

Lingaggi di Programmazione

Corso di Laurea in "Informatica"

La gestione della memoria

Valeria Carofiglio

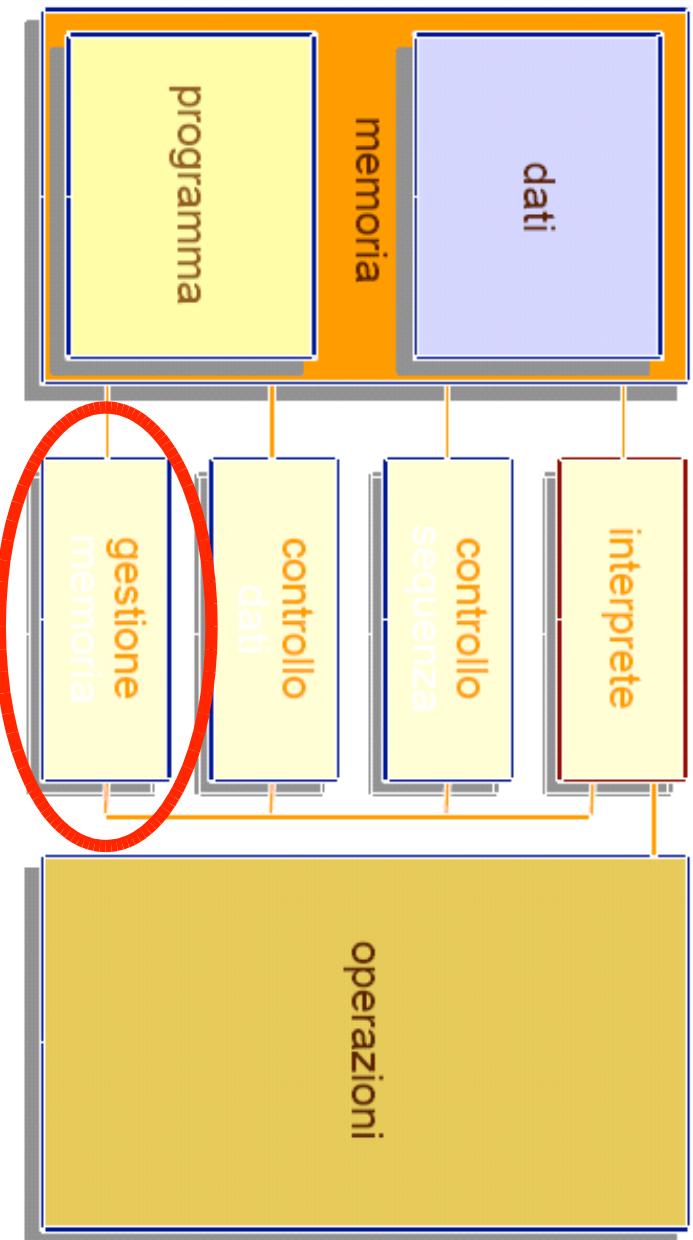
(Questo materiale è una rivisitazione del materiale
prodotto da Nicola Fanizzi)

Contenuti

- Capire la gestione della memoria nella macchina astratta
- Tecniche di gestione della memoria
 - Gestione statica
 - Gestione dinamica
- Implementazione dell'ambiente e delle regole dello scope

Gestione della memoria...

Funzionalità dell'interprete della macchina astratta



Allocazione di memoria per programmi e dati

- **COME** devono essere disposti in memoria
- **QUANTO TEMPO** devono rimanere in memoria
- **QUALI STRUTTURE DATI** per il reperimento delle informazioni dalla memoria

..livello di macchina astratta

Tipologie

- Macchina astratta di basso livello

- Gestione statica: dati e programmi in memoria prima dell'esecuzione e fino alla fine

- Macchina astratta di alto livello

- Gestione dinamica:
 - allocazione e deallocazione decisa durante l'esecuzione
 - Ricorsione: il compilatore non può stabilire un numero massimo di sottoprogrammi attivi
 - Uso di una Pila (politica LIFO)
 - Allocazione dinamica esplicita (Es. Linguaggio C)
 - Uso dell'Heap

Gestione statica

allocata dal compilatore (prima dell'esecuzione)

- Tutti gli oggetti in una zona fissata (dal compilatore)
 - Variabili globali (visibili in tutto il programma)
 - Istruzioni del codice oggetto (non cambiano durante l'esec.)
 - Costanti (indipendenti dall'esecuzione)
- Tabelle interne prodotte dal comp. (per il supporto a run time):
 - Gestione nomi
 - Controllo sui tipi
 - Garbage collection
- Linguaggio senza ricorsione
- Zona di memoria fissata per ogni sottoprogramma
 - Variabili locali
 - Parametri
 - Indirizzo di ritorno
 - Info di bookkeeping (valori di registri salvati, inf. Per debug..)

Gestione statica

Ad ogni sottoprogramma una zona di memoria per le informazioni locali

Sottoprogramma p_1

Sottoprogramma p_n

Informazioni
di Sistema

Ind. di Ritorno

Parametri

Variabili
Locali

Risultati
Intermedi

Sottoprogramma p_2

Informazioni
di Sistema

Ind. di Ritorno

Parametri

Variabili
Locali

Risultati
Intermedi

...

Sottoprogramma p_n

Informazioni
di Sistema

Ind. di Ritorno

Parametri

Variabili
Locali

Risultati
Intermedi

Chiamate successive della stessa procedura condividono la zona di memoria

Un esempio di ricorsione

Calcolare il valore dell' n -esimo numero di fibonacci

(ovvero $Fib(0)=Fib(1)=1$

$Fib(n)= Fib(n-1)+Fib(n-2)$ per $n>1$)

```
int fib(n) {  
    if (n==0) return 1;  
    else if (n==1) return 1;  
    else return fib(n-1) + fib(n-2);  
}
```

I numero di chiamate
attive
contemporaneamente
dipende da n

Ogni chiamata richiede un proprio spazio di memoria per le informazioni locali (parametri, ris.intemedi, indirizzi di ritorno)

Necessità di allocazione e deallocazione di memoria a run-time



Gestione dinamica

permessa dalla maggior parte dei LdP moderni

- Strutturazione a blocchi dei programmi
 - In-line
 - Corpi di sottoprogrammi
- Apertura e chiusura dei blocchi (es. ricorsione)
- politica LIFO (gestita usando una pila)

Gestione Dinamica

un esempio

A:

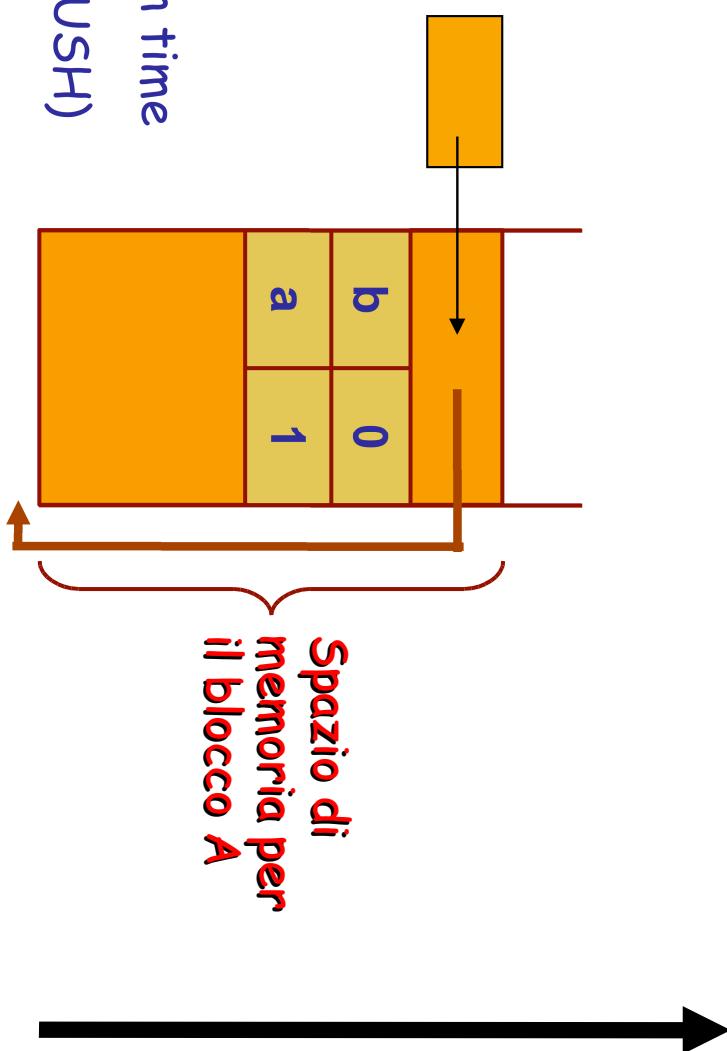
```
{  
int a = 1;  
int b = 0;  
}
```

B:

```
{  
int c = 3;  
int b = 3;  
}
```

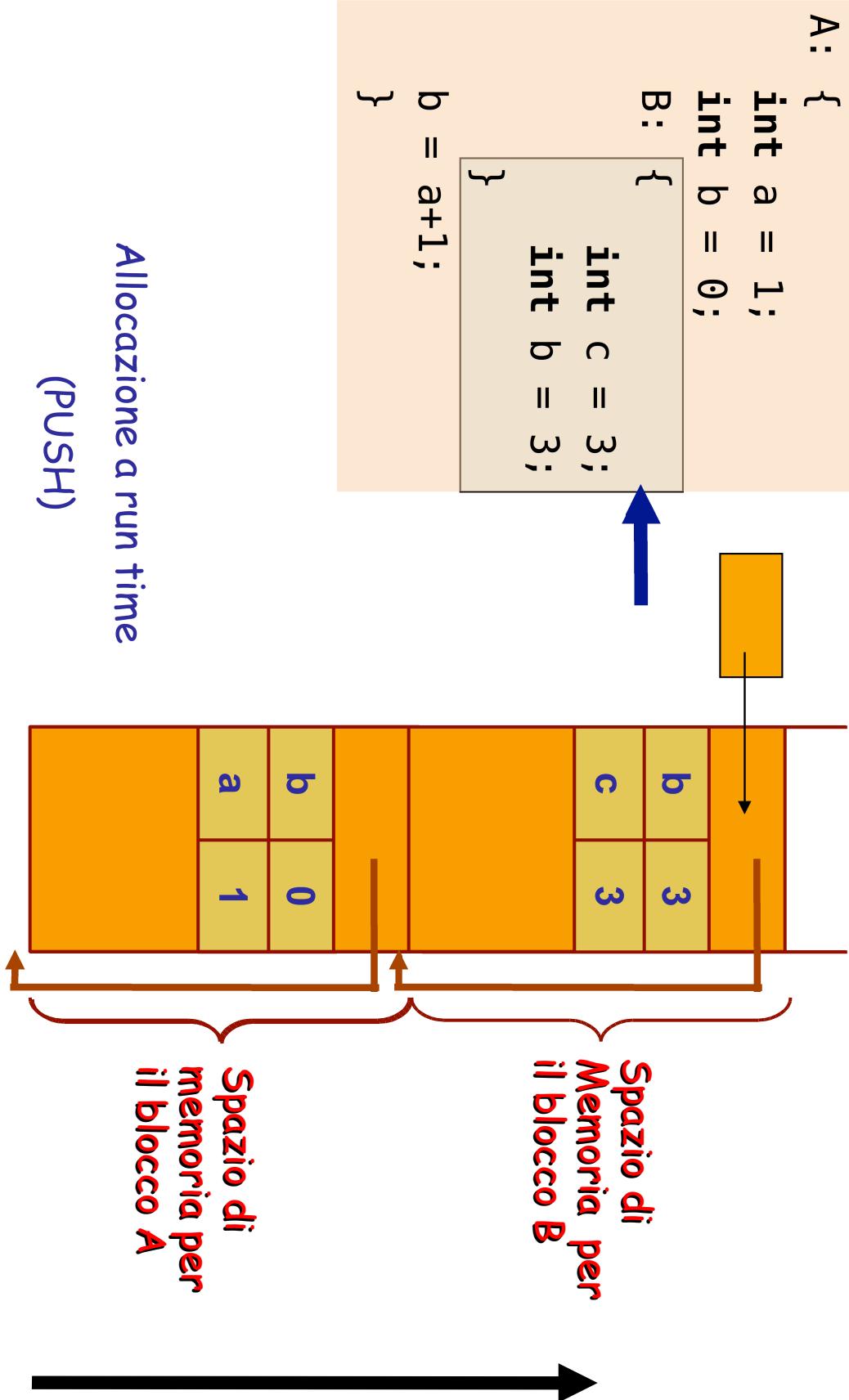
b = a+1;

Allocazione a run time
(operazione di PUSH)



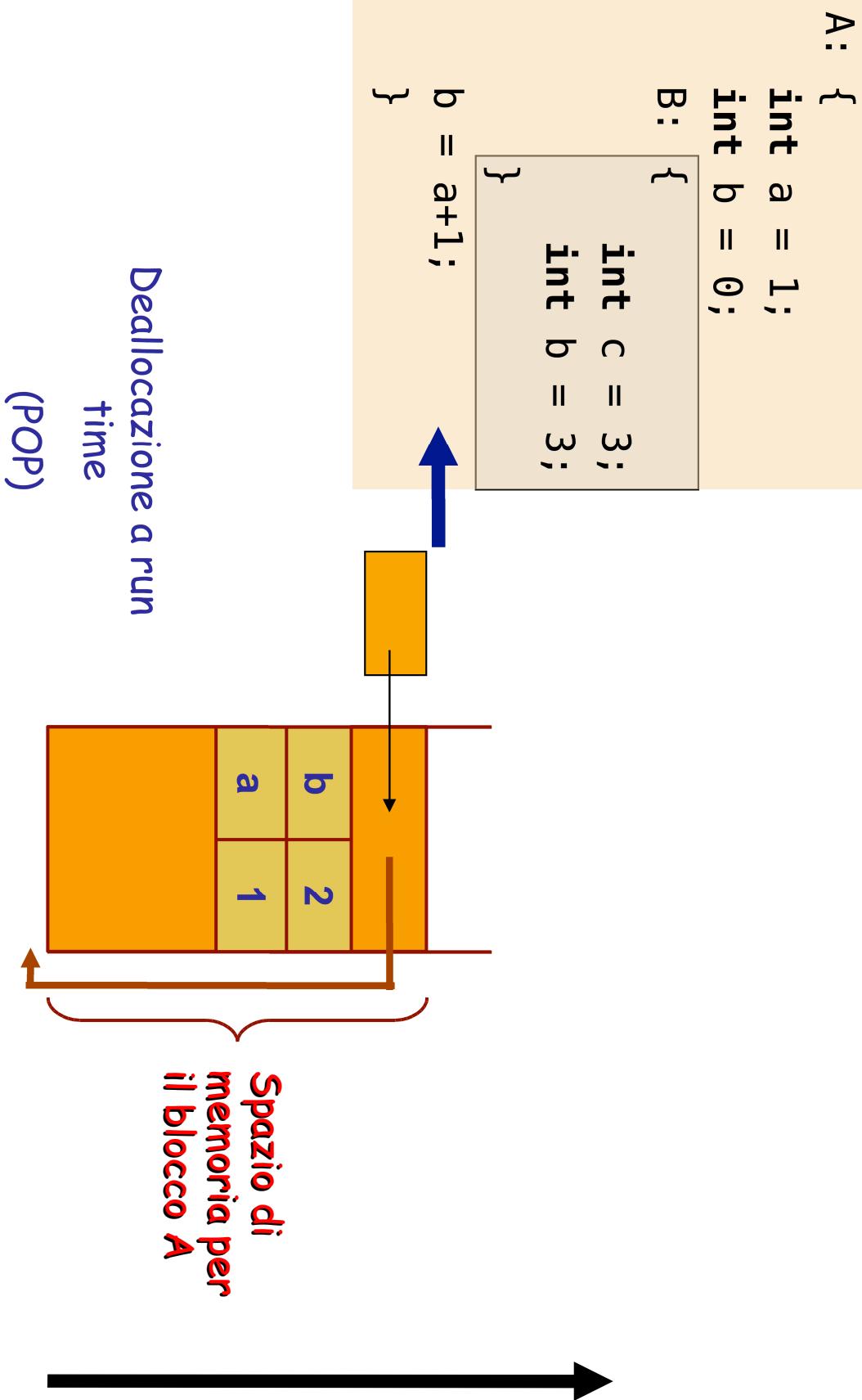
Gestione Dinamica

un esempio



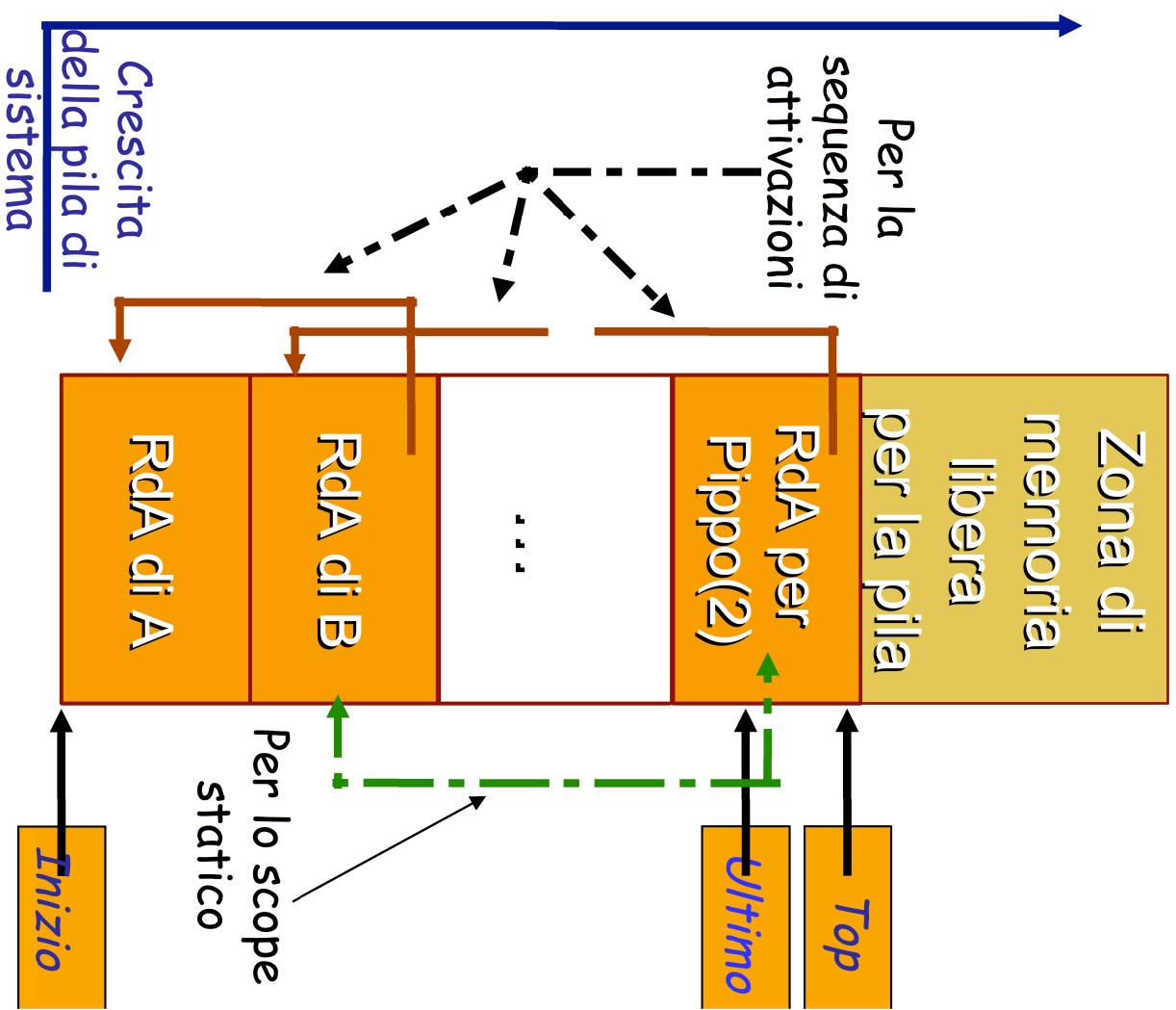
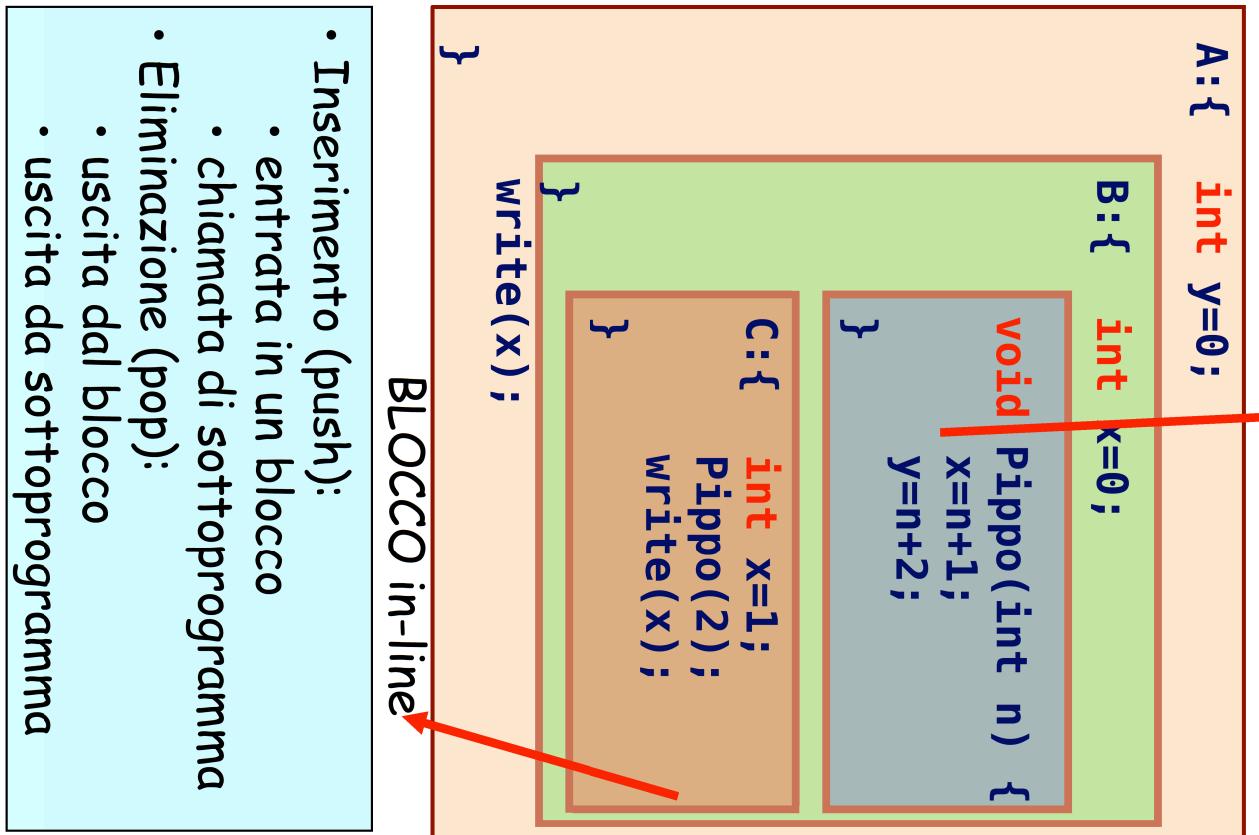
Gestione Dinamica

un esempio



corpo di sottoprogramma

Esempio



Record di attivazione

- Spazio di memoria allocata su pila per
 - **Blocchi in-line**
 - **Corpi di sottoprogramma**



ATTENZIONE: RdA
associato ad
attivazione di blocco
NON a dichiarazione

Pila di run-time

(o sistema)

Anche per linguaggi senza ricorsione

Rispetto ad una politica interamente di tipo statico

per ottimizzare l'occupazione della memoria

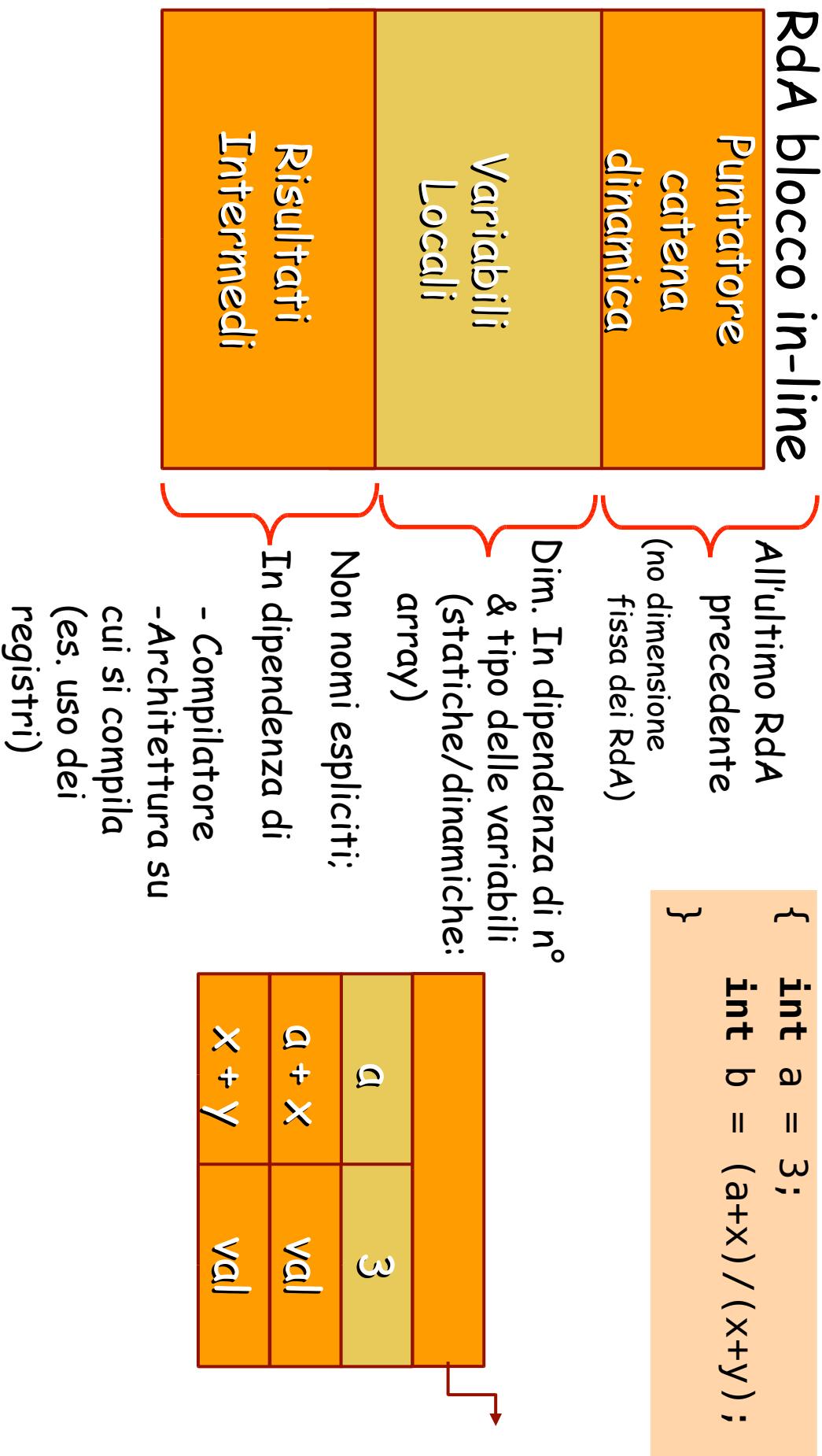
(es. N° chiamate procedura < n° procedure definite)

Record di attivazione: blocchi in-line

Struttura generica di un Rda: Dimensione non fissa

Esempio: Ris. Intermedi

```
{ int a = 3;  
  int b = (a+x) / (x+y);  
}
```



Record di attivazione: blocchi in-line

- Un RdA contiene

- Risultati intermedi
 - Calcoli complessi semplificati dal compilatore
- Variabili locali
 - Dichiarate nel blocco
- Dimensioni note a compile-time
 - Eccezione: dim. array dinamici gestite tramite (parte fissa, parte variabile)
- Puntatore catena dinamica
 - Detto anche link dinamico o di controllo
 - Punto al precedente RdA sulla pila
 - Necessario date le dimensioni potenzialmente diverse dei RdA

Record di attivazione: sottoprogrammi

Associati ad ogni attivazione di sottoprogramma

(conseguenza di chiamata - dinamicamente)

Non alla definizione della procedura stessa (staticamente)

Sottoprogramma p

Puntatore di Catena Dinamica

Puntatore Catena Statica

Indirizzo di Ritorno

Indirizzo del Risultato

Parametri

Variabili Locali

Risultati Intermedi

Per le regole di scope statico

Prima istruzione da eseguire
dopo il sottoprogramma

Nel Rda del chiamante.
Solo nelle
funzioni.

Parametri attuali

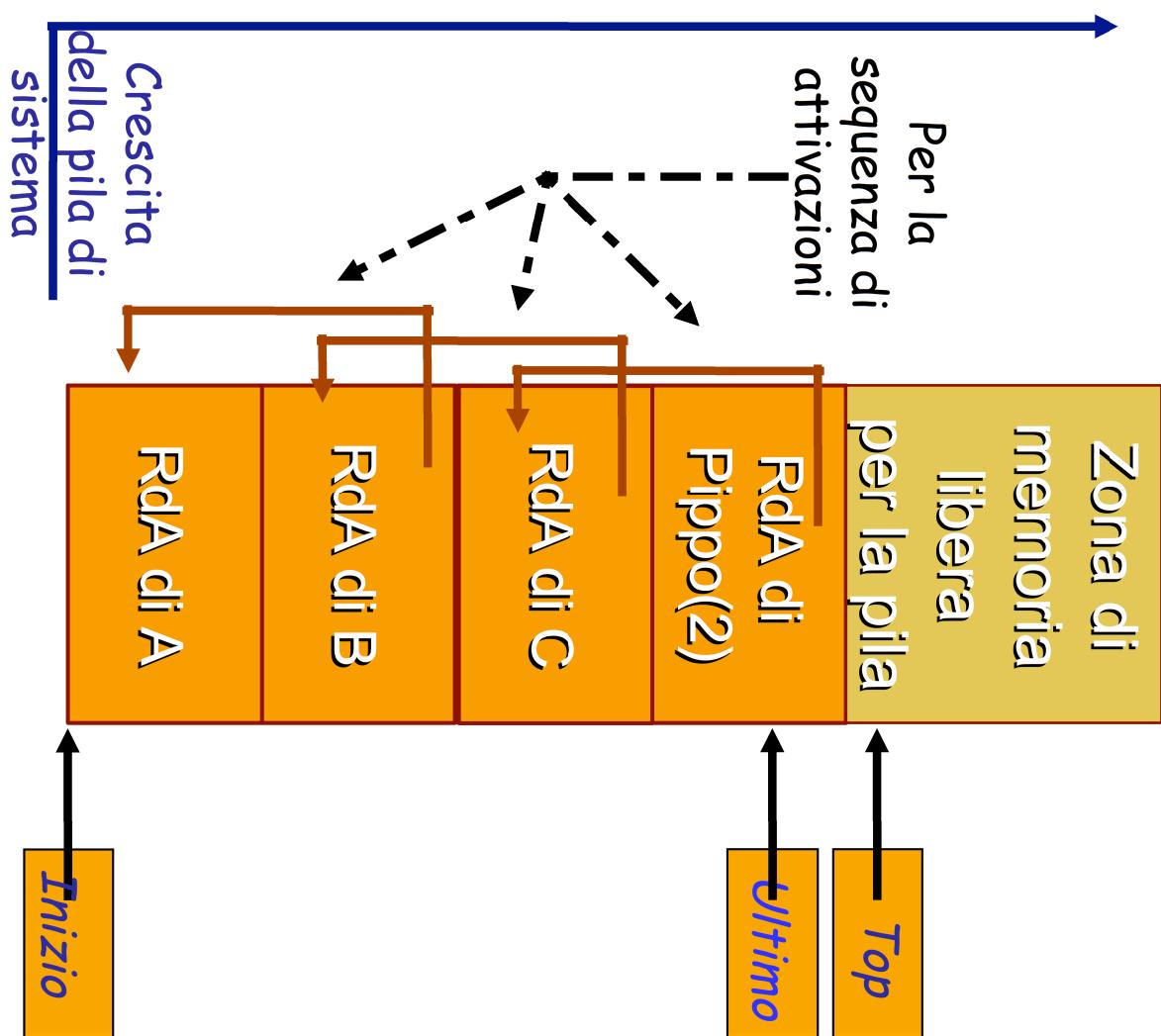
Disposizione dei campi
dipendente dalle
implementazioni

Esempio: indirizzo di ritorno

```

A:{ int y=0;
}
B:{ int x=0;
    void Pippo(int n) {
        x=n+1;
        y=n+2;
    }
}
C:{ int x=1;
    Pippo(2);
    write(x);
}
}
write(x);
}

```



Record di attivazione: sottoprogrammi

Associati ad ogni attivazione di sottoprogramma

(conseguenza di chiamata - dinamicamente)

Non alla definizione della procedura stessa (staticamente)

Sottoprogramma p

Puntatore di Catena Dinamica

Puntatore Catena Statica

Indirizzo di Ritorno

Indirizzo del Risultato

Parametri

Variabili Locali

Risultati Intermedi

Per le regole di scope statico

Prima istruzione da eseguire
dopo il sottoprogramma

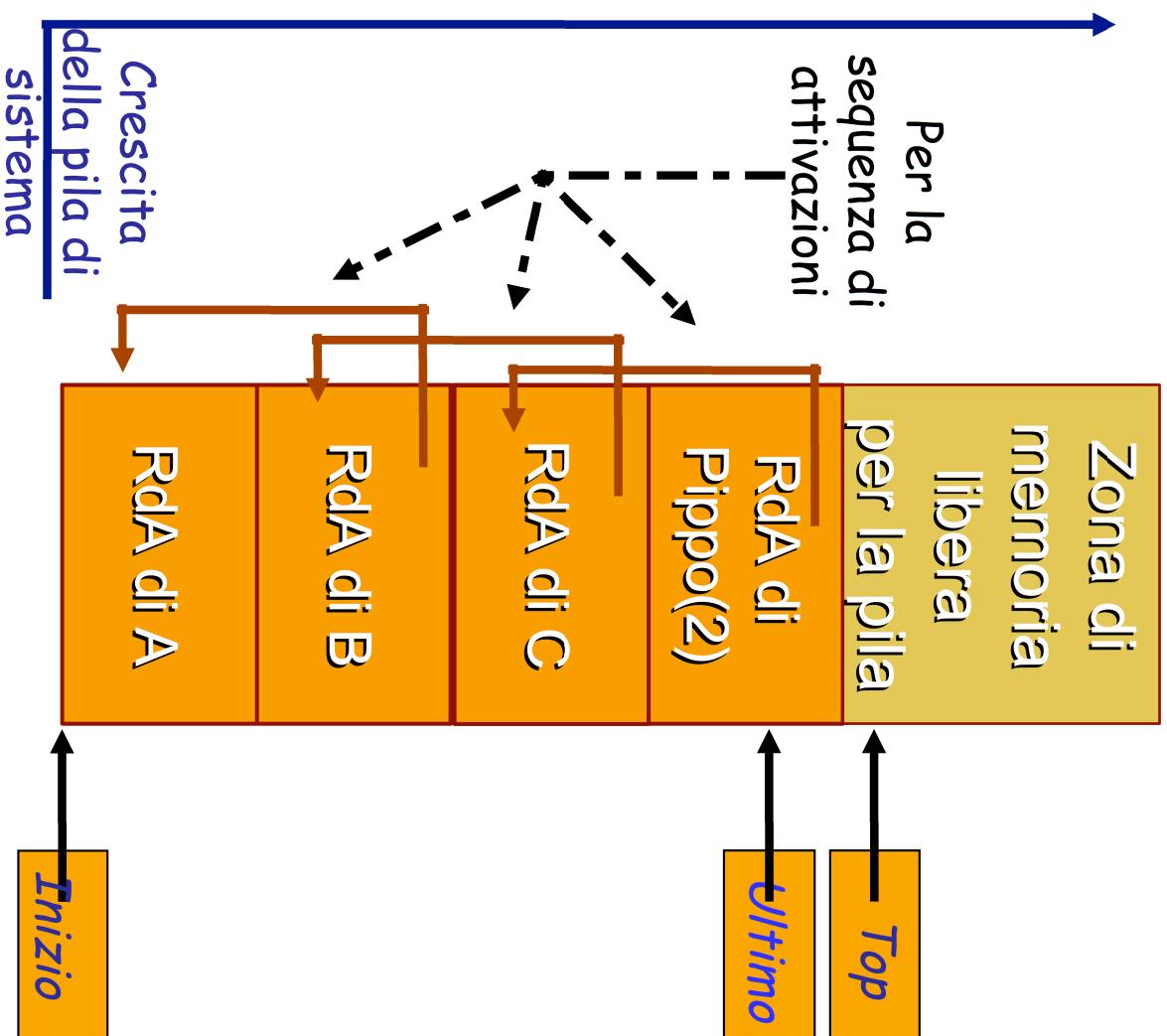
Nel Rda del chiamante.
Solo nelle
funzioni.

Parametri attuali

Disposizione dei campi
dipendente dalle
implementazioni

Esempio: indirizzo del risultato

```
A: { int y=0;  
      ...  
    }  
  
B: { int x=0;  
      int Pippo(int n) {  
        x=n+1;  
        y=n+2;  
      }  
      ...  
    }  
  
C:{ int x=1;  
  x=Pippo(2);  
  write(x);  
}  
write(x);  
}
```



Record di attivazione: sottoprogrammi

Associati ad ogni attivazione di sottoprogramma

(conseguenza di chiamata - dinamicamente)

Non alla definizione della procedura stessa (staticamente)

Sottoprogramma p

Puntatore di Catena Dinamica

Puntatore Catena Statica

Indirizzo di Ritorno

Indirizzo del Risultato

Parametri

Variabili Locali

Risultati Intermedi

Per le regole di scope statico

Prima istruzione da eseguire
dopo il sottoprogramma

Nel Rda del chiamante.
Solo nelle
funzioni.

Parametri attuali

Disposizione dei campi
dipendente dalle
implementazioni

Record di attivazione: sottoprogrammi

come recuperare le informazioni dei Rda

Risultati Intermedi

Puntatore di Catena Dinamica

Puntatore Catena Statica

Indirizzo di Ritorno

Indirizzo del Risultato

Parametri

Variabili Locali

Risultati Intermedi

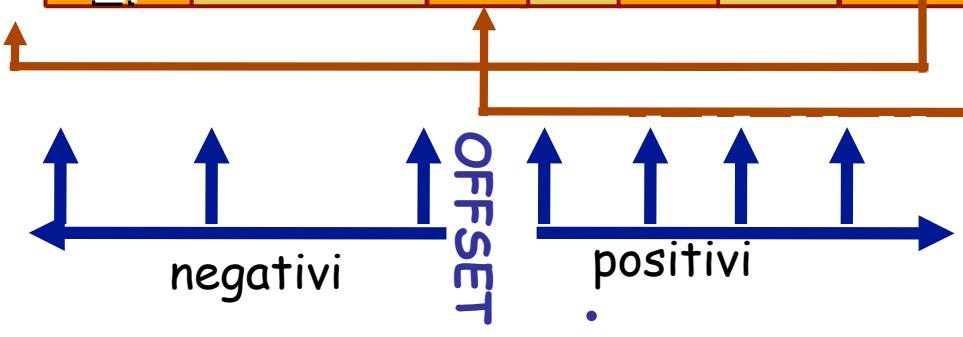
Osservazioni

- Si usano offset per individuare i vari campi

- Non compaiono identificatori di variabili locali: il compilatore calcola gli indirizzi relativi rispetto ad una posizione fissa del Rda in cui le variabili sono definite (offset)

Ottimizzazione del codice prodotto

- Es.: salvataggio di informazioni nei registri anziché nel Rda



Record di attivazione

anche per linguaggi senza ricorsione

per ottimizzare l'occupazione della memoria

Esempio di spreco di memoria nella gestione interamente statica

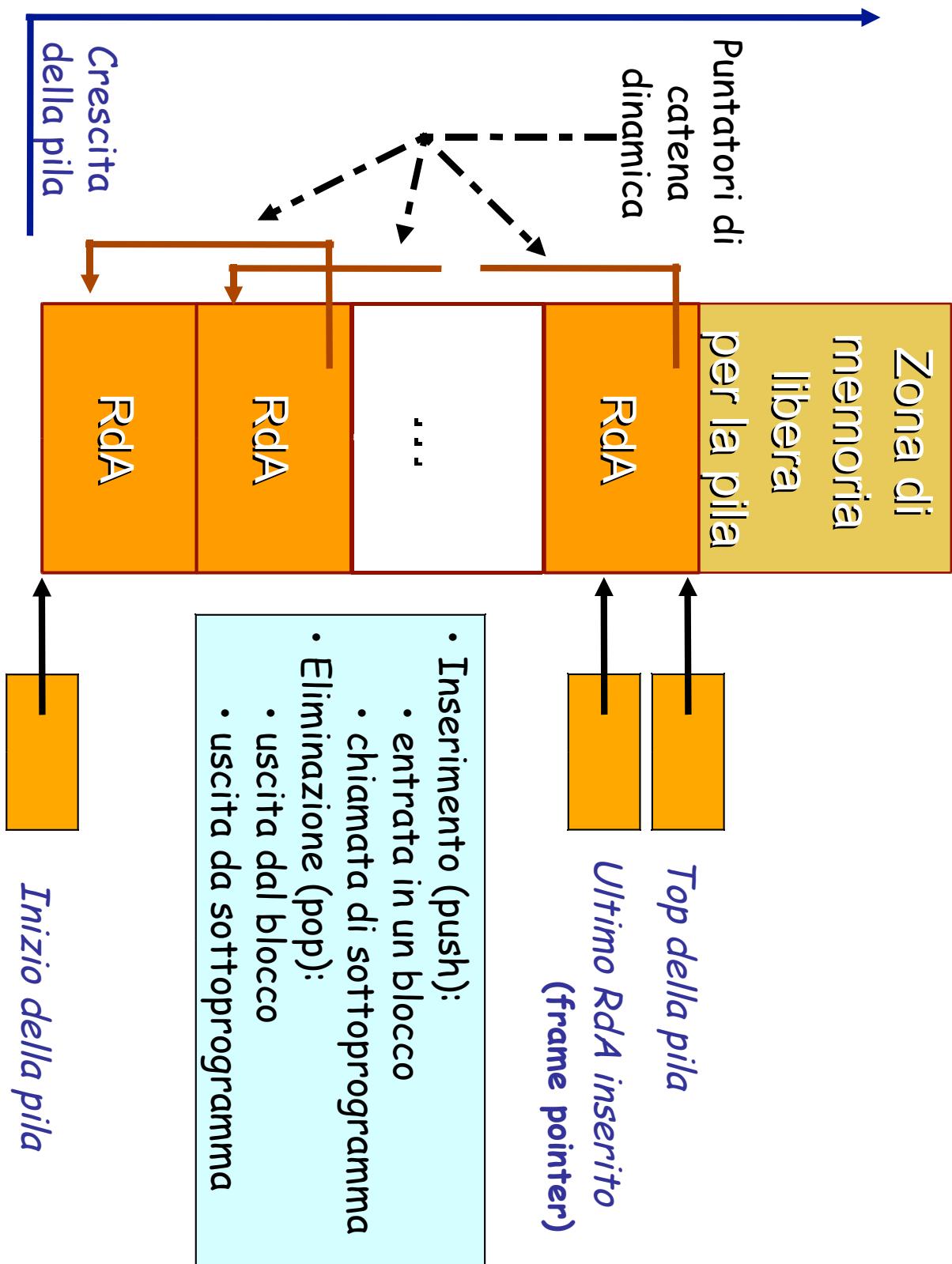
Un programma in cui esistono chiamate solo ad alcune delle procedure definite
(n° procedure chiamate < n° procedure definite)

Perchè:

RdA associati ad ogni attivazione di sottoprogramma
(conseguenza di chiamata - dinamicamente)
Non alla definizione della procedura stessa (staticamente)

Meglio di una politica non interamente di tipo statico

Gestione della pila di run-time



Gestione della pila a run-time

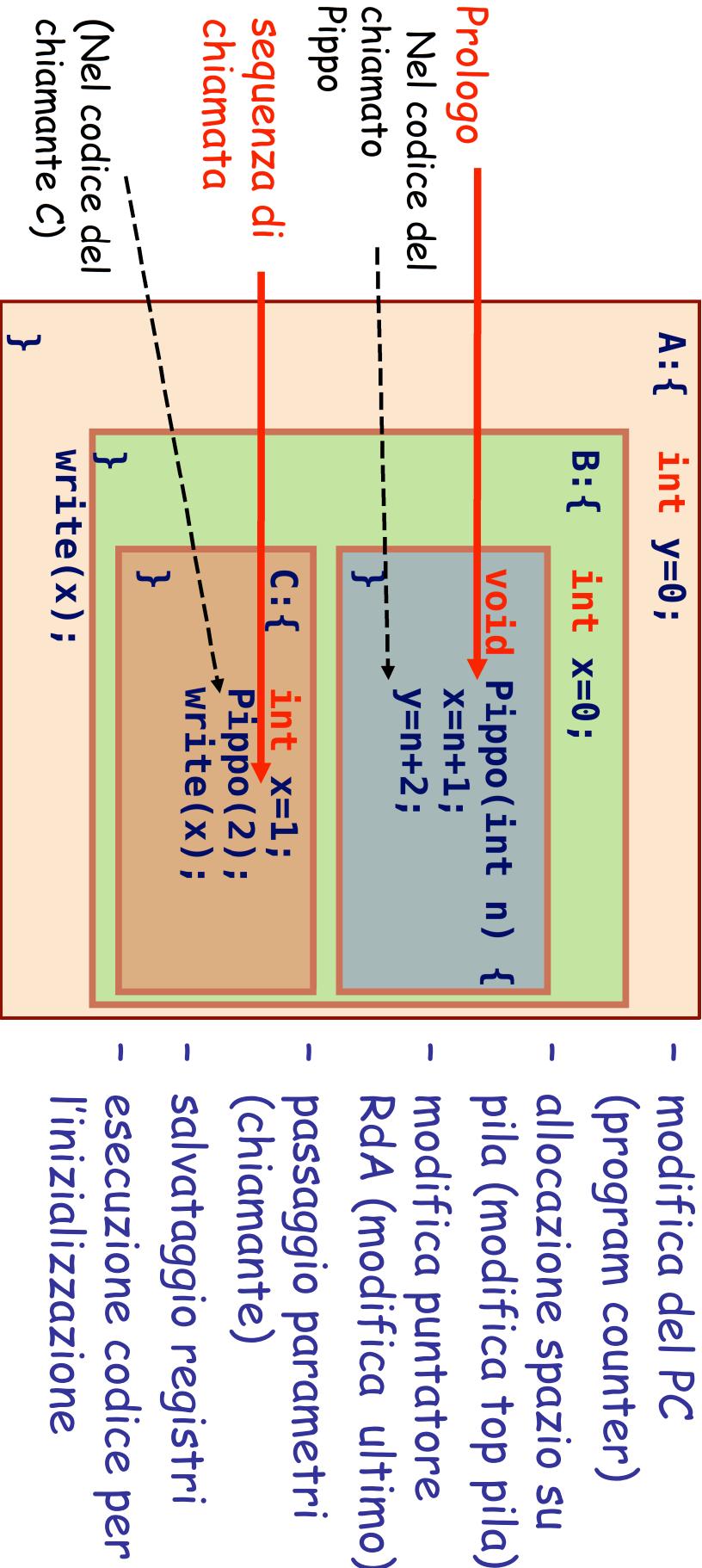
Il/l' compilatore/interprete inserisce frammenti di codice prima e dopo (la chiamata a) il sottoprogramma

- nel chiamante (o blocco esterno):
 - sequenza di chiamata: codice aggiunto
 - eseguita, in parte, prima della chiamata (entrata)
 - eseguita, in parte, dopo il ritorno dalla chiamata (uscita)
- nel chiamato (o blocco interno):
 - prologo
 - eseguito subito dopo la chiamata (entra)
 - epilogo
 - eseguito subito prima del ritorno dalla chiamata (uscita)

I frammenti divisi in dipendenza del compilatore e dalla specifica implementazione: meglio confinare attività il più possibile al chiamato (codice aggiunto una sola volta)

Gestione della pila a run-time

Il/l' compilatore/interprete inserisce frammenti di codice prima e dopo (la chiamata a) il sottoprogramma



I frammenti divisi in dipendenza del compilatore e dalla specifica implementazione: meglio confinare attività il più possibile al chiamato

Gestione della pila:

azioni della sequenza di chiamata e del prologo

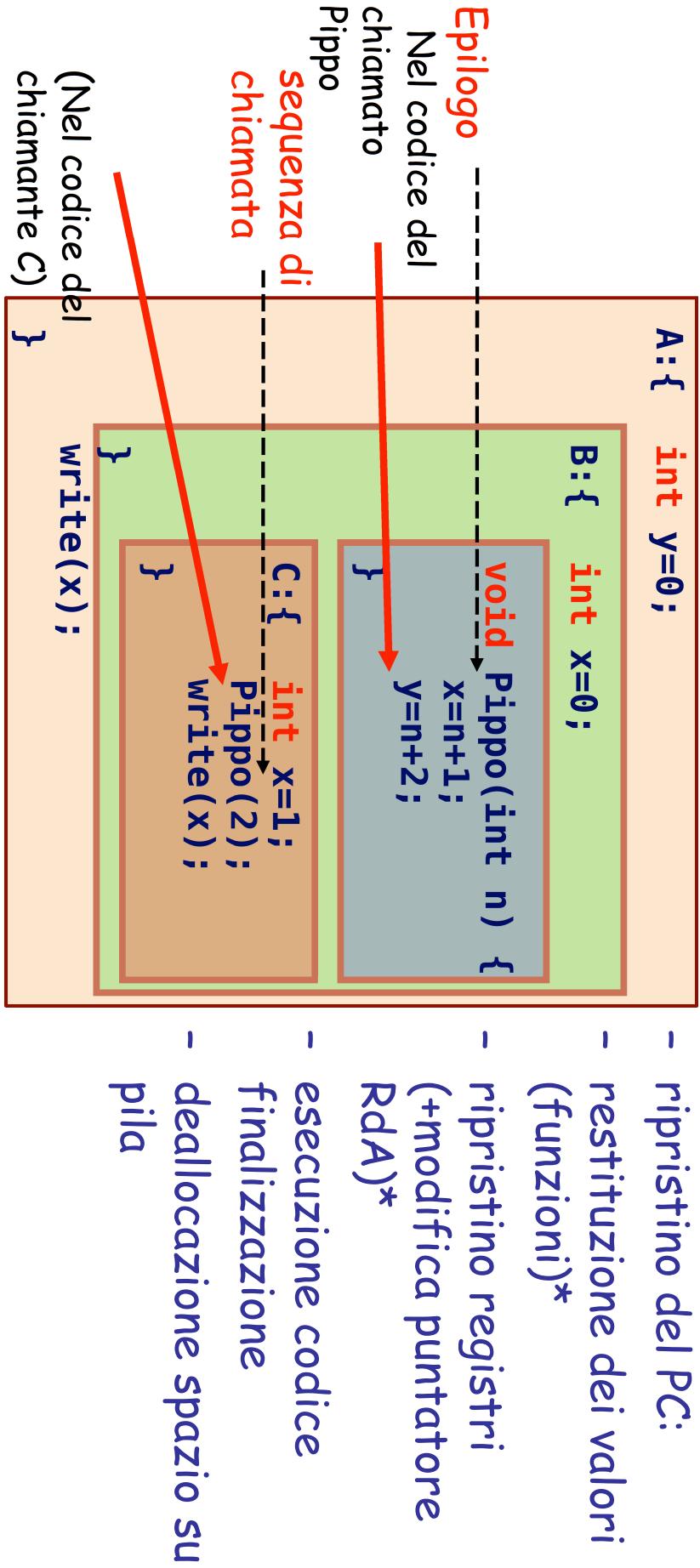
- modifica del PC (**program counter**)
 - Affinchè punti alla prima istruzione del corpo del sottoprogramma
 - Con salvataggio del valore originale (indirizzo di ritorno)
- **allocazione spazio su pila**
 - Spazio per il nuovo record di attivazione
 - Con modifica del puntatore alla zona di memoria libera
- **modifica puntatore Rda**
 - Affinchè punti al nuovo Rda inserito, relativo al sottoprogramma chiamato

Gestione della pila (cont.): azioni della sequenza di chiamata e del prologo

- passaggio parametri
 - Compito del chiamante (diverse chiamate-diversi parametri)
- salvataggio registri
 - Valori per la gestione del controllo : il puntatore al vecchio RdA → puntatore della catena dinamica
- esecuzione codice per l'inizializzazione:
 - Costrutti esplicativi per inizializzare alcuni elementi del RdA (in dip. Dei LdP)

Gestione della pila a run-time

I/l' compilatore/interprete inserisce frammenti di codice prima e dopo (la chiamata a) il sottoprogramma



I frammenti divisi in dipendenza del compilatore e dalla specifica implementazione: meglio confinare attività il più possibile al chiamato

Gestione della pila (cont.): azioni del ritorno di controllo di chiamate e dell'epilogo

- ripristino del PC:
 - Per restituire il controllo al chiamante
- restituzione dei valori (funzioni)
 - valori di ritorno (funzioni)/informazioni dal chiamato al chiamante → memorizzati nell'apposita zona del RdA del chiamante (reperibile dal chiamto - catena dinamica)*
- ripristino registri (+modifica puntatore RdA)*
 - Vecchi valori dei registri
 - Puntatore al RdA precedente
- esecuzione codice finalizzazione
 - Prima che siano distrutti gli oggetti locali (in dip. Dei LdP)
- deallocazione spazio su pila
 - Spazio per il nuovo record di attivazione
 - Con modifica del puntatore alla zona di memoria libera

Gestione dinamica con heap

allocazione esplicita della memoria in C

```
int *p, *q; /* puntatori ad intero*/
p = malloc (sizeof (int)); /* alloca memoria puntata da p*/
q = malloc (sizeof(int)); /* alloca memoria puntata da q*/
*p = 0; /* differenzia e assegna */
*q = 0; /* differenzia e assegna */
free(p); /* dealloca memoria */
free(q); /* dealloca memoria */
```

Gradi di libertà nell'ordine di allocazione e deallocazione



IMPOSSIBILE gestione LIFO

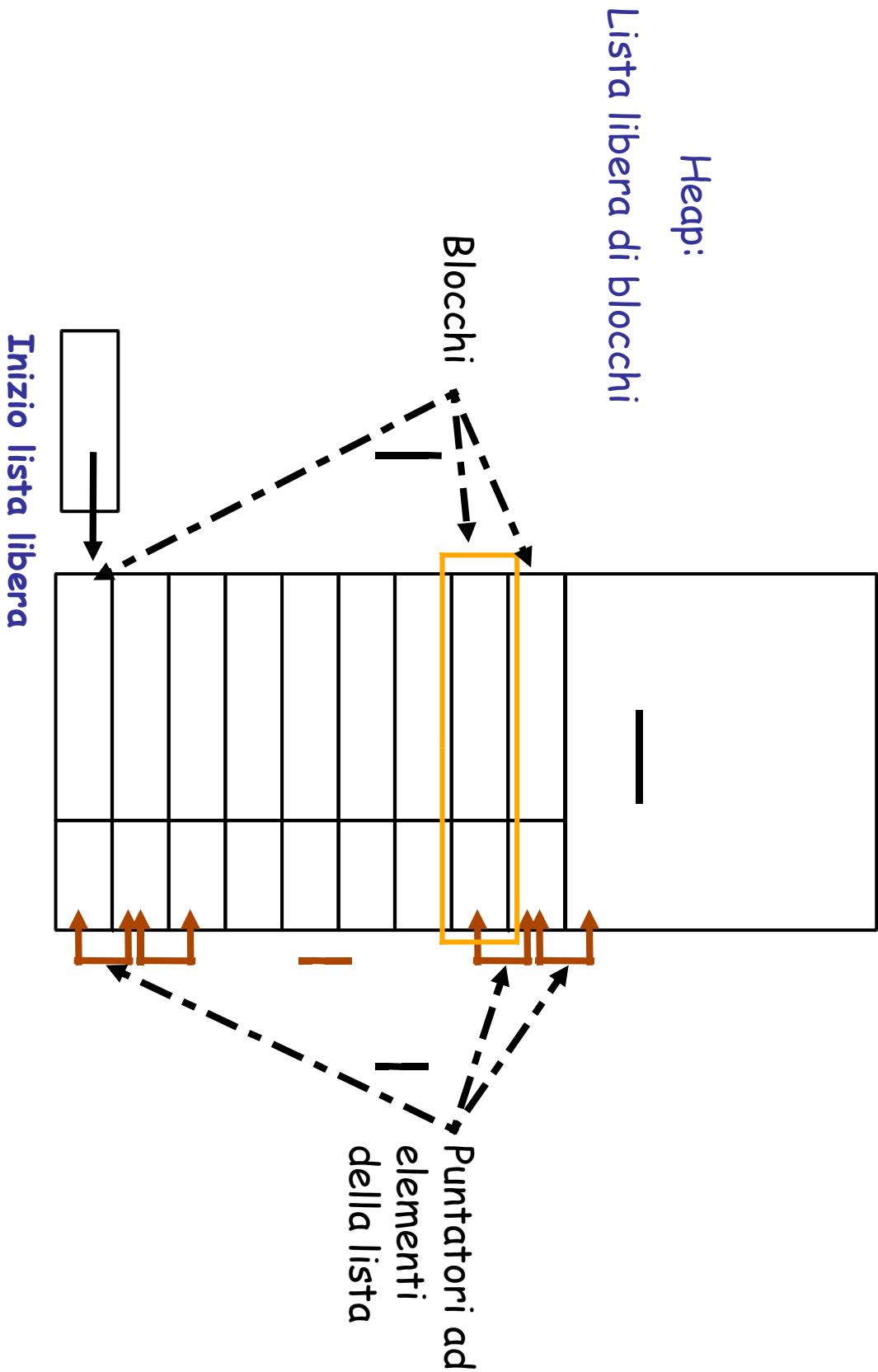
Gestione dinamica con heap

per ling. con allocazione esplicita della memoria

- Esempi:
 - C, C++, Pascal, ...
- **HEAP: altra zona di memoria diversa dallo stack**
(per allocazioni e deallocazioni in tempi arbitrari)
 - gestito con blocchi di dim. fissa
 - gestito con blocchi di dim. variabile

Gestione dinamica con heap

Heap suddiviso in blocchi di dimensione fissa e limitata



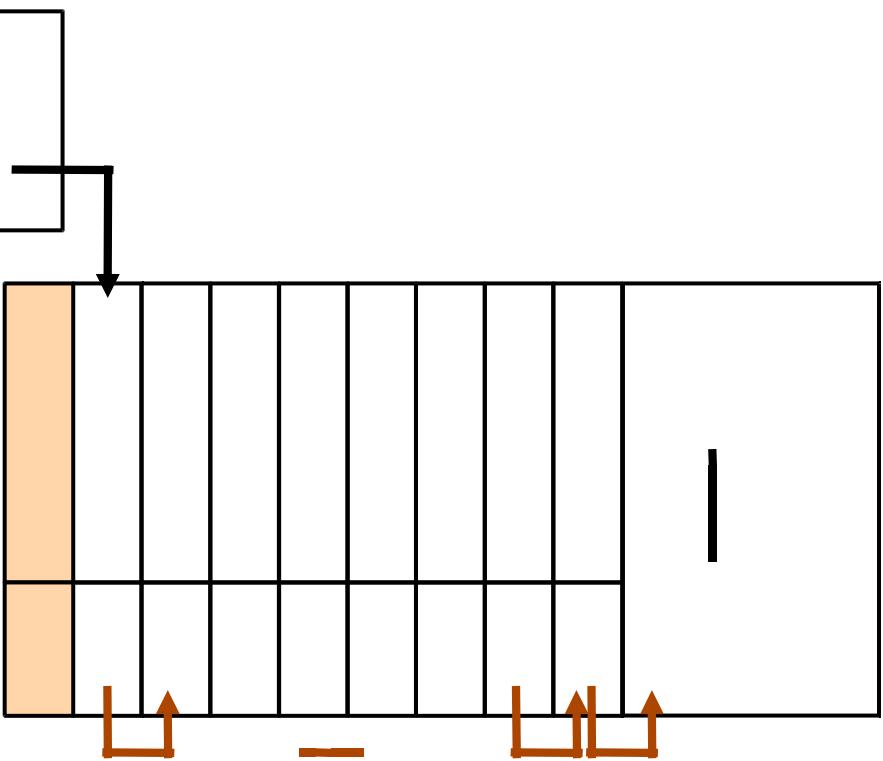
Gestione dinamica con heap (cont.)

Heap suddiviso in blocchi di dimensione fissa e limitata

```
int *p, *q;  
p = malloc (sizeof (int));  
q = malloc (sizeof(int));  
*p = 0;  
*q = 0;  
free(p);  
free(q);
```

Allocazione

- Il primo elemento della lista libera viene rimosso
- Il puntatore a tale elemento viene restituito all'op. che richiede la memoria
- Si aggiorna il puntatore al primo elemento della lista libera all'elemento successivo



Inizio lista libera
Blocchi allocati sullo heap

Gestione dinamica con heap (cont.)

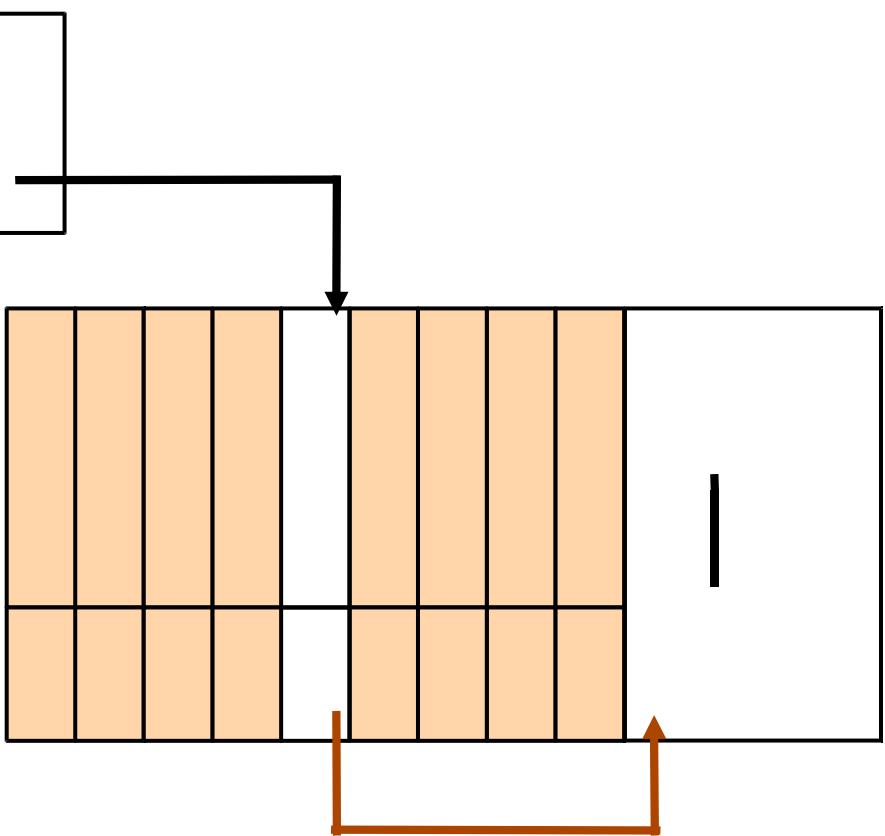
Heap suddiviso in blocchi di dimensione fissa e limitata

```
int *p, *q;  
p = malloc (sizeof (int));  
q = malloc (sizeof(int));  
*p = 0;  
*q = 0;  
free(p);  
free(q);
```

Deallocazione

- il blocco liberato viene collegato in testa alla lista libera
- si aggiorna il puntatore all'elemento successivo a quello aggiunto

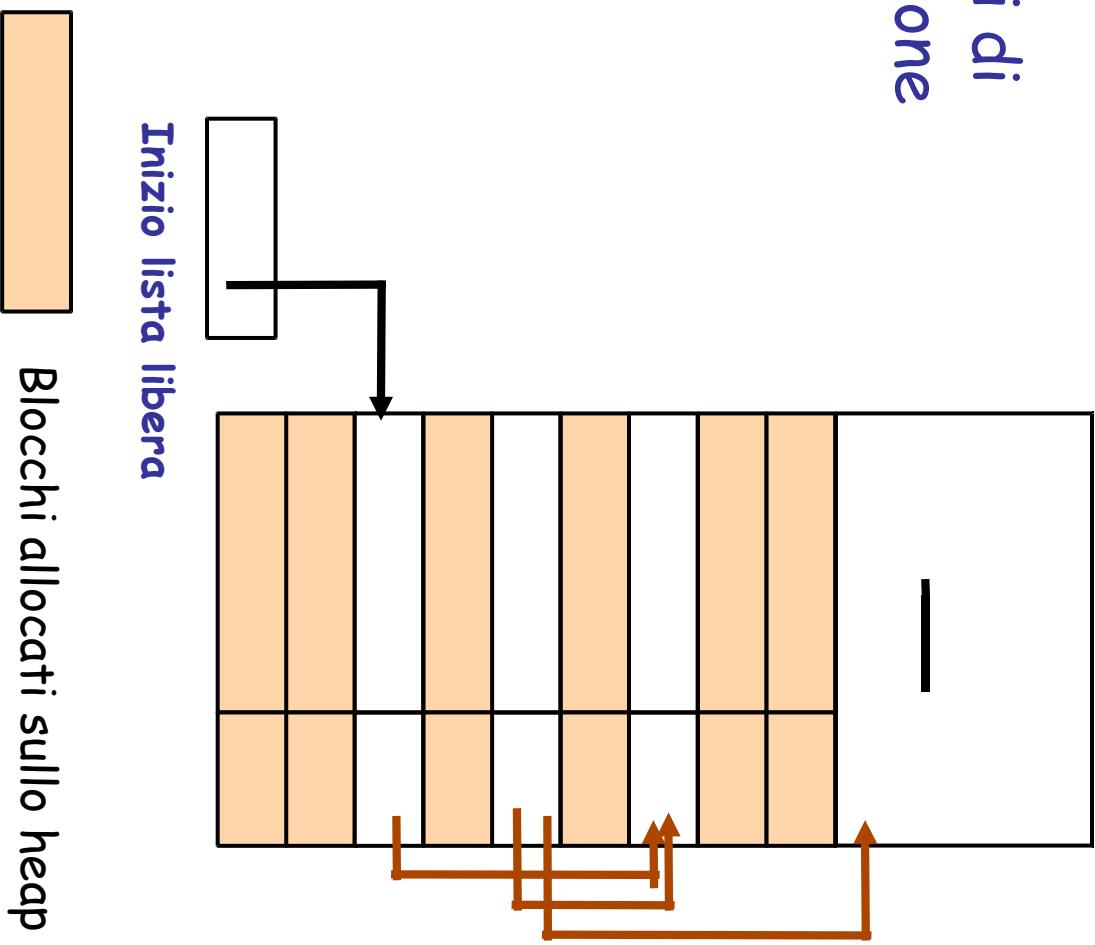
Inizio lista libera



Gestione dinamica con heap (cont.)

Heap suddiviso in blocchi di dimensione fissa e limitata

Dopo alcune operazioni di
allocazione e deallocazione



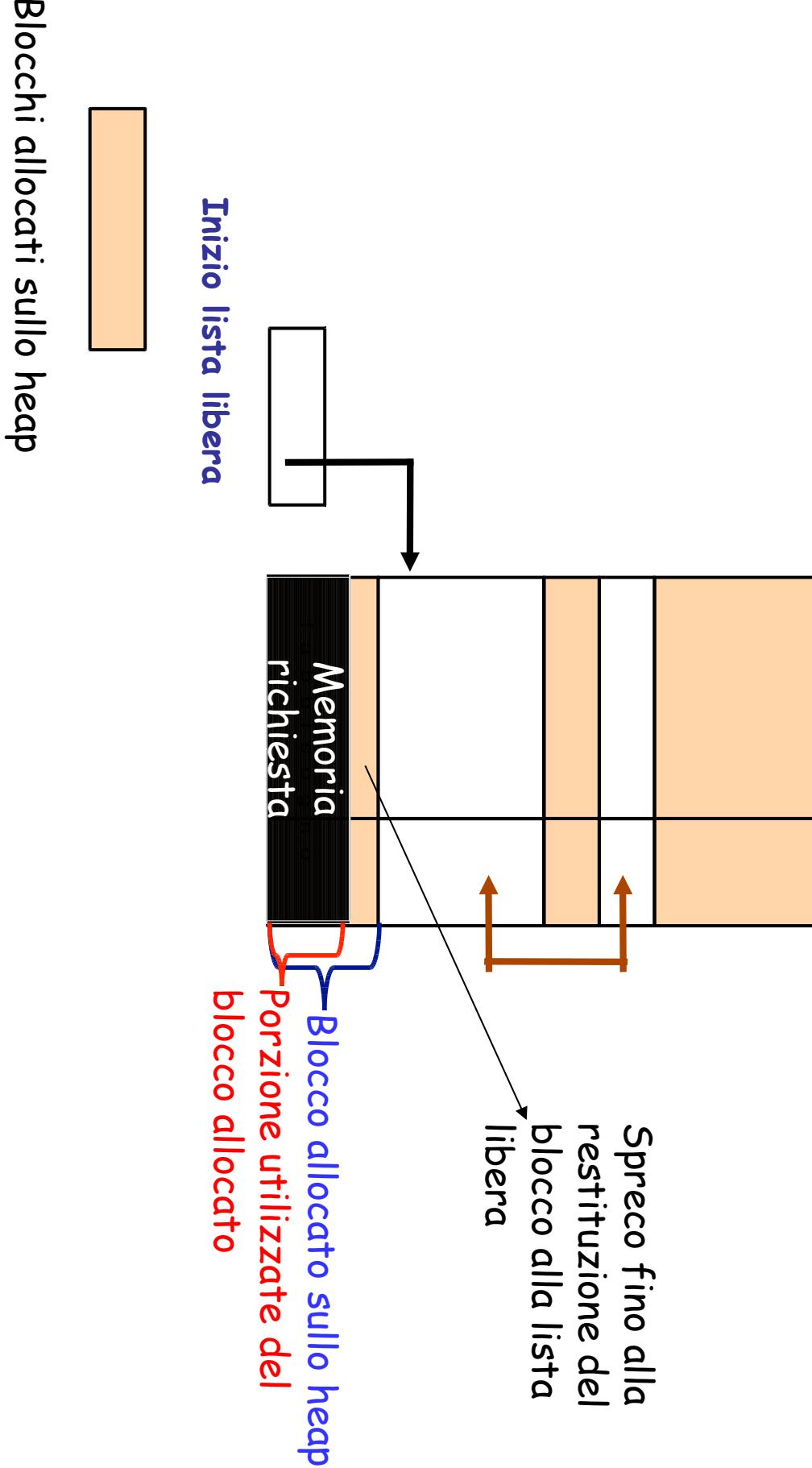
Gestione dinamica con heap (cont.)

Heap suddiviso in blocchi di dimensione variabile

- dim. fissa: inadeguatezza per strutture di dimensione variabile a run-time
 - es. vettore di dim. Variabili (locazioni consecutive)
- Dim. Variabile: gestione tesa ad ottimizzare
 - (con un compromesso)
 - l'occupazione della memoria
 - Evitare frammentazione
 - interna: porzione inutilizzata di blocco allocato (dimensione necessaria di poco maggiore di quella di un blocco)
 - esterna: memoria libera sufficiente nel complesso, ma non in un unico blocco
 - » necessità di ricompattamento
 - l'efficienza della gestione dello heap (op. A run time)
 - basso numero di operazioni aggiuntive richieste (overhead)

Frammentazione interna

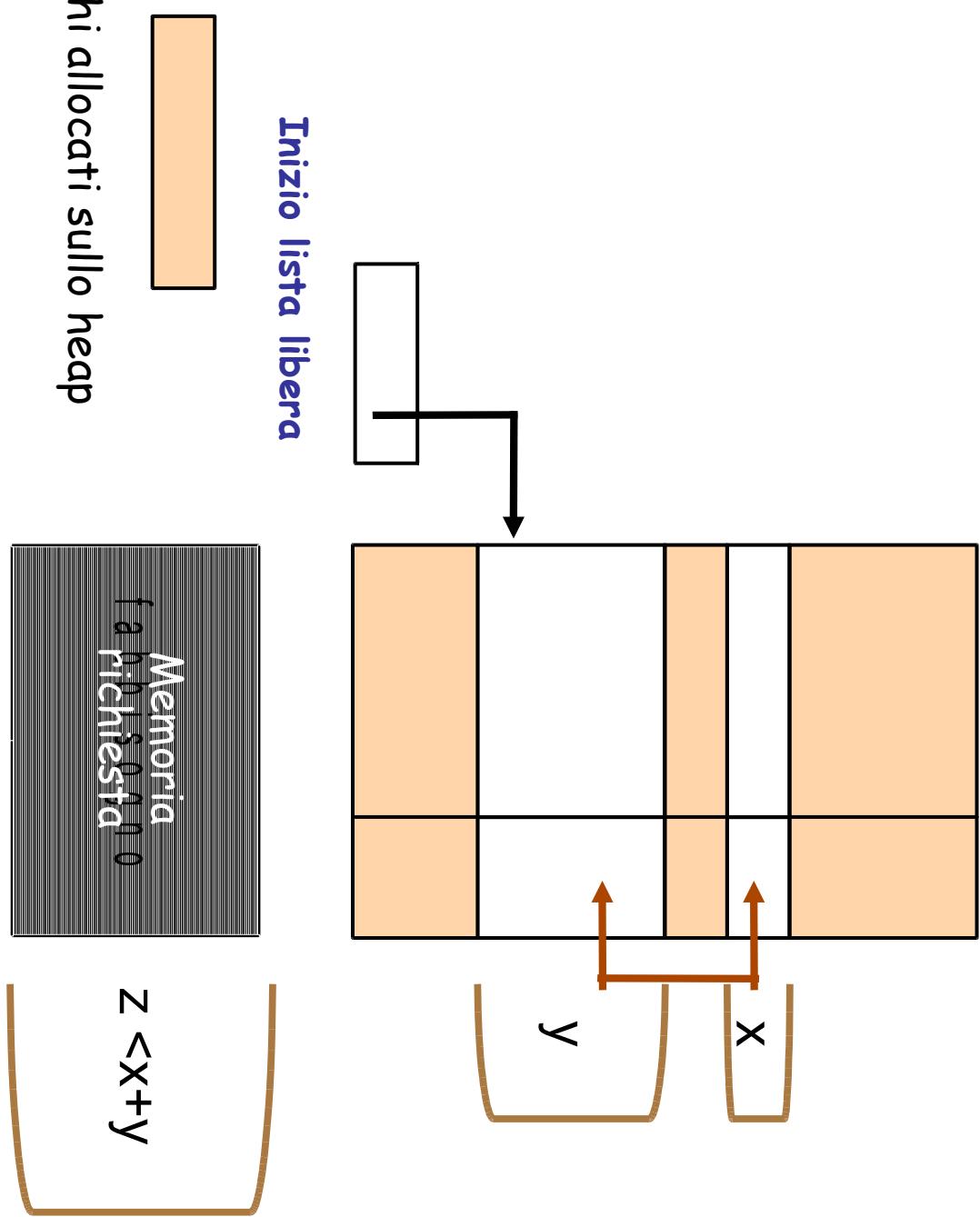
(porzione inutilizzata di blocco allocato)



Blocchi allocati sullo heap

Frammentazione esterna

(la memoria richiesta non è allocabile)



Heap suddiviso in blocchi di dim. Variabile

(POSSIBILITA' 1: lista libera unica)

Inizialmente con un unico blocco
pari a tutta la mem. disponibile (heap)

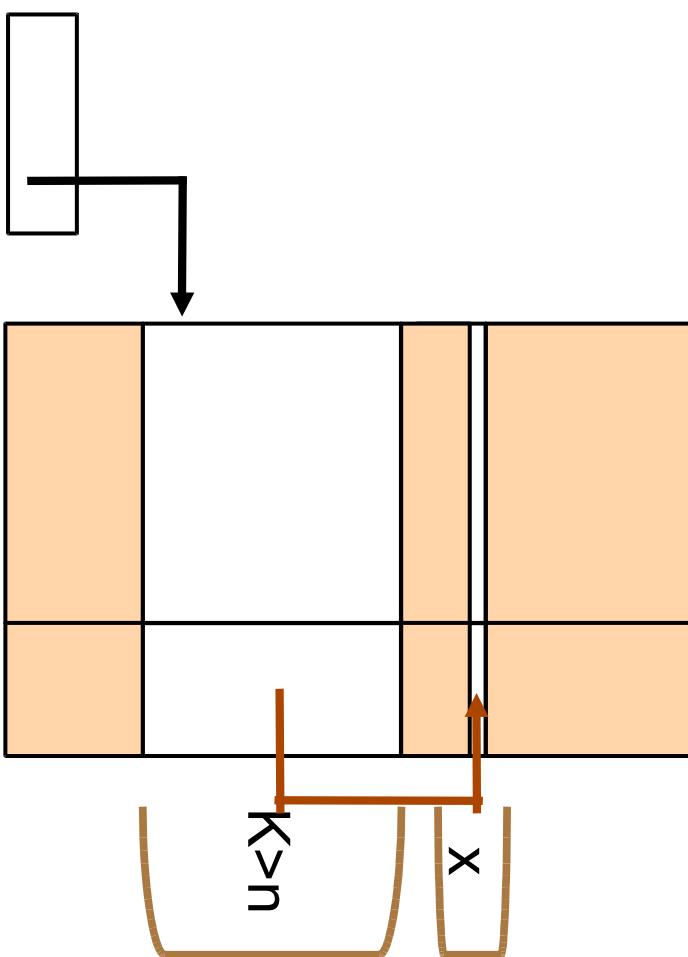
- **Allocazione di n parole:** soddisfatta usando le prime n parole allocabili
 - avanza il puntatore della lista libera di n
- **Successive allocazioni:** analogamente
- **Spazio dello Heap terminato:**
 - utilizzare lo spazio deallocated. Due modi:
 - **Utilizzo diretto della lista libera**
 - **Compattazione della memoria libera**

Utilizzo diretto della lista libera
si mantiene una lista di blocchi liberi di dim. variabile

Ma quando termina lo spazio dello Heap

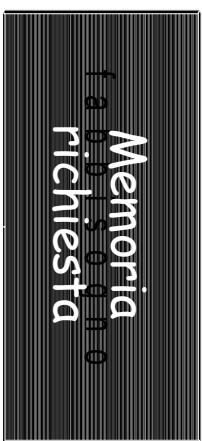
Si cerca un blocco di
dimensione k adeguato
(sufficientemente
grande)

$k > n$



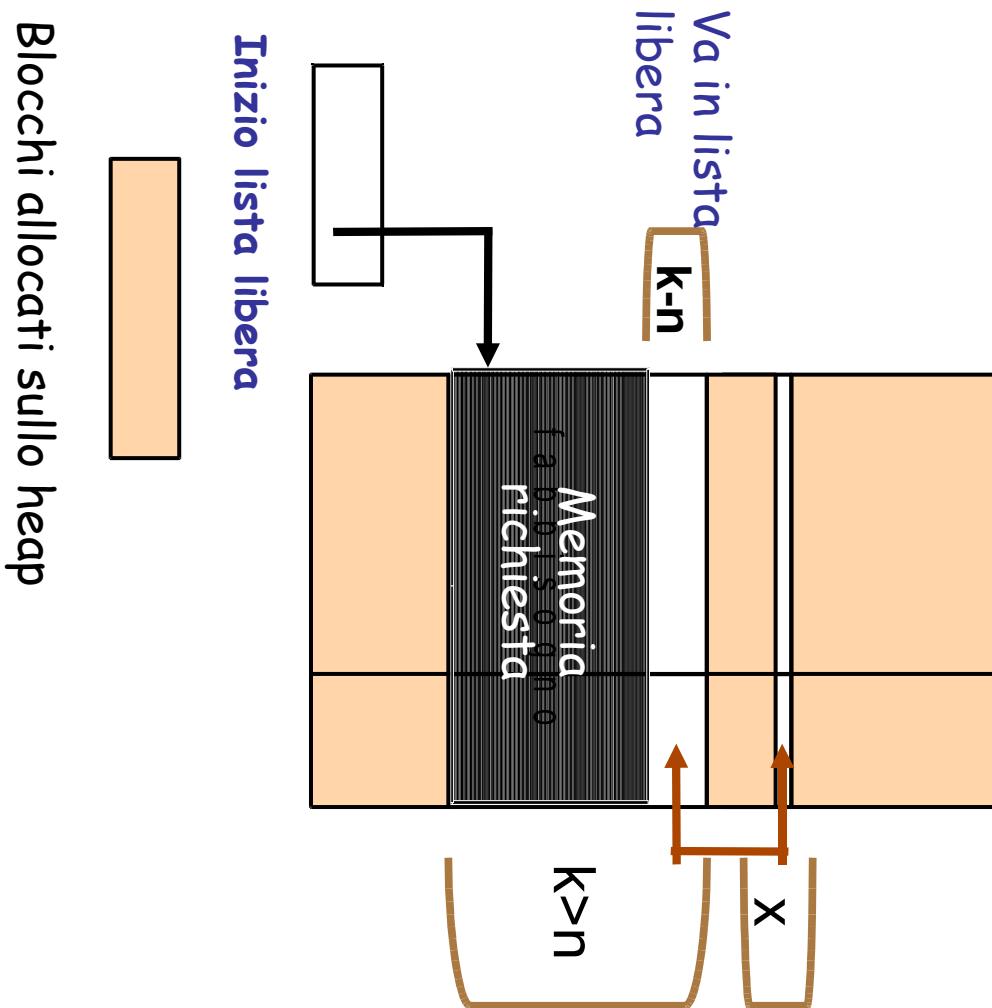
Inizio lista libera

Blocchi allocati sullo heap



Utilizzo diretto della lista libera
si mantiene una lista di blocchi liberi di dim. variabile

Ma quando termina lo spazio dello Heap

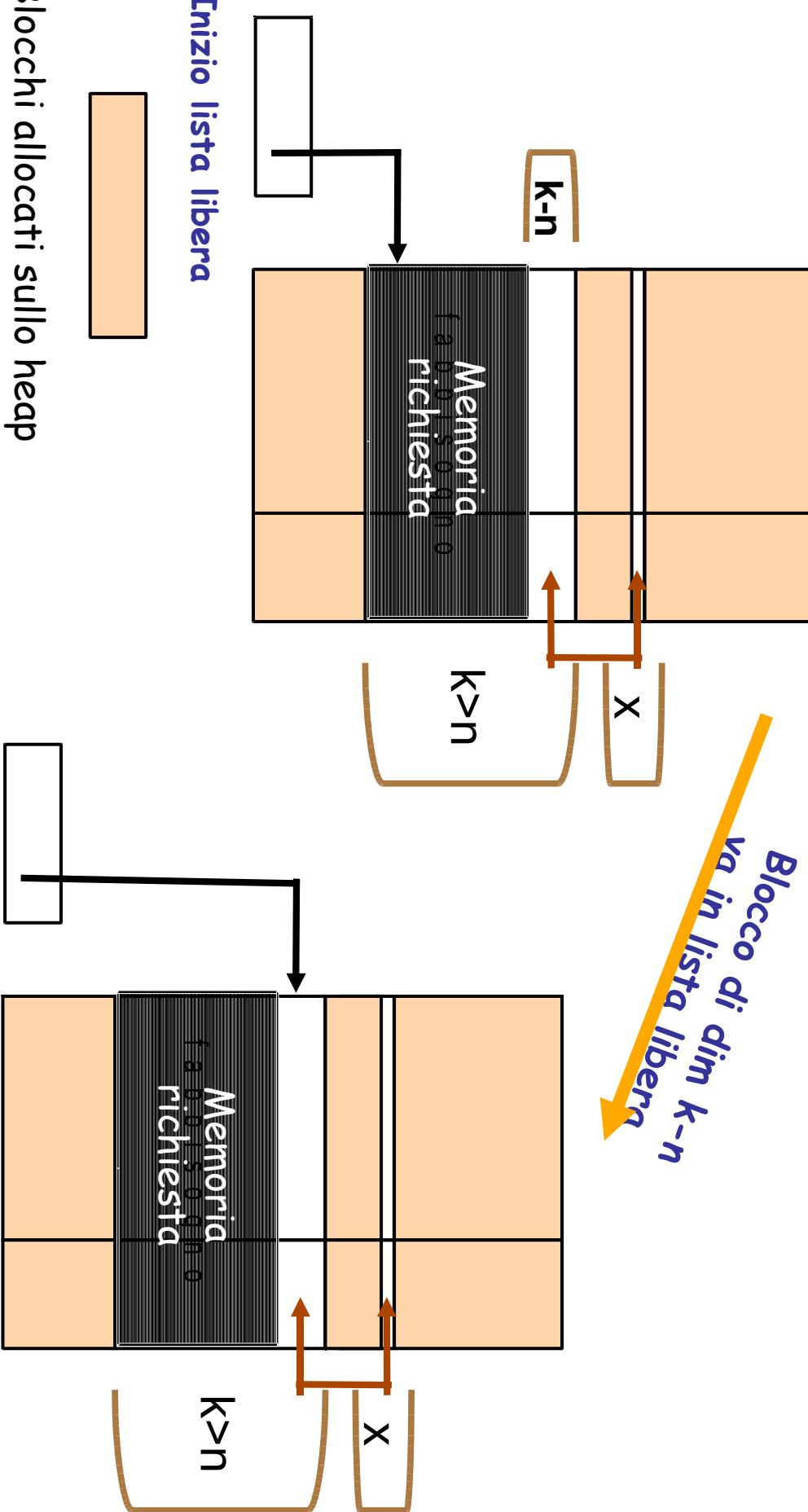


Blocchi allocati sullo heap

**Utilizzo diretto della lista libera
si mantiene una lista di blocchi liberi di dim. variabile**

Ma quando termina lo spazio dello Heap

Blocco va in lista dim k-n



**Utilizzo diretto della lista libera
si mantiene una lista di blocchi liberi di dim. variabile**

Ricerca del blocco di dimensione adeguata

- **Politica first-fit:** Il primo blocco di memoria di dimensione sufficiente (migliore efficienza)
- **Politica best-fit:** Il più piccolo blocco di memoria di dimensione sufficiente (migliore occupazione)

costo:

lineare con il numero di blocchi liberi

(se si mantengono I blocchi ordinati per lunghezza le due tecniche coincidono)

**Utilizzo diretto della lista libera
si mantiene una lista di blocchi liberi di dim. variabile**

- **deallocazione:** si ricompatta il blocco con i blocchi liberi adiacenti (solo quelli):
 - Costo costante: inserimento diretto
 - Costo lineare: inserimento in lista ordinata per dimensione crescente

Compattazione della memoria libera

si mantiene una lista di blocchi liberi di dim. variabile

Ma quando termina lo spazio dello Heap

- Allocazione come prima
- **in condizione di fine spazio direttamente allocabile:**
 - spostamento dei blocchi attivi (deallocati&non restituiti alla L.L.)
 - compattamento dei blocchi liberi (Tutti non solo quelli adiacenti) in un'estremità dello Heap
- Spostamento del puntatore allo Heap
- Possibile se i blocchi di memoria sono spostabili (es. Blocchii non spostabili: blocchi i cui indirizzi sono memorizzati sulla pila)

Heap suddiviso in blocchi di dim. Variabile

(POSSIBILITA' 2: liste libere multiple)

Molte liste di dimensioni diverse

- **Richiesta: spazio di allocazione di dimensione n**
 - selezione della lista con blocchi di dim. maggiori o uguali a n
 - allocazione (ftramn. interna se il blocco allocabile ha dim. Maggiori di n)

Dimensione dei blocchi nelle liste

Due possibilità

-
- 1: Buddy system
(dimensione dei blocchi:
Potenze di 2)
 - 2: Fibonacci Heap
(dimensione dei blocchi:
numeri di Fibonacci)

Heap suddiviso in blocchi di dim. Variabile **(liste libere multiple: Molte liste di dimensioni diverse** **(1) Buddy system**

(dimensione dei blocchi: potenze di 2)

Allocazione: spazio richiesto di dimensione n

- k più piccolo intero tale che $2^k \geq n$



Spazio da ricercare per l'allocazione:

dimensione 2^k nella opportuna lista libera

Non esiste

Spazio da ricercare per l'allocazione:

dimensione 2^{k+1} nella opportuna lista libera



Spazio trovato diviso in due:

dimensione $2^{k+1} / 2 = 2^k$



Deallocazione di un blocco risultante da divisione:

Si ricerca il suo compagno (buddy). Se libero, i blocchi riuniti (dim 2^{k+1})

Heap suddiviso in blocchi di dim. Variabile

(**liste libere multiple**: Molte liste di dimensioni diverse)

(2) Heap di Fibonacci

(dimensione dei blocchi: numeri di fibonacci)

Ricordate che i numeri di fibonacci sono i termini della successione

$$\text{Fib}(0)=\text{Fib}(1)=1$$

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2), \text{ per } n>1$$

Allocazione e deallocazione:

Analogia al metodo del Buddy System

Ma

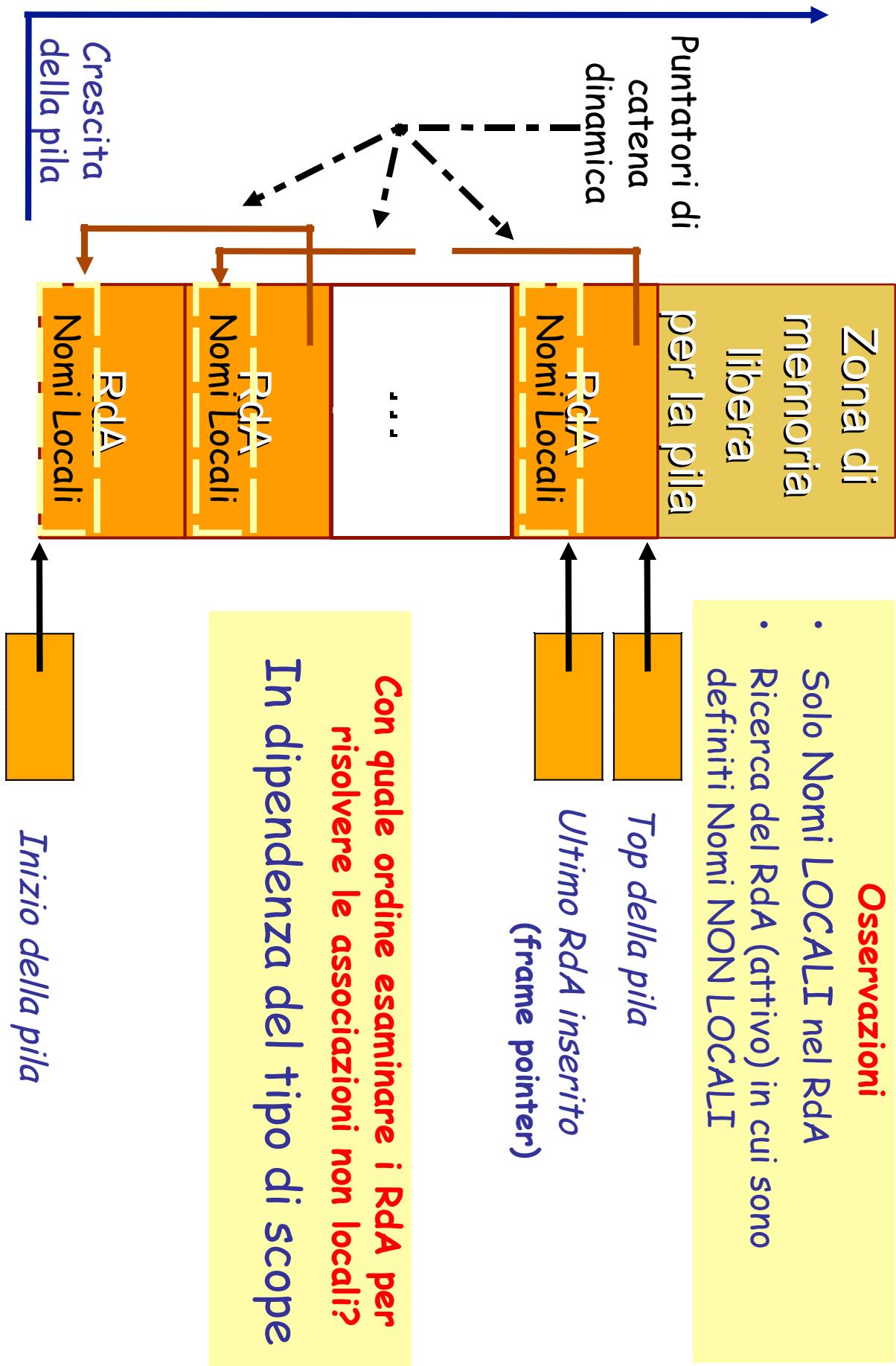
La successione cresce più lentamente della successione 2^n

(minore frammentazione interna)

Implementazione delle regole di Scope

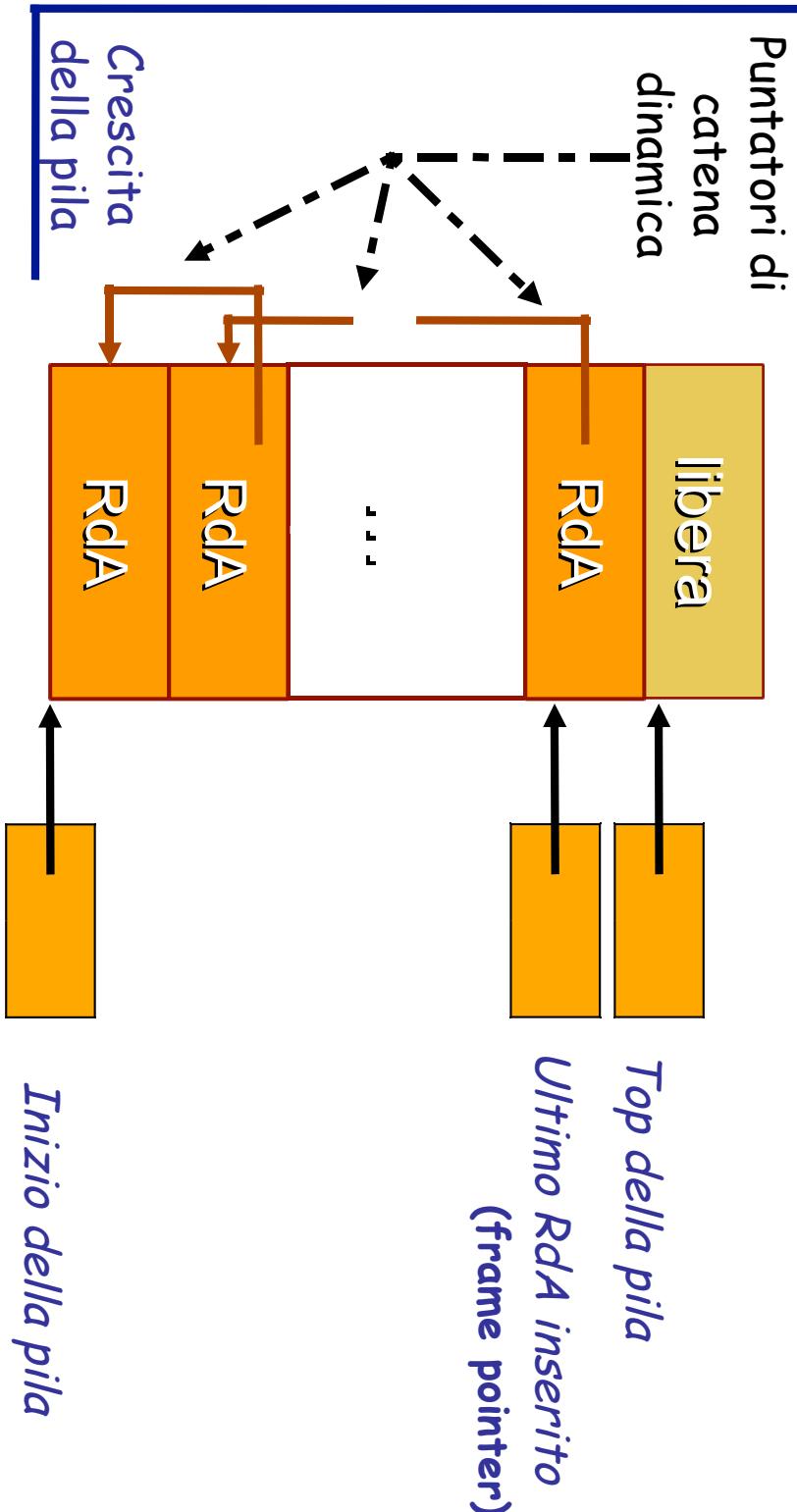
Le strutture richieste e
la loro gestione

Gestione della pila di run-time



Implementazione dello scope statico: catena statica

ordine di esplorazione dei RdA non è quello dei blocchi nella pila
(vedi puntatori di catena dinamica)



Implementazione dello scope statico (cont.)

catena statica (cont.)

// RdA contiene spazio per i soli nomi locali
Associazioni Nomi non locali → esplorare gli altri RdA

Puntatore di catena DINAMICA

A:{ int y=0;

B:{ int x=0;

void Pippo(int n) {

x=n+1;

y=n+2;

}

C:{ int x=1;

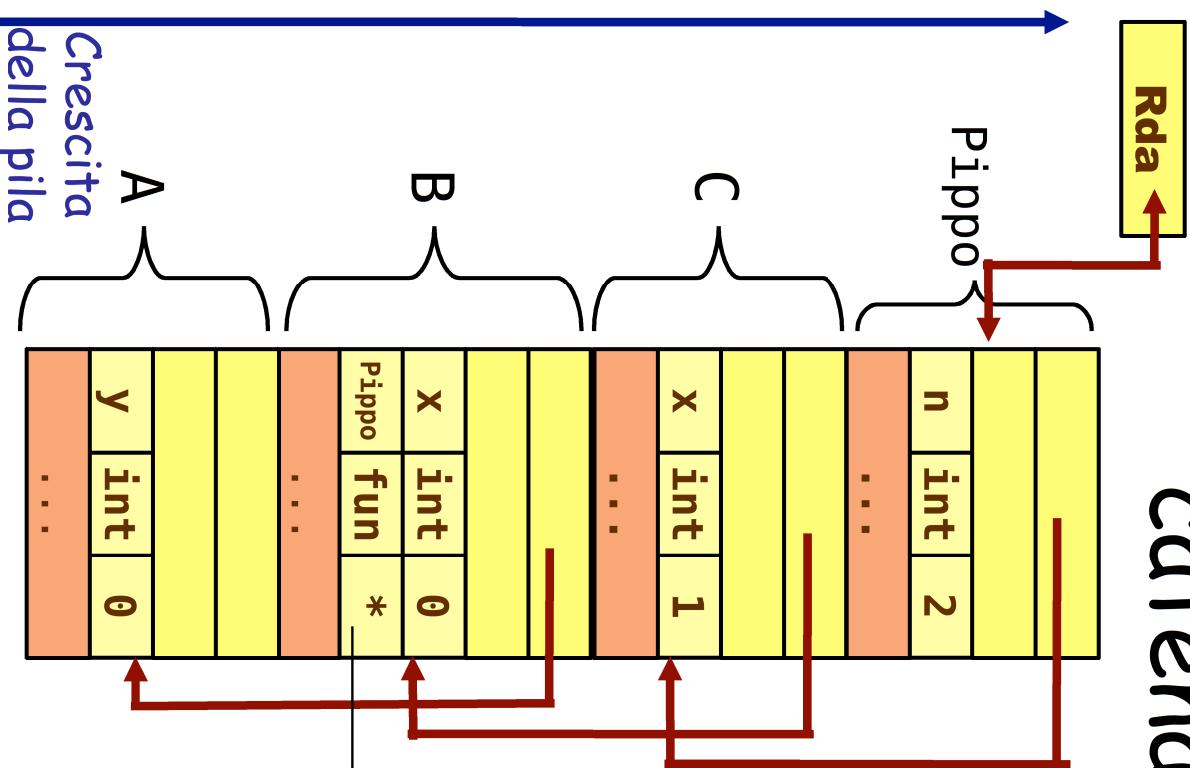
Pippo(2);

write(x);

}

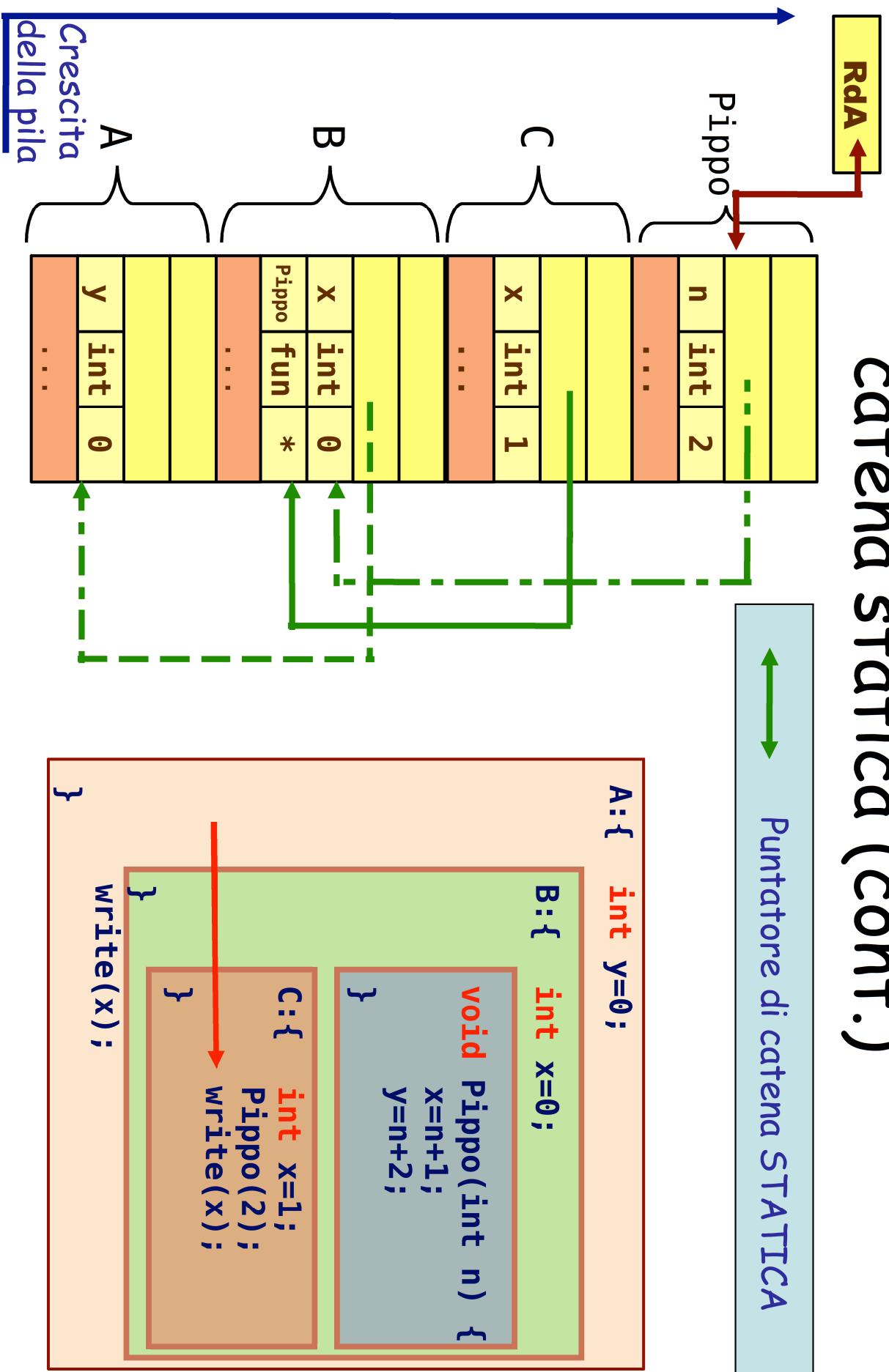
write(x);

Crescita della pila



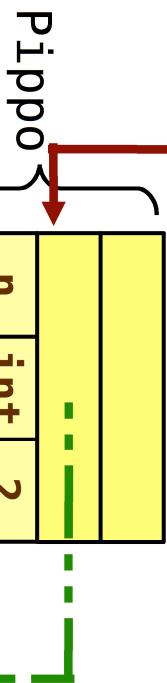
Implementazione dello scope statico:

catena statica (cont.)



Implementazione dello scope statico (cont.)

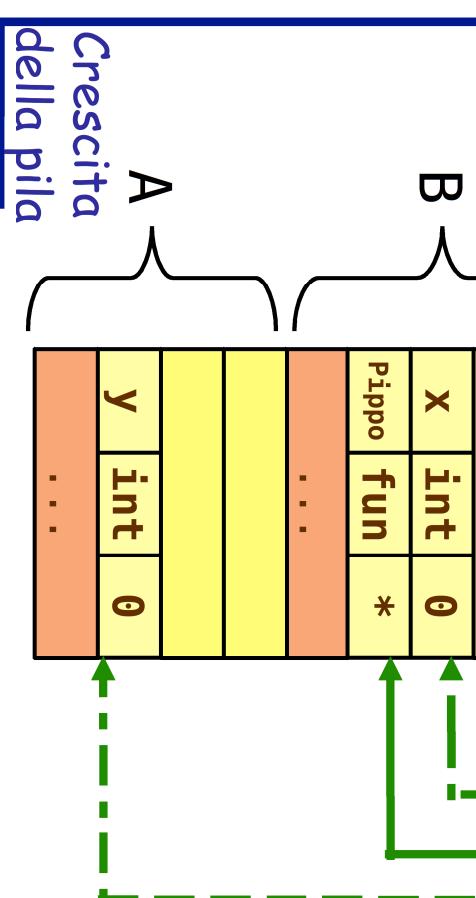
RdA



Pippo

C

B



secondo regole di scope statico:
Pippo è un blocco annidato in B →
La x usata in Pippo (blocco C) e quella
non locale definita in B

A:{ **int** y=0;

B:{ **int** x=0;

void Pippo(**int** n) {

x=n+1;

y=n+2;

}

C:{ **int** x=1;

Pippo(2);

write(x);

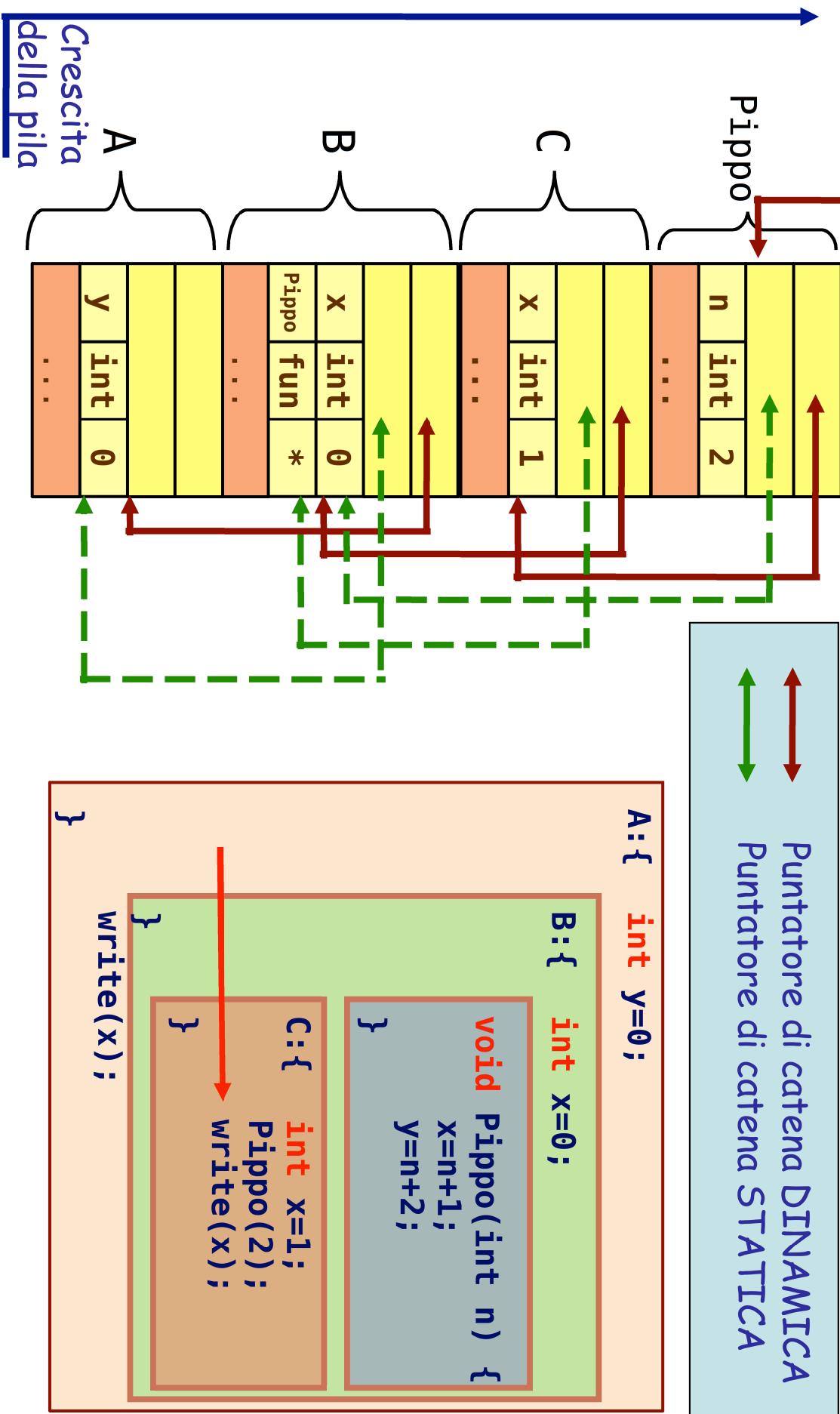
}

write(x);

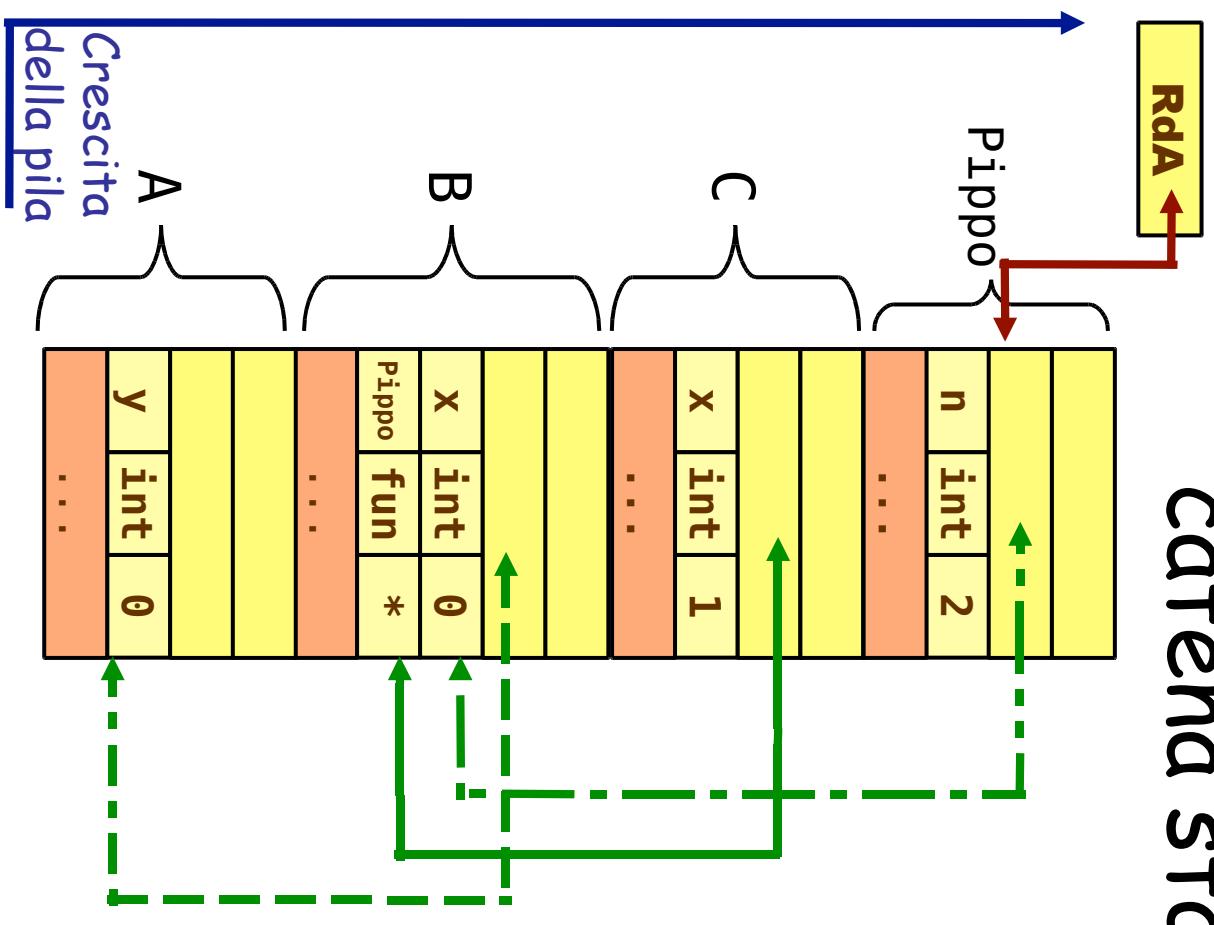
Crescita
della pila

Implementazione dello scope statico:

catena statica (cont.)



Implementazione dello scope statico: catena statica (cont.)



secondo regole di scope statico
La è usata in Pippo e quella non locale
definita in B

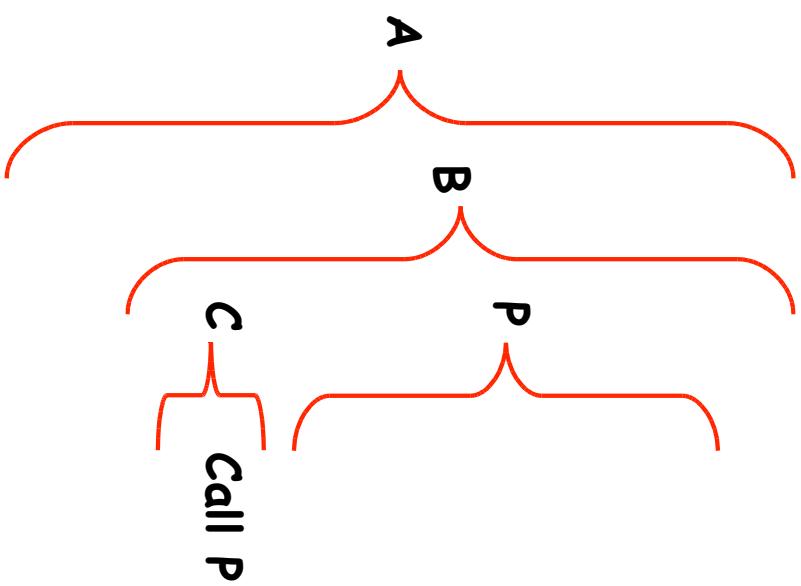
Implementazione

- RdA del blocco Pippo collegato mediante puntatore di catena statica al RdA del blocco B
- se B attivo anche i blocchi esterni che lo contengono devono essere attivi (RdA sulla pila)
- RdA del blocco B collegato mediante puntatore di catena statica al RdA del blocco immediatamente esterno ad esso (A - RdA)

Implementazione dello scope statico: catena statica

GENERALIZZANDO

- ordine di esplorazione dei RdA non è quello dei blocchi nella pila
- RdA del blocco B collegato mediante **puntatore di catena statica** al blocco "immediatamente esterno" ad esso (A)
 - se B attivo anche i blocchi esterni che lo contengono devono essere attivi (ovvero RdA presente sulla pila)
- Se C è il blocco che contiene una chiamata procedura, il blocco "immediatamente esterno" è quello in cui la procedura è definita

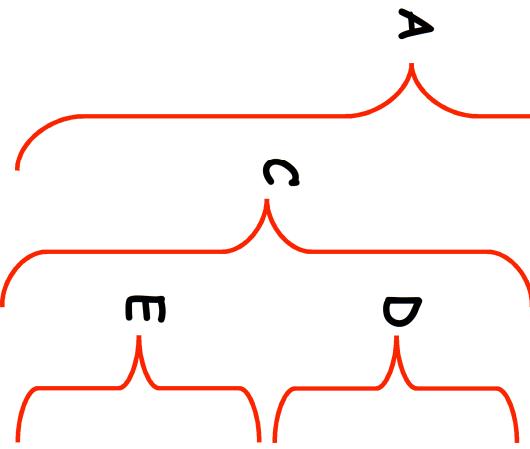
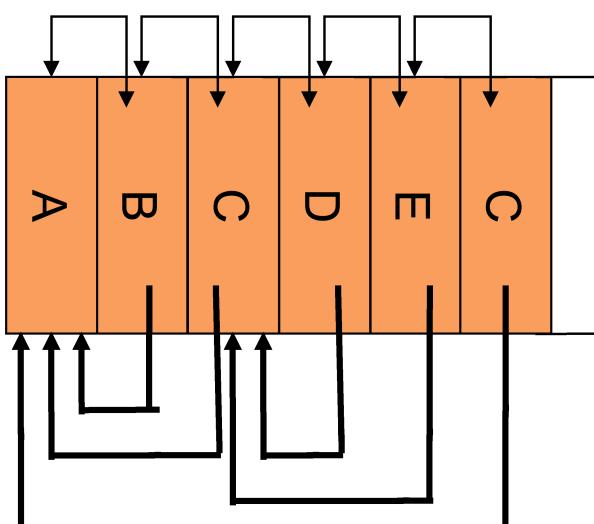


Implementazione dello scope statico: catena statica

Esempio: ABCD ed E sono blocchi annidati

Struttura statica di annidamento dei blocchi di un programma

catena statica per la sequenza di chiamate ABCDEC



I puntatori di catena statica sono determinabili a tempo di compilazione

Gestione della pila a run-time

Include la gestione della catena statica

All'entrata di un blocco: il blocco chiamante calcola il puntatore di catena statica del blocco chiamato, e glielo passa

```
A:{  
    int y=0;  
}
```

```
B:{  
    int x=0;  
}
```

```
void Pippo(int n) {  
    x=n+1;  
    y=n+2;  
}
```

```
C:{  
    int x=1;  
    Pippo(2);  
    write(x);  
}
```

```
}  
write(x);  
}
```

La sequenza di chiamata per B
(frammenti di codice inseriti dal compilatore in A)

calcola il puntatore di catena statica
per B e glielo passa (a B)

Alla chiamata di Pippo(2):

La sequenza di chiamata per Pippo(2)
(frammenti di codice inseriti dal compilatore in C)
calcola il puntatore di catena statica
per Pippo(2) e glielo passa

Implementazione dello scope statico:

calcolo puntatore di catena statica

Il blocco chiamato è all'esterno del chiamante

perché il chiamato sia visibile

il blocco chiamato deve trovarsi in un blocco esterno al chiamante che includa anche il bocco del chiamante (il RDA del chiamato è sulla pila)

A:{

int y=0;

B:{ **int x=0;**

void Pippo(int n) {

x=n+1;

y=n+2;

}

Pippo è visibile in C perchè

C e la definizione di Pippo sono entrambi interni al blocco B

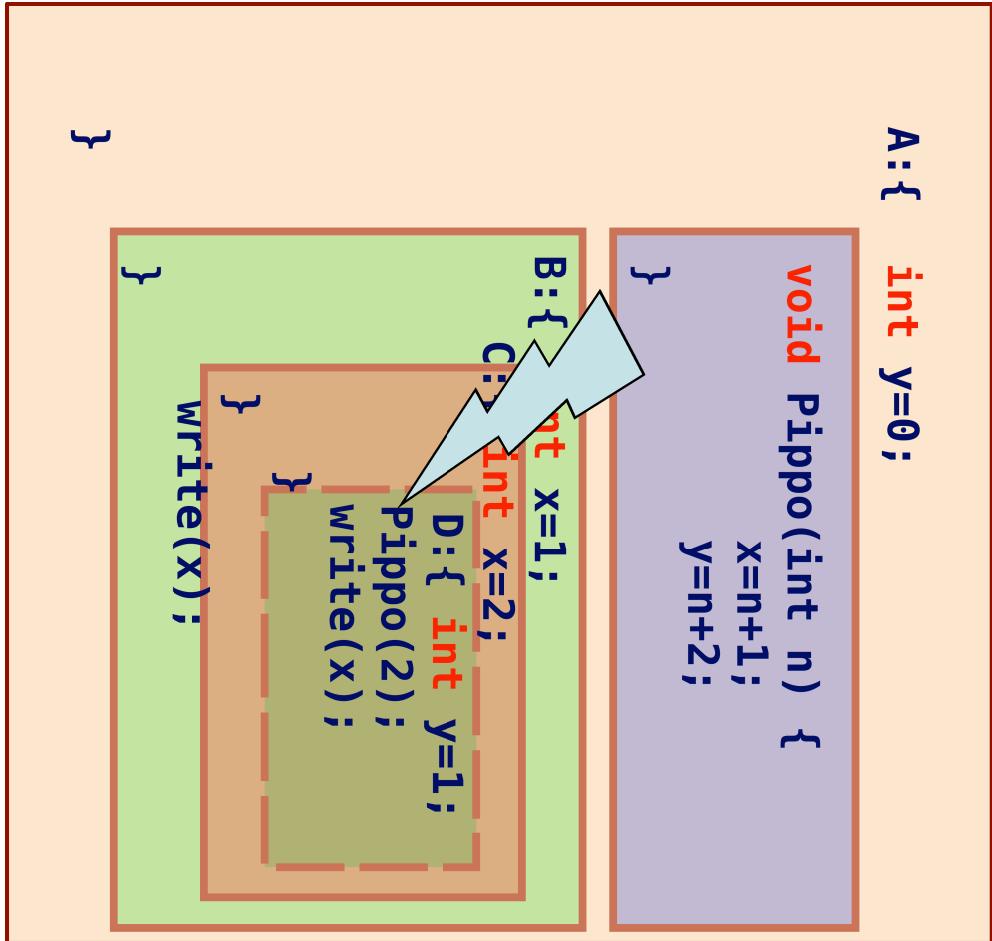


}

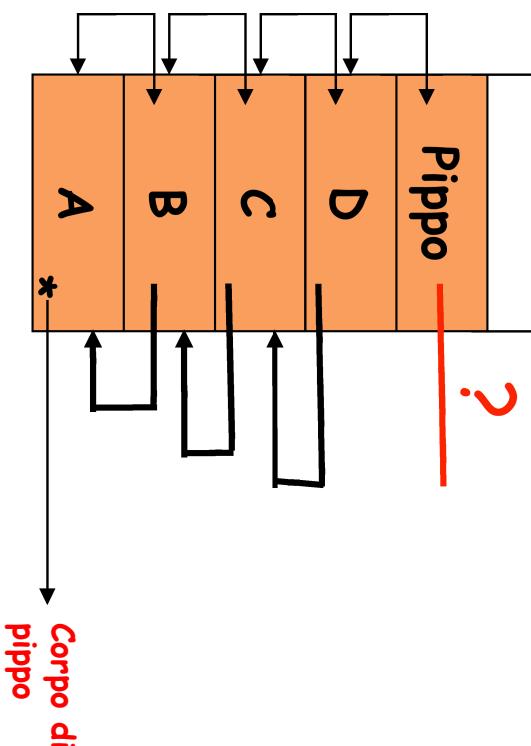
write(x);

RdA di Pippo() è già sulla pila

Implementazione dello scope statico: calcolo puntatore di catena statica **Il blocco chiamato è all'esterno del chiamante**



Pila di sistema alla chiamata
Pippo(2)



Implementazione dello scope statico: calcolo puntatore di catena statica

Il blocco chiamato è all'esterno del chiamante

```
A:{ int y=0;
```

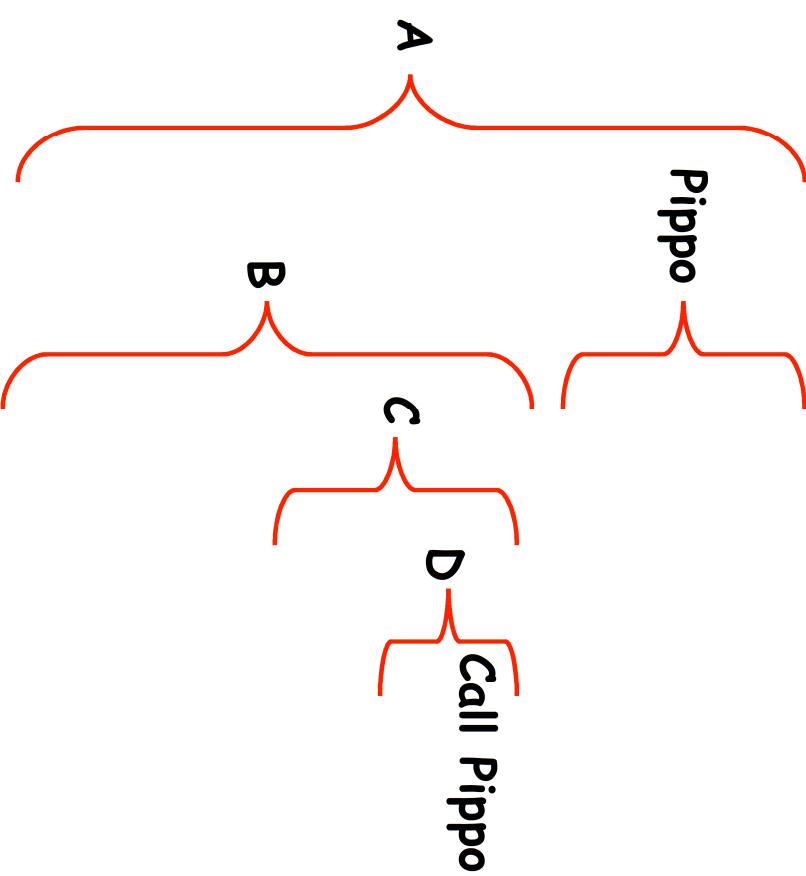
```
void Pippo(int n) {  
    x=n+1;  
    y=n+2;  
}
```

```
B:{ int x=1;  
C:{ int x=2;  
D:{ int y=1;  
Pippo(2);  
write(x);  
}  
}
```

Liv. 1

2
3
4

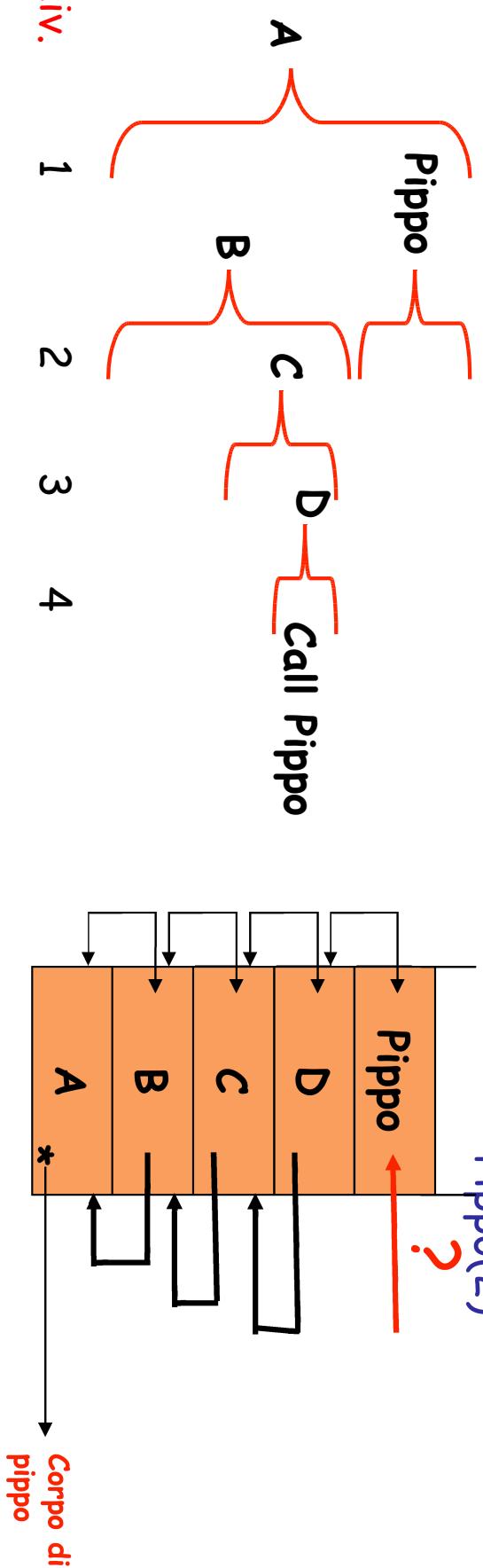
Struttura statica del programma



Implementazione dello scope statico: calcolo puntatore di catena statica

Il blocco chiamato è all'esterno del chiamante

Struttura statica del programma



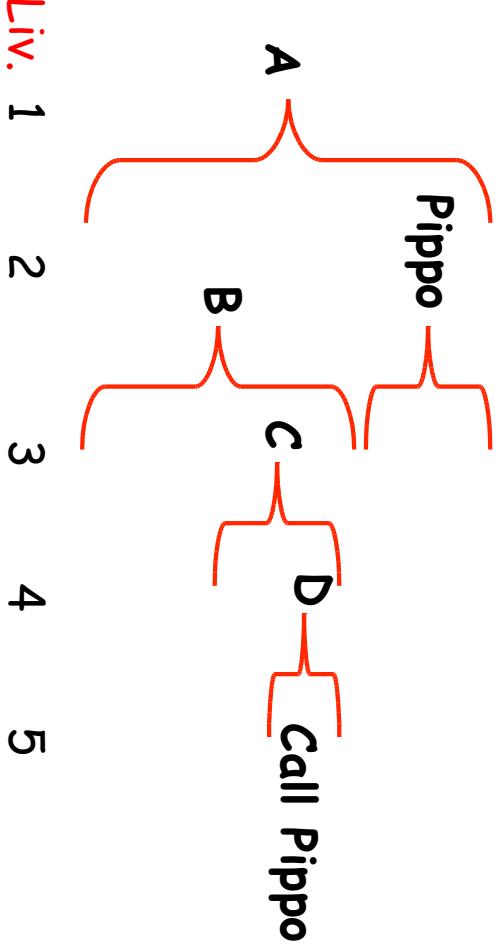
Liv. Annid chiamante D = $n=3$ Liv. Annid. Chiamato Pippo = $m=1$

$$K = 3 - 1 = 2$$

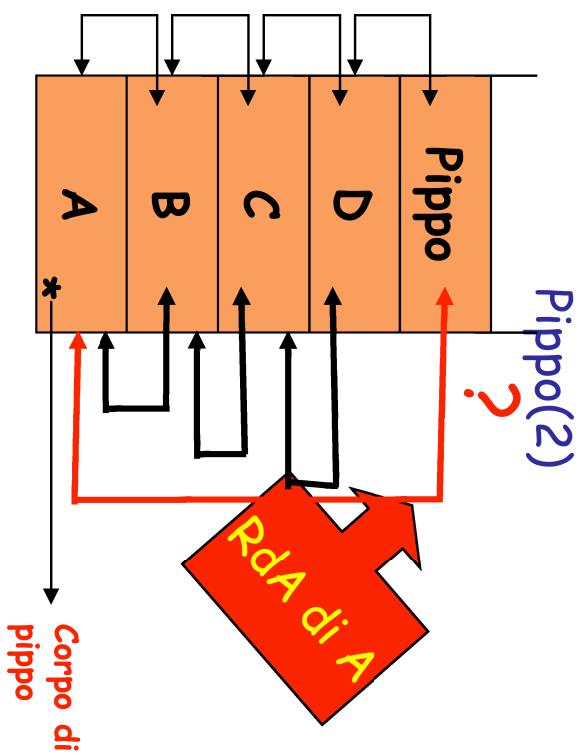
D risale la sua catena statica di ($K =$) 2 livelli e

Implementazione dello scope statico: calcolo puntatore di catena statica **Il blocco chiamato è all'esterno del chiamante**

Struttura statica del programma



Pila di sistema alla chiamata



...passa l'inf. Al chiamato (pippo)

Implementazione dello scope statico:

calcolo puntatore di catena statica

Il blocco chiamato è all'esterno del chiamante

CALCOLO sulla struttura del programma:
GENERALIZZAZIONE

Livello di annidamento del chiamante: n

Livello di annidamento del chiamato: m

$k = n - m$ sono i livelli di annidamento

- tra chiamante e chiamato
- (calcolato dal chiamante)

Per risalire all'ambiente non locale corretto del chiamato
(ovvero per calcolare il puntatore di catena statica del chiamato),
il chiamante segue k livelli della sua catena statica

Implementazione dello scope statico:

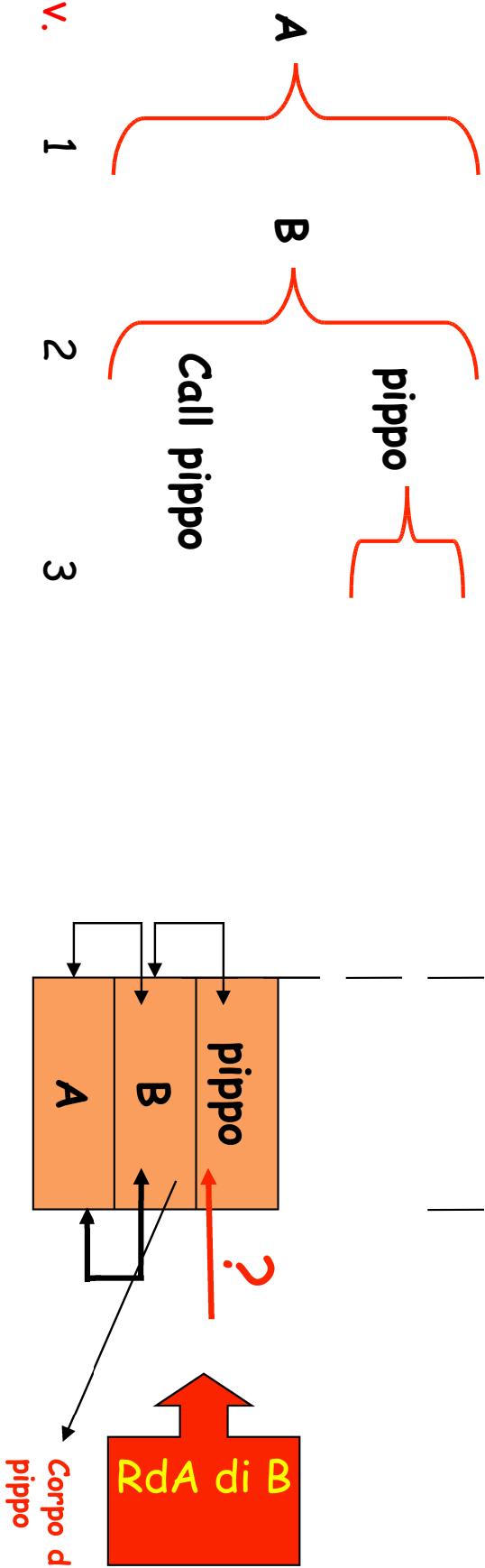
calcolo puntatore di catena statica

Il blocco chiamato è all'interno del chiamante

(es. pippo dichiarato e chiamato internamente a B)

- per la visibilità: il chiamato è dichiarato nel blocco della chiamata

• Il chiamante calcola il link di catena statica risalendo al suo blocco esterno (e passa l'inf. Al RdA del chiamato)



Implementazione dello scope statico

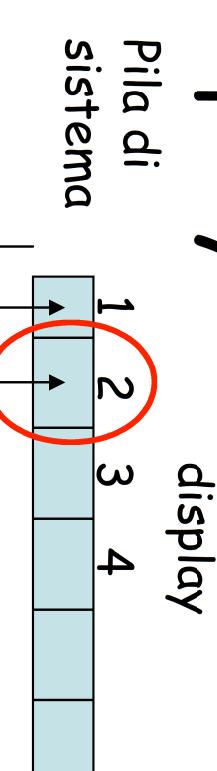
il compilatore (tramite il prologo) memorizza nel suo Rda
il puntatore di catena statica ricevuto

- il compilatore tiene traccia dei livelli di annidamento dei blocchi usando la **tabella dei simboli** (nomi e inf. per la gestione degli oggetti denotati dai nomi)
 - Identificati e numerati gli scope in base al livello di annidamento
 - Ad ogni riferimento di nome è associato anche un numero che indica lo scope (porzione di codice) che contiene la dichiarazione di tale nome
 - tuttavia non può risolvere tutti i calcoli perché alcuni sono legati ai Rda presenti sulla pila a run-time
- **svantaggio:**
per k livelli di distanza
si devono scorrere k livelli di catena statica (k accessi a memoria)

Implementazione dello scope:

tecnica del display

consente di ridurre ad un numero costante (2) gli accessi



- **display**: vettore di dim. pari a numero di livelli di annidamento

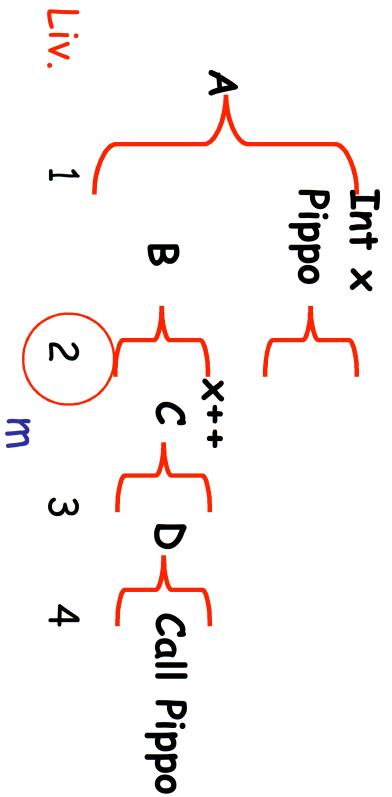
- **display[k]**: puntatore al RDA di livello k attivo (presente sulla pila)

- **Finchè il chiamato non termina la propria esecuzione il display attivo sono i primi m elementi**

- per ogni riferimento ad un nome non locale dichiarato in un blocco esterno di livello k:
 - L'oggetto dichiarato è reperibile accedendo alla posizione k del display e seguendo il puntatore.

Struttura statica del programma

Dichiarazione di x



Implementazione dello scope:

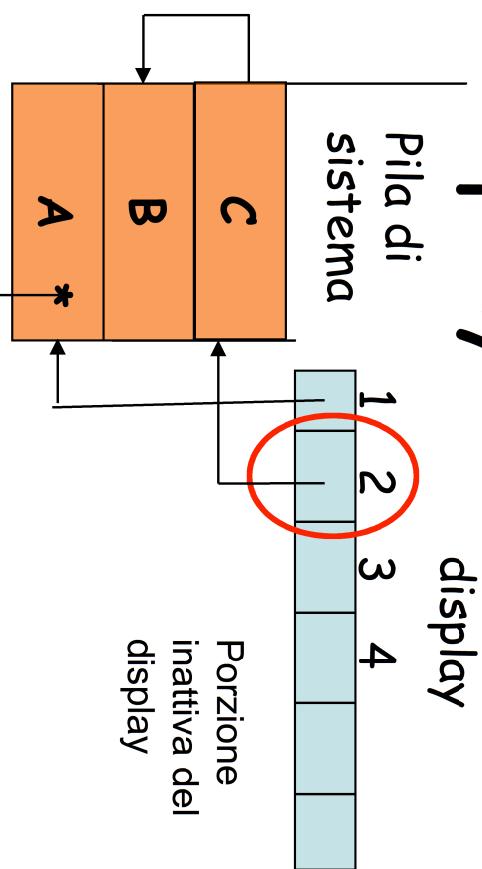
tecnica del display

All'entrata o all'uscita di un ambiente di livello k

(blocco in-line o chiamata a procedura)

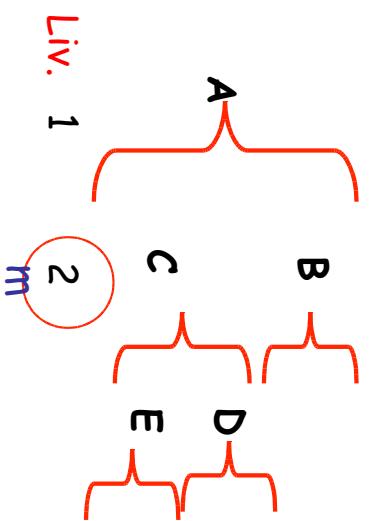
- La posizione k del display deve essere aggiornata:

- display[k] precedente viene salvato nel Rda del chiamato
- display[k] attuale contiene il valore del puntatore al Rda del chiamato (del nuovo blocco attivo del livello k)



Struttura statica del programma

Dichiarazione di x

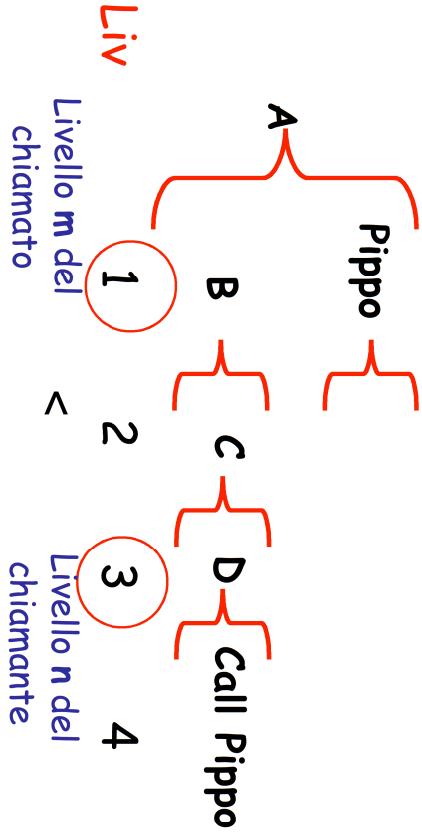


Implementazione dello scope:

tecnica del display

I chiamato si trova all'esterno del chiamante

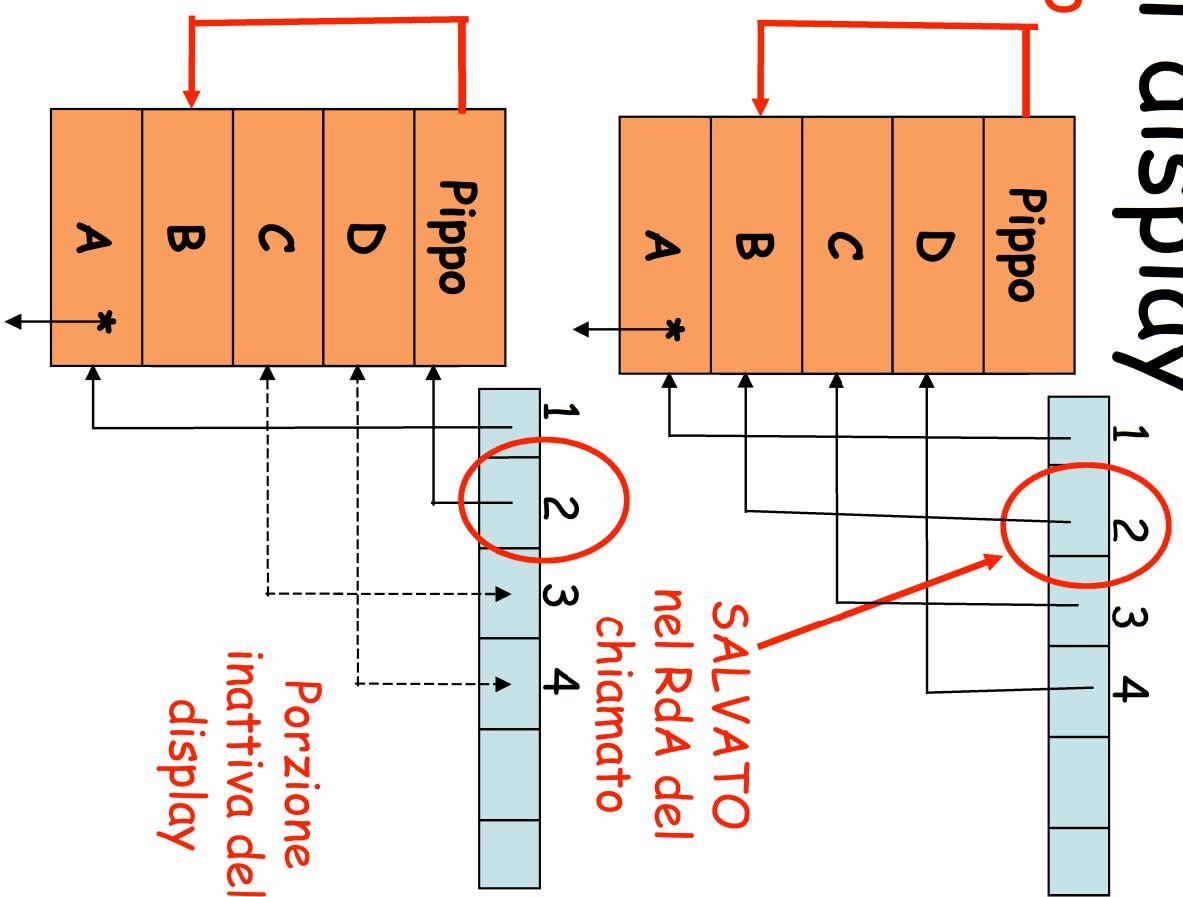
Struttura statica del programma



La posizione m del display deve essere aggiornata:

display[2] precedente viene salvato nel RdA di Pippo (il chiamato)

display[2] attuale contiene il valore del puntatore al RdA di pippo che è il nuovo blocco attivo del livello 2)



Implementazione dello scope:

tecnica del display

I chiamato si trova all'esterno del chiamante

Struttura statica del programma

Elemento m (= 2)
del display conterrà
il puntatore al RdA del chiamato (Pippo)

Finchè il chiamato (Pippo)

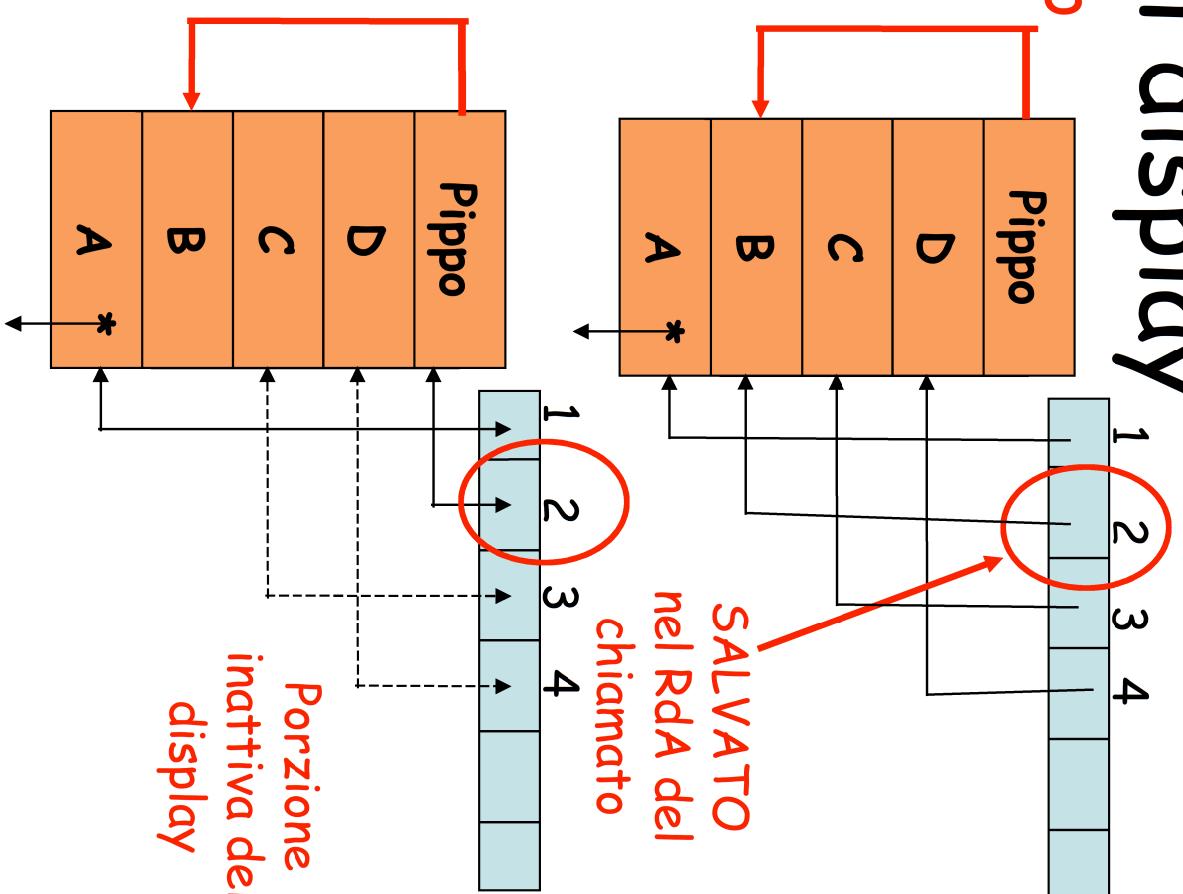
non termina la propria esecuzione,

il display attivo sono i primi m elementi

aggiornata:

display[1] precedente viene salvato nel RdA di
Pippo (il chiamato)

display[1] attuale contiene il valore del
puntatore al RdA di pippo che è il nuovo
blocco attivo del livello m)



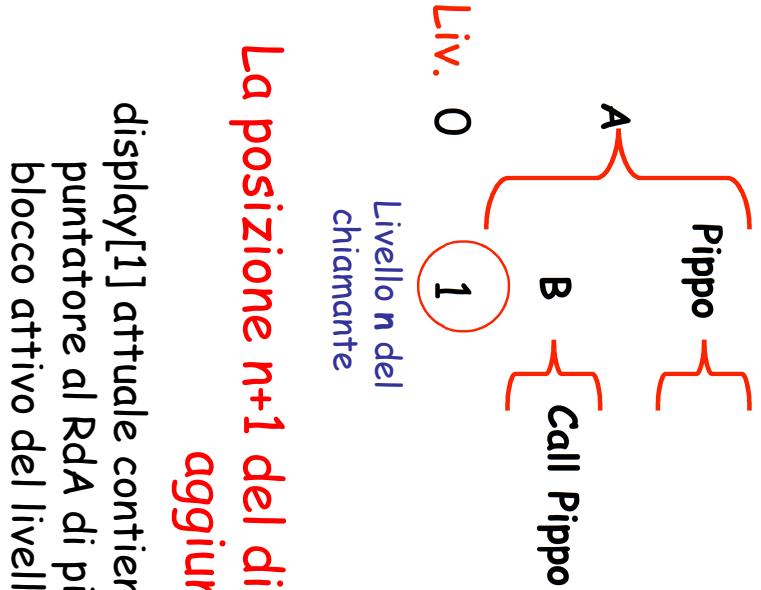
Implementazione dello scope:

tecnica del display

Il chiamato si trova all'interno

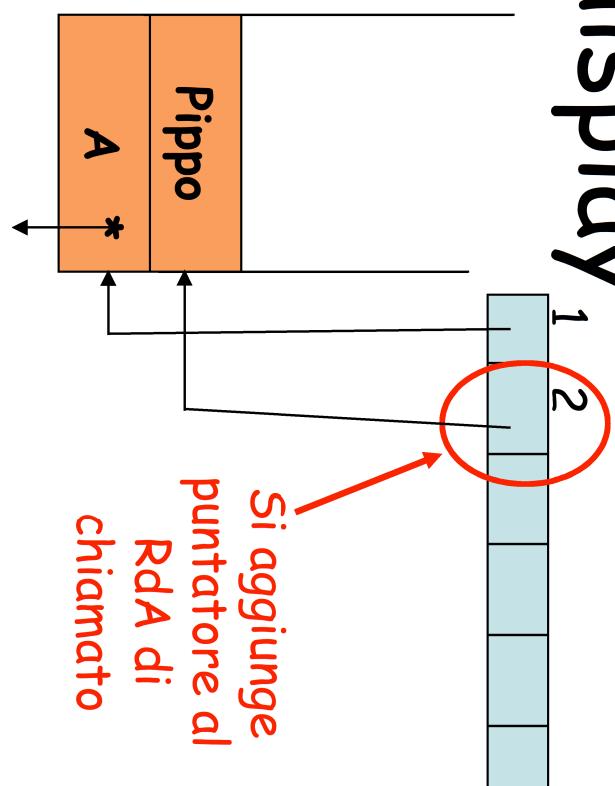
del chiamante

Struttura statica del programma



La posizione $n+1$ del display deve essere aggiunta:

display[1] attuale contiene il valore del puntatore al RdA di pippo che è il nuovo blocco attivo del livello m)



Anche in questo caso il valore precedente deve essere memorizzato nel RdA del chiamato (chiamate precedenti che usano $n+1$)

Implementazione dello scope statico: tecnica del display

consente di ridurre ad un numero costante (2) gli accessi

- gestione semplice ma più costosa (rispetto alla catena statica):
 - chiamata/entrata sottoprogramma/blocco di livello k:
 - il chiamato
- salva il valore di `display[k]`, se presente (op. aggiuntiva)
 - condivisione del `display` in tutto o in parte col chiamante
- aggiorna `display[k]` con il puntatore del Rda

Implementazione dello Scope dinamico: possibilità

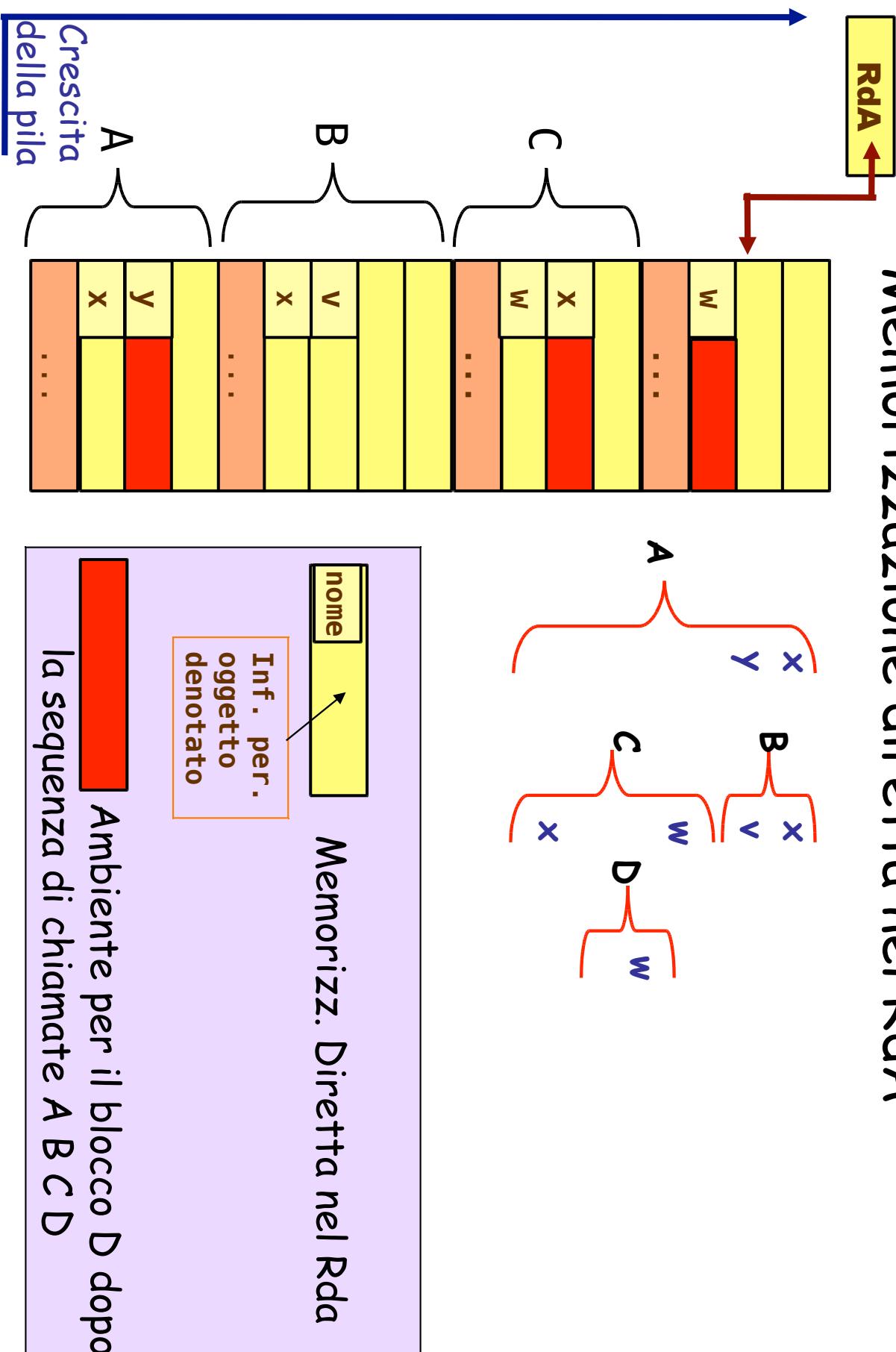
ordine di esplorazione dei Rda è quello dei blocchi nella pila
(vedi puntatori di catena dinamica)

Associazioni nome-oggetto denotato
(ovvero gli ambienti locali):

- 1) Memorizzate direttamente nel Rda
- 2) Memorizzate separatamente in una lista di
associazioni (A-list)
- 3) Memorizzate in un'unica tabella centrale (Tabella
Centrale dell'ambiente)

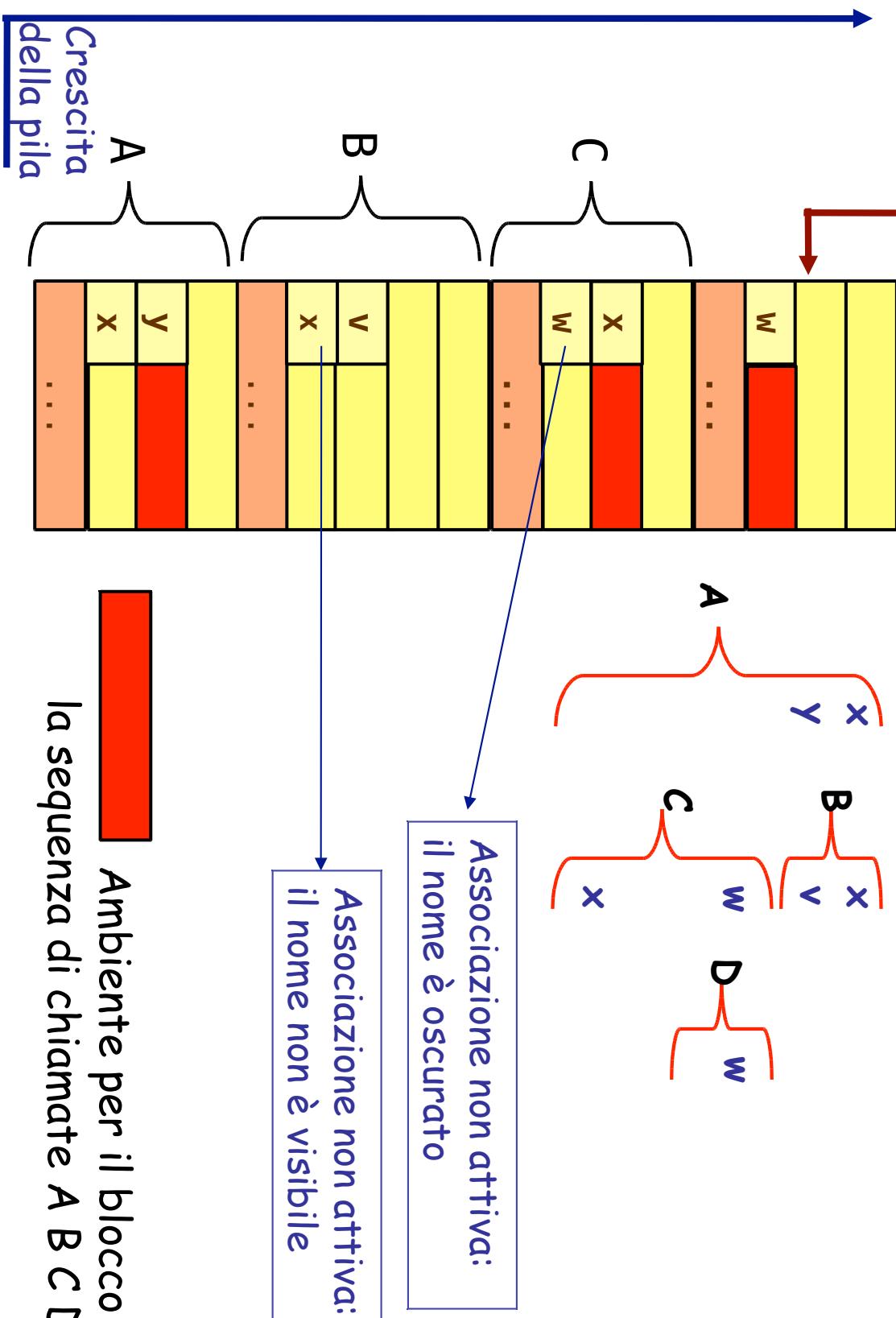
Implementazione dello Scope dinamico:

Memorizzazione diretta nel Rda



Implementazione dello Scope dinamico:

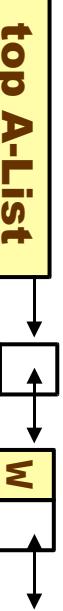
Memorizzazione diretta nel RdA



Crescita della pila

Ambiente per il blocco D dopo la sequenza di chiamate A B C D

Implementazione dello Scope dinamico:

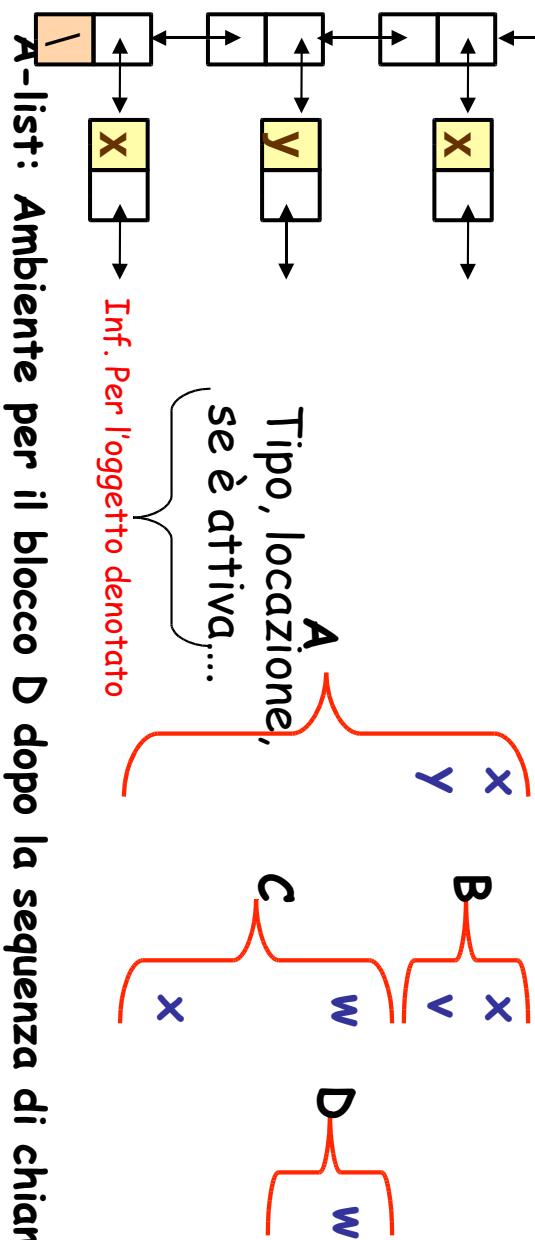


Memorizzazione nella A-list

Quando l'esecuzione di un programma

1) entra in un nuovo blocco:
le associazioni locali vengono inserite
nella A-list (gestita LIFO)

2) Esce da un blocco: le associazioni locali
vengono rimosse dalla A-list



Crescita della pila
A-list: Ambiente per il blocco D dopo la sequenza di chiamate A B C D

Implementazione dello Scope dinamico: Memorizz. In RdA / A-list

ordine di esplorazione dei RdA è quello dei blocchi nella pila

(vedi puntatori di catena dinamica)

- vantaggio: più semplice:
 - ambienti non locali considerati secondo l'ordine di chiamata
- svantaggi:
 - necessità di memorizzare esplicitamente i nomi in strutture presenti a run-time (no nello Scope Statico)
 - A-list: evidente perché necessario
 - Mem. In RdA: scope dinamico → determinazione dei RdA da usarsi per i riferimenti non locali non determinabile staticamente → come si ritrovano le associazioni per quel nome? → ricerca a tempi di esecuzione
 - può essere inefficiente la ricerca a run-time (es. Variabili globali → esplorazione di tutti RdA/lista)

Implementazione dello Scope dinamico: Tabella centrale dell'ambiente

Per ovviare agli svantaggi di mem.in Rda/A-list

- tutti i blocchi in un'unica tabella
(CRT: Central Referencing environment Table)
 - tutti i nomi
 - per ogni nome:
 - flag (associazione attiva sì/no)
 - puntatore alle informazioni associate (locazione, tipo, ...)
 - se tutti i nomi sono noti a compile-time
 - ricerca in tempo costante (ind. tabella+offset)
 - altrimenti: ricerca hash

Implementazione dello Scope dinamico: Tabella centrale dell'ambiente

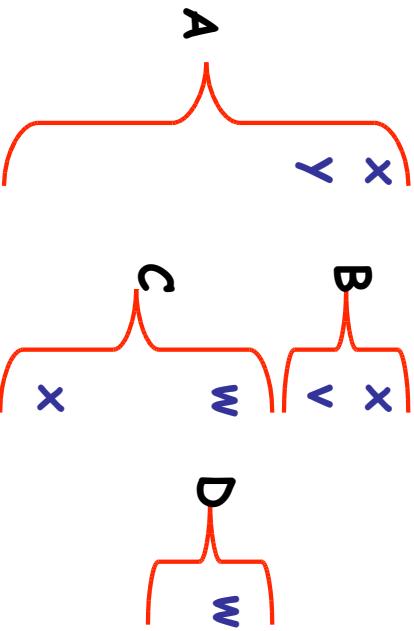
- Ingresso da un blocco A in un blocco B più costose:
 - modifica della CRT per un nuovo ambiente locale di A
 - non necessariamente in entry contigue nella tabella
 - salvataggio associazioni deattivate (in una pila) per successivo ripristino all'uscita da B
- Struttura usata per la gestione: una pila per ogni nome
 - esplicita
 - top: associazione attiva per quel nome
 - altre: associazioni deattivate per quel nome
 - nascosta (separata dalla CRT)
 - (nome, flag, rif. ad oggetto)
- non è richiesta ricerca
 - accesso (diretto o hash) alla tabella, che contiene l'info

Implementazione dello Scope dinamico:

Tabella centrale con pile esplicite

Quando l'esecuzione di un programma

- 1) entra in un nuovo ambiente:
 - modifica della CRT per un nuovo ambiente locale
 - salvataggio associazioni deattivate (in una pila) per successivo ripristino all'uscita da B



Crescita della pila

Dichiar. in A

X	
---	--

X	
Y	

Dichiar. in A

Tipo, loc.	
/	

X	
Y	
V	/
W	/

CRT: Ambiente per il blocco
D dopo la sequenza di chiamata A

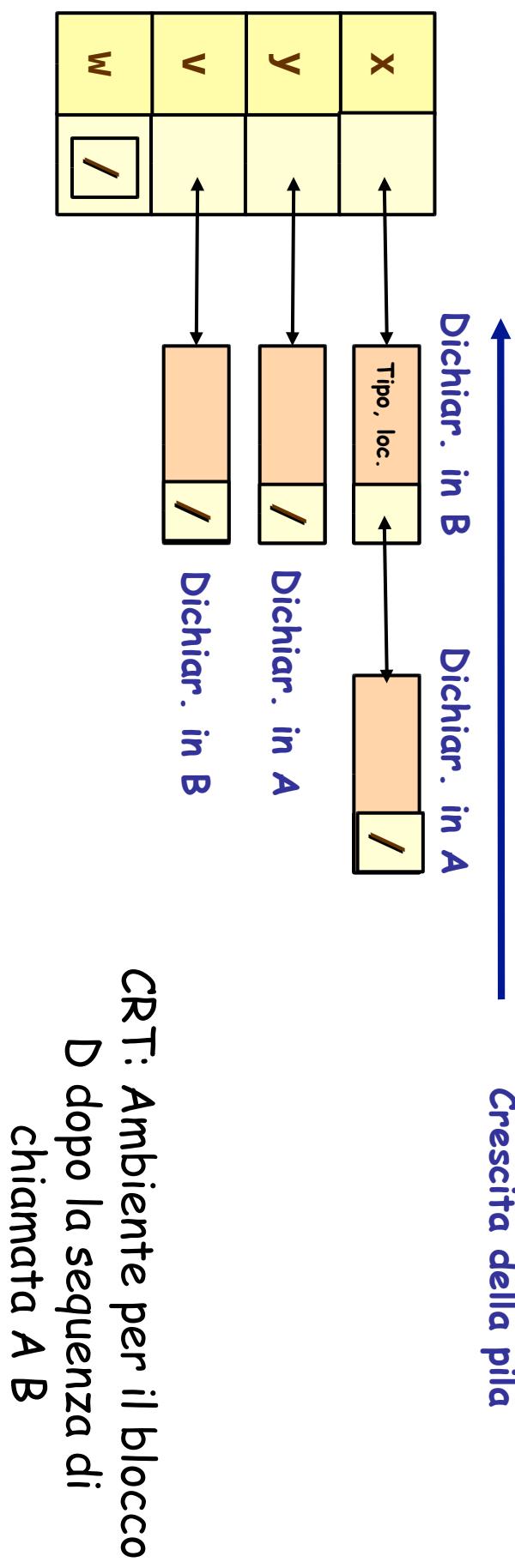
Implementazione dello Scope dinamico:

Tabella centrale con pile esplicite

Quando l'esecuzione di un programma

- 1) entra in un nuovo ambiente:
 - modifica della CRT per un nuovo ambiente locale
 - salvataggio associazioni deattivate (in una pila) per successivo ripristino all'uscita da B

Crescita della pila



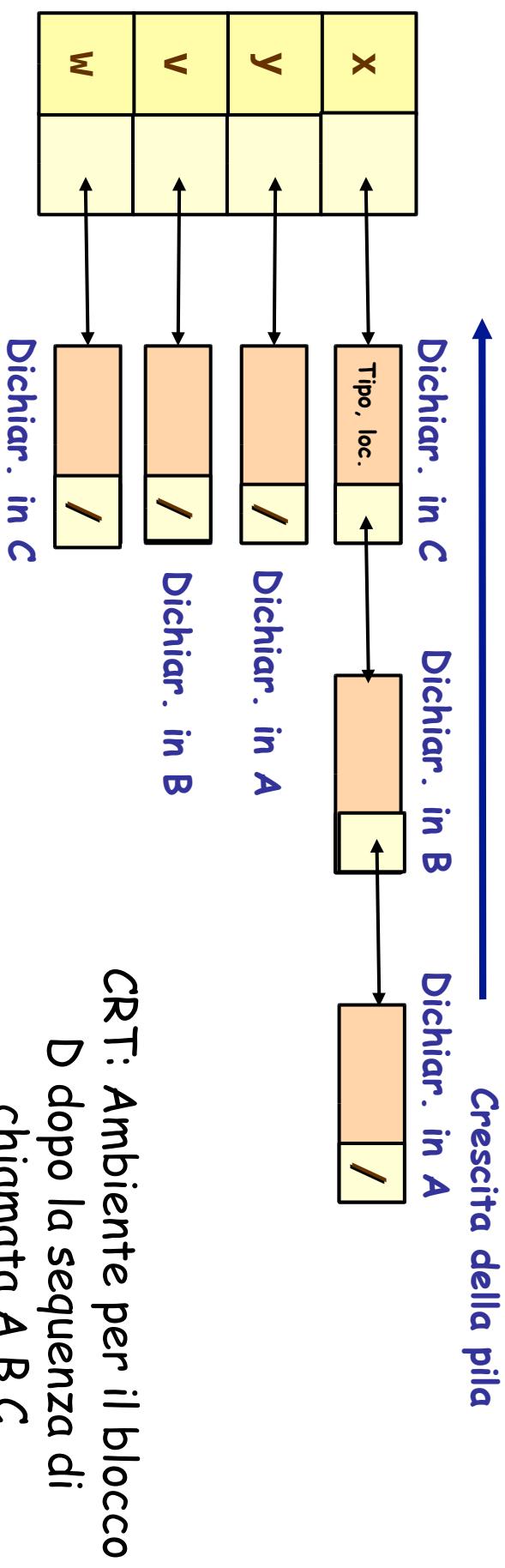
CRT: Ambiente per il blocco
D dopo la sequenza di chiamata A B

Implementazione dello Scope dinamico:

Tabella centrale con pile esplicite

Quando l'esecuzione di un programma

- 1) entra in un nuovo ambiente:
 - modifica della CRT per un nuovo ambiente locale
 - salvataggio associazioni deattivate (in una pila) per successivo ripristino all'uscita da B

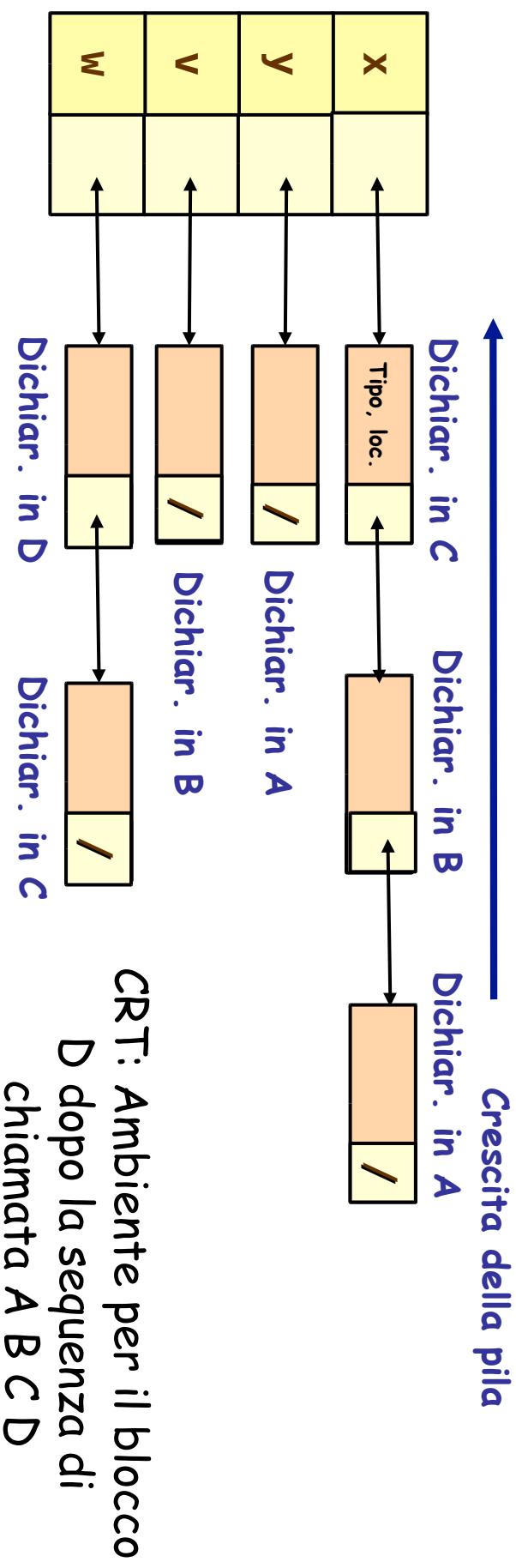


Implementazione dello Scope dinamico:

Tabella centrale con pile esplicite

Quando l'esecuzione di un programma

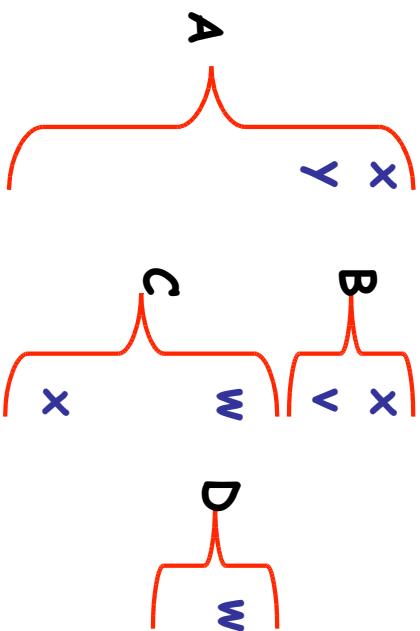
- 1) entra in un nuovo ambiente:
 - modifica della CRT per un nuovo ambiente locale
 - salvataggio associazioni deattivate (in una pila) per successivo ripristino all'uscita da B



Implementazione dello Scope dinamico:

Tabella centrale con pila nascosta

Quando l'esecuzione di un programma
1) esce da un blocco le associazioni
deattivate vengono memorizzate nella
pila nascosta



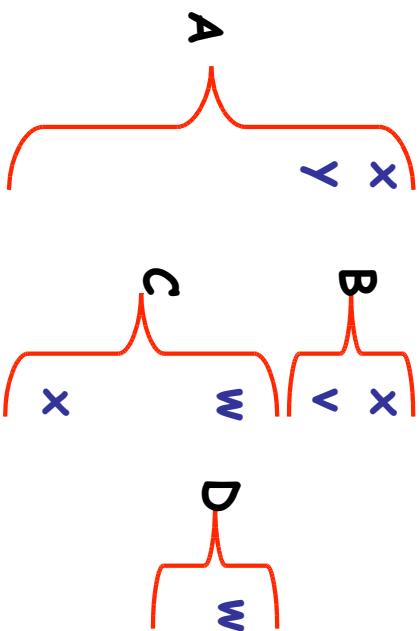
CRT:	A	B
x	1	a1
y	1	a2
v	0	-
w	0	-

nome
↓ Puntatore all'oggetto denotato
↓ Flag: ass.attiva/non attiva

Implementazione dello Scope dinamico:

Tabella centrale con pila nascosta

Quando l'esecuzione di un programma
1) esce da un blocco le associazioni
deattivate vengono memorizzate nella
pila nascosta



CRT:
A

X	1	a1
Y	1	a2
V	0	-
W	0	-

AB

X	1	b1
Y	1	a2
V	1	b2
W	0	-

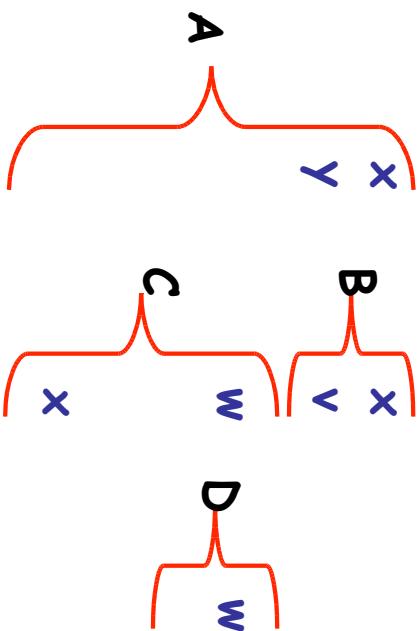
Pila nascosta:

X	a1
---	----

Implementazione dello Scope dinamico:

Tabella centrale con pila nascosta

Quando l'esecuzione di un programma
1) esce da un blocco le associazioni
deattivate vengono memorizzate nella
pila nascosta



CRT:
A

X	1	a1
Y	1	a2
V	0	-
W	0	-

AB

X	1	b1
Y	1	a2
V	1	b2
W	0	-

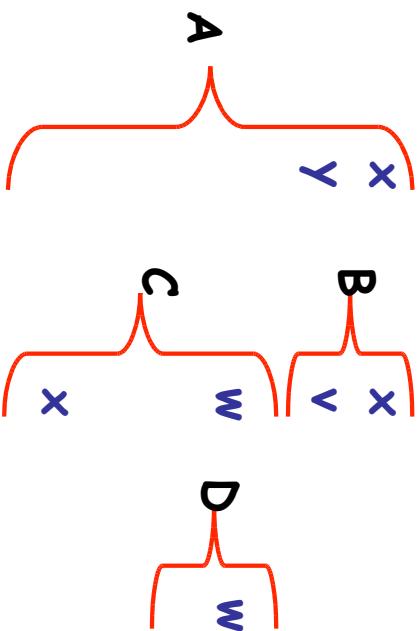
Pila nascosta:

X	a1
---	----

Implementazione dello Scope dinamico:

Tabella centrale con pila nascosta

Quando l'esecuzione di un programma
1) esce da un blocco le associazioni
deattivate vengono memorizzate nella
pila nascosta



CRT: A AB ABC

X	1	a1
y	1	a2
v	0	-
w	0	-

X	1	b1
y	1	a2
v	1	b2
w	0	-

X	1	c1
y	1	a2
v	0	b2
w	1	c2

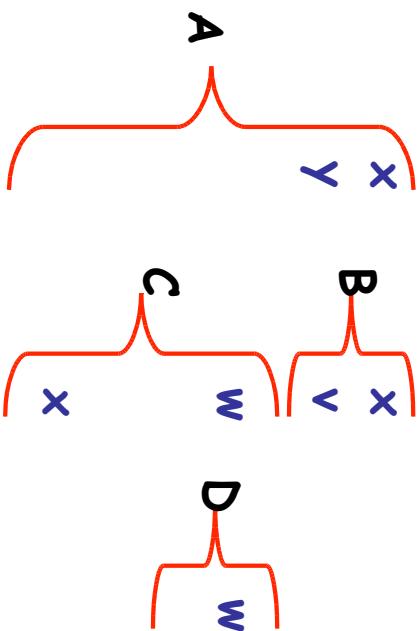
Pila nascosta:

x	a1
---	----

x	a1
x	b1

Implementazione dello Scope dinamico:

Tabella centrale con pila nascosta



Quando l'esecuzione di un programma
1) esce da un blocco le associazioni
deattivate vengono memorizzate nella
pila nascosta

CRT:

A	X	1	a1
B	y	1	a2
C	v	0	-
D	w	0	-

AB

A	X	1	b1
B	y	1	a2
C	v	1	b2
D	w	0	-

ABC

A	X	1	c1
B	y	1	a2
C	v	0	b2
D	w	1	c2

ABCD

A	X	1	c1
B	y	1	a2
C	v	0	b2
D	w	1	d1

Pila nascosta:

X	a1
---	----

X	a1
X	b1

X	a1
X	b1
W	c2