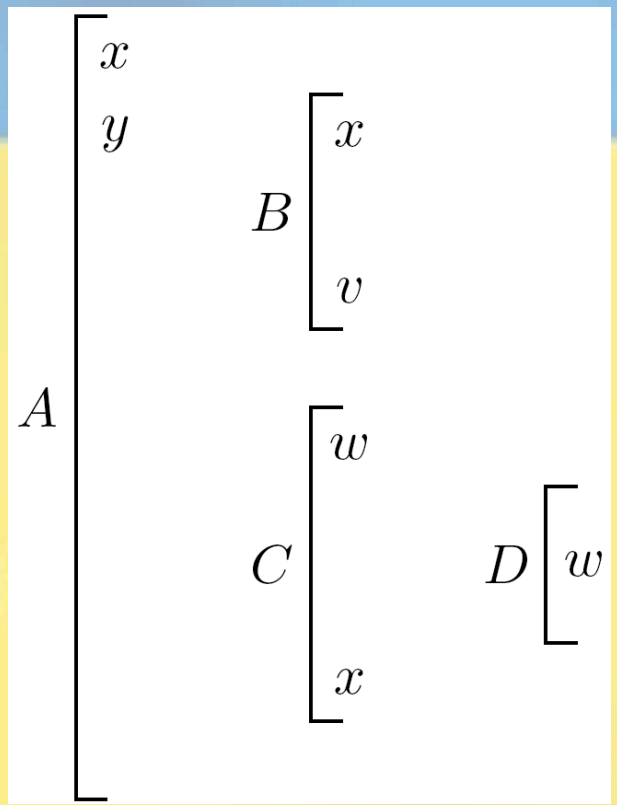
- Memorizzare i nomi negli RdA
- Ricerca per nome risalendo la pila
- Esempio: chiamate A,B,C,D

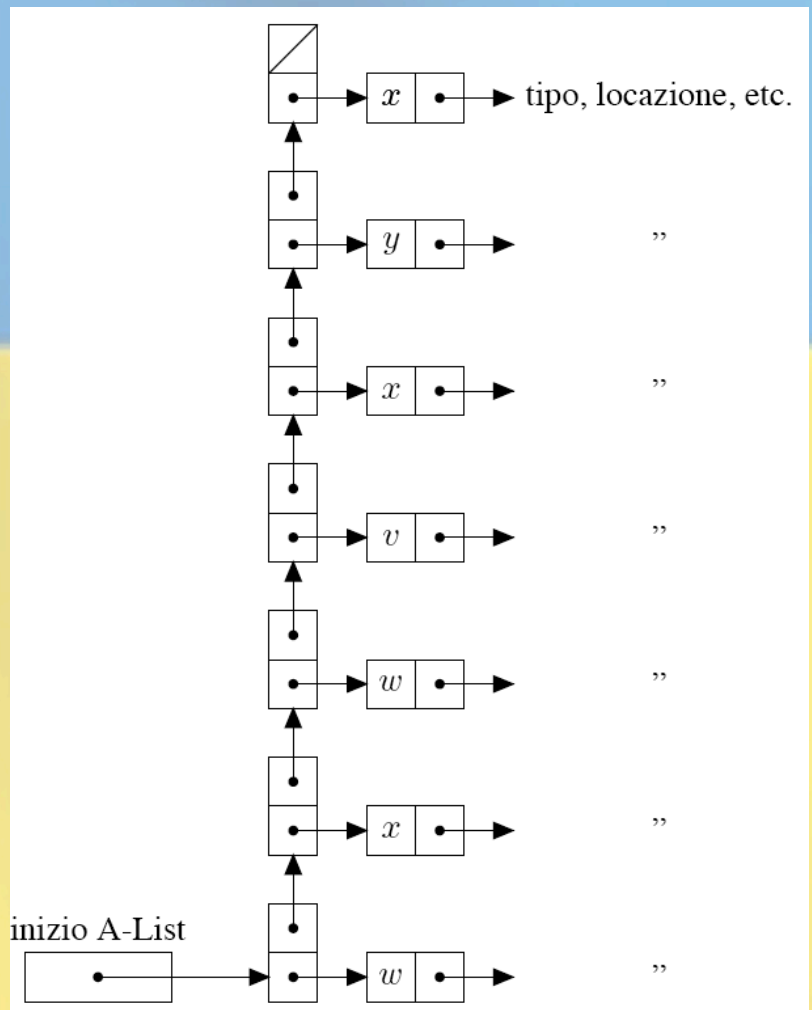




Variante: A-list

- 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

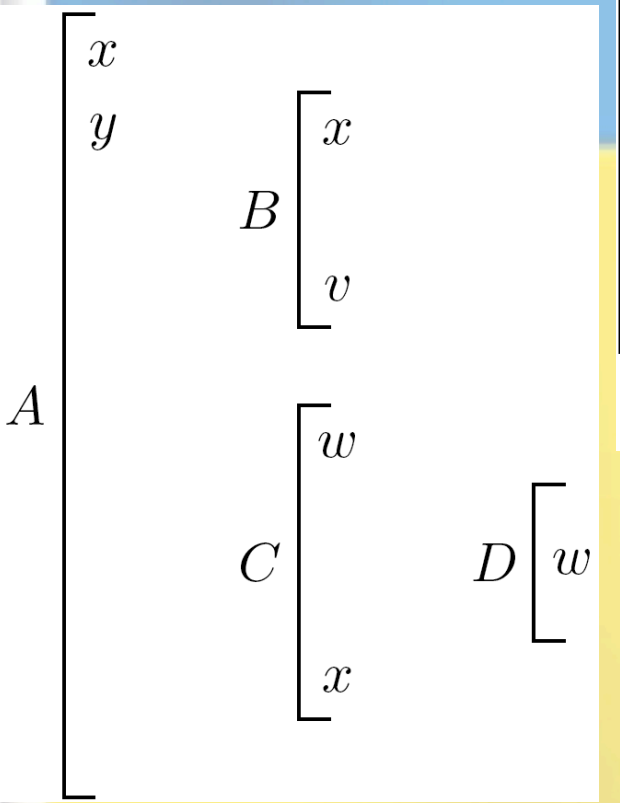
- Possiamo ridurre il tempo d'accesso medio, aumentando il tempo di ingresso/uscita da blocco...

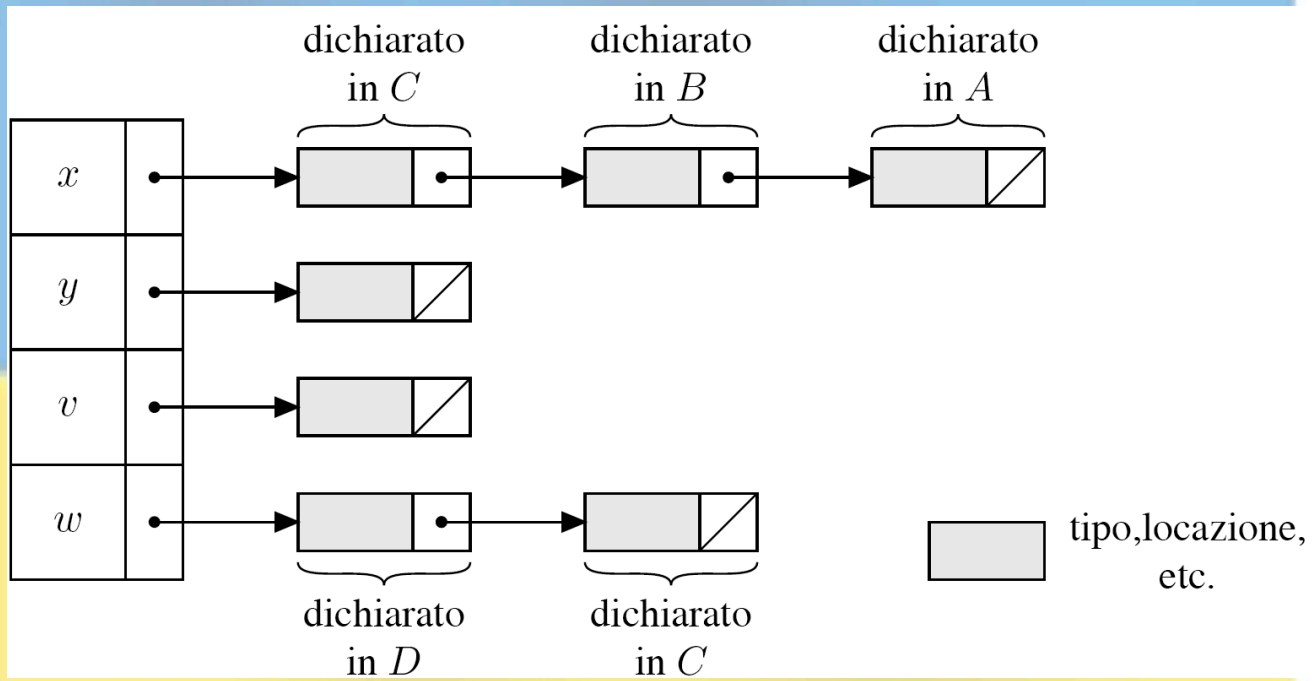
Tabella centrale dei riferimenti, CRT

- Evita le lunghe scansioni delle A-list
- Una tabella mantiene tutti i nomi distinti del programma
  - se i nomi son noti staticamente, si può accedere all'elemento della tabella 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

evoluzione della CRT

A

x	1	α_1
y	1	α_2
v	0	-
w	0	-

AB

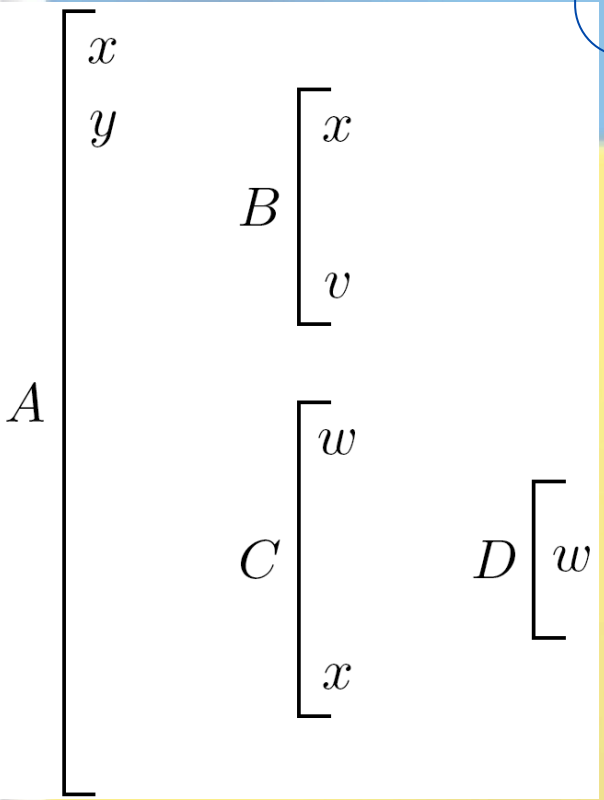
x	1	β_1
y	1	α_2
v	1	β_2
w	0	-

ABC

x	1	γ_1
y	1	α_2
v	0	β_2
w	1	γ_2

ABCD

x	1	γ_1
y	1	α_2
v	0	β_2
w	1	δ_1



x	α_1

x	α_1
x	β_1

x	α_1
x	β_1
w	γ_2

refuso su libro

evoluzione della pila nascosta

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...