

AA 2016-2017

## 22. Funzioni e procedure

# Breve storia dei sottoprogrammi



- ✎ **Astrazione di una sequenza di istruzioni**
  - un frammento di programma (sequenza di istruzioni) risulta utile in diversi punti del programma
  - riduce il “costo della programmazione” se si può dare un nome al frammento e viene inserito automaticamente il codice del frammento ogni qualvolta nel “programma principale” c’è un’occorrenza del nome
    - ✓ **macro e macro-espansione**

# Macro in C



```
#define MULT(x, y) x * y
```

```
int z = MULT(3 + 2, 4 + 2);
```

```
int z = 3 + 2 * 4 + 2;  
// 2 * 4 valutato prima
```

Cosa viene assegnato a z?

13!!!

Code in-lining

# Breve storia dei sottoprogrammi



- ✉ **Astrazione sul controllo:** si riduce anche l'occupazione di memoria se esiste un meccanismo che permette al programma principale
  - di trasferire il controllo a una unica copia del sottoprogramma memorizzata separatamente
  - di riprendere il controllo quando l'esecuzione del frammento è terminata
  - ed è un meccanismo supportato direttamente dall'hardware (**codice rientrante**)

# Breve storia dei sottoprogrammi

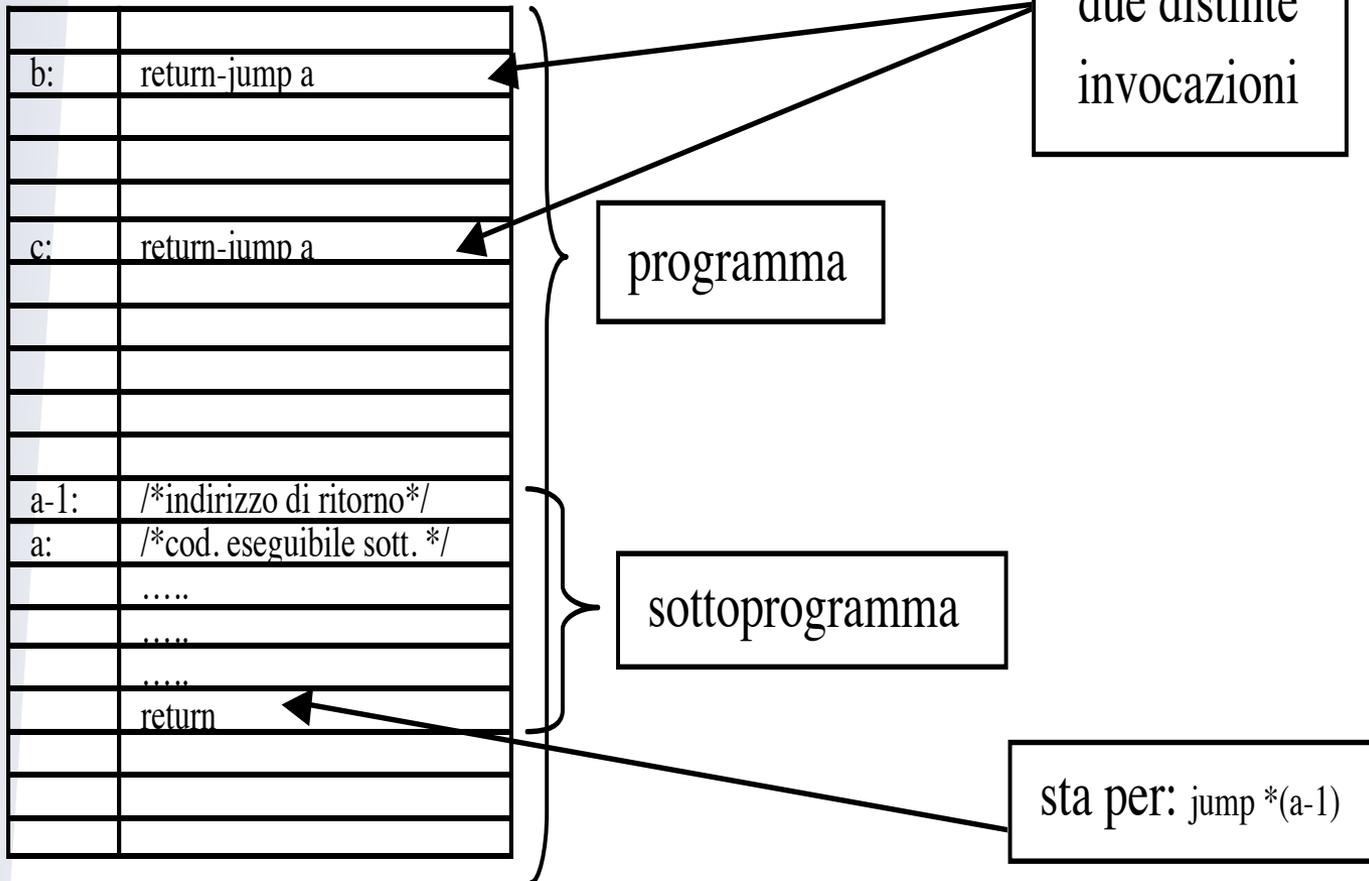


- ✎ Ancora meglio se permettiamo **astrazione via parametrizzazione**
  - astraendo dall'identità di alcuni dati
  - la cosa è possibile anche con le macro e il codice rientrante
    - ✓ macro-espansione con rimpiazzamento di entità diverse
    - ✓ associazione di informazioni variabili al codice rientrante



# Cosa fornisce l'hardware?

- ⌘ Primitiva di **return jump** con opportune strutture ausiliarie
- ⌘ Viene eseguita (nel programma chiamante) l'istruzione **return jump a** memorizzata nella cella **b**
  - il controllo viene trasferito alla cella **a** (entry point della subroutine)
  - l'indirizzo dell'istruzione successiva del chiamante (**b + 1**) viene memorizzato in qualche posto noto, per esempio nella cella (**a - 1**) (**punto di ritorno**)
- ⌘ quando nella subroutine si esegue una operazione di return
  - il controllo ritorna all'istruzione (del programma chiamante) memorizzata nel punto di ritorno



# Archeologia: FORTRAN



- ④ Una **subroutine** è un pezzo di codice compilato, al quale sono associati
  - una cella destinata a contenere (a tempo di esecuzione) i punti di ritorno relativi alle chiamate
  - alcune celle destinate a contenere i valori degli eventuali parametri
  - l'ambiente locale è statico

# Semantica della subroutine à la FORTRAN



- ✎ Si può definire facilmente attraverso la ***copy rule statica*** (“macro-espansione”)
  - ogni chiamata di sottoprogramma è *testualmente rimpiazzata* da una copia del codice
    - ✓ facendo qualcosa per i parametri
    - ✓ ricordandosi che le dichiarazioni sono eseguite una sola volta
- ✎ Il sottoprogramma non è semanticamente qualcosa di nuovo: è solo un (importante) strumento metodologico (astrazione!)

# Semantica della subroutine à la FORTRAN



- ✎ Osservazione: non è compatibile con la ricorsione
  - la macroespansione darebbe origine ad un programma infinito
  - l'implementazione à la FORTRAN (con un solo punto di ritorno) non permetterebbe di gestire più attivazioni presenti allo stesso tempo
- ✎ Il fatto che le subroutine FORTRAN siano concettualmente statiche fa sì che
  - non esista di fatto il concetto di attivazione
  - l'**ambiente locale** sia necessariamente **statico**

# Attivazione



- ✎ Se ragioniamo in termini di attivazioni, la semantica può essere ancora definita da una **copy rule**, ma **dinamica**
  - ogni chiamata di sottoprogramma è **rimpiazzata a tempo di esecuzione** da una copia del codice
- ✎ Il sottoprogramma è ora semanticamente qualcosa di nuovo
- ✎ Ragionare in termini di attivazioni
  - rende naturale la ricorsione
  - porta ad adottare la regola dell'**ambiente locale dinamico**

# Le strutture di implementazione



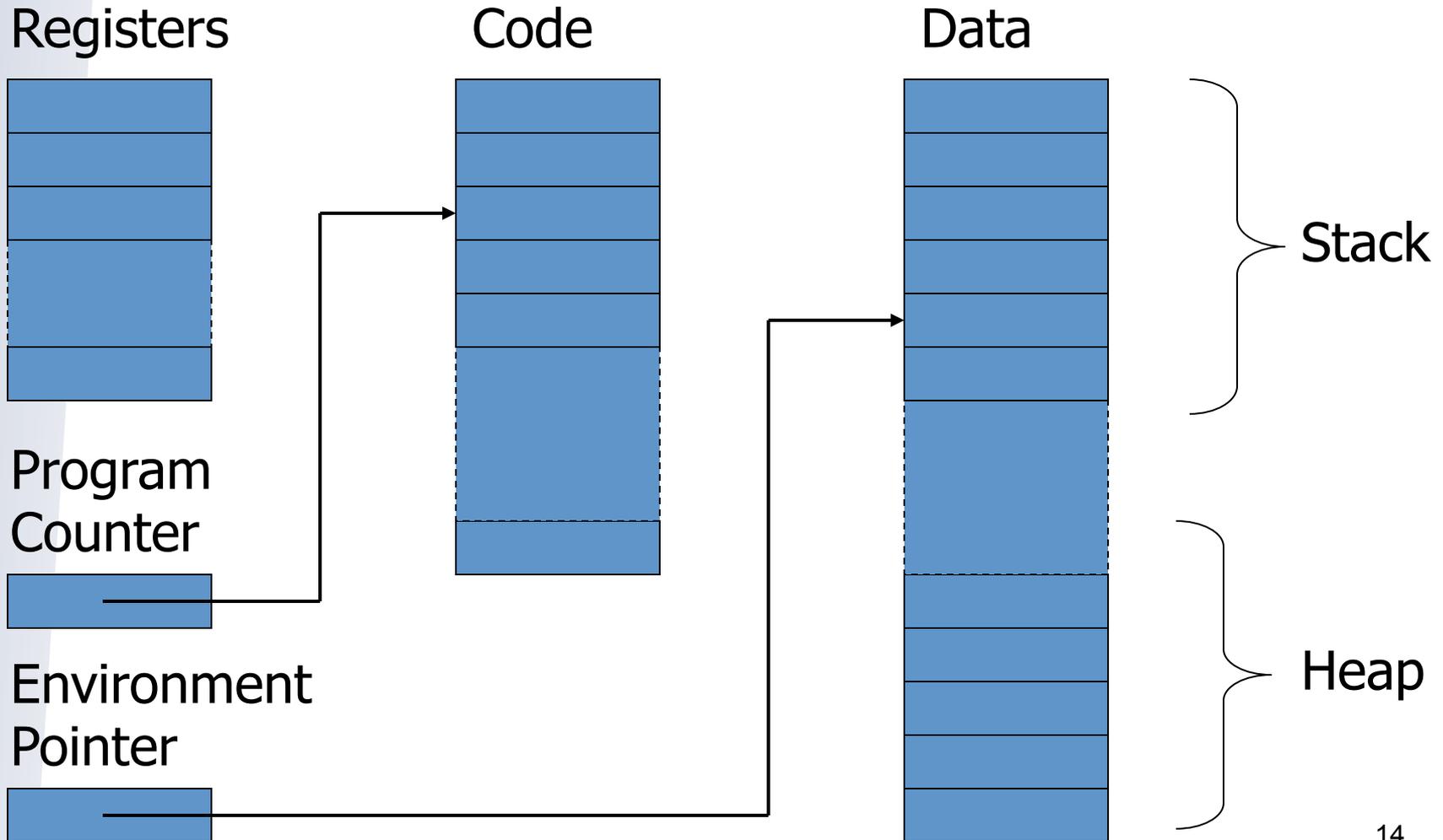
- ✎ Invece delle informazioni **staticamente** associate al codice compilato di FORTRAN
  - punto di ritorno, parametri, ambiente e memoria locale si usano i **record di attivazione**
  - contenenti le stesse informazioni ma associati dinamicamente alle varie chiamate di sottoprogrammi
- ✎ Dato che l'accesso ai sottoprogrammi segue una politica LIFO
  - l'ultima attivazione creata nel tempo è la prima che ritorna possiamo organizzare i record di attivazione in una pila

# Cosa è un vero sottoprogramma



- ✎ Astrazione procedurale (operazioni)
  - astrazione di una sequenza di istruzioni
  - astrazione via parametrizzazione
- ✎ Luogo di controllo per la gestione dell'ambiente e della memoria
  - in assoluto, l'aspetto più interessante dei linguaggi, intorno al quale ruotano tutte le decisioni semantiche importanti
  - binding: statico o dinamico

# Modello di macchina hw





# Meccanismo call/return di sottoprogramma

- ✎ Chiamante
  - crea una istanza del record di attivazione
  - salva lo stato dell'unità corrente di esecuzione
  - effettua il passaggio dei parametri
  - inserisce il punto di ritorno
  - trasferisce il controllo al chiamato
- ✎ Chiamato (prologo)
  - salva il valore corrente di Environment Pointer (EP) e lo memorizza nel link dinamico
  - definisce il nuovo valore di EP
  - alloca le variabili locali

# Meccanismo call/return di sottoprogramma



## 🕒 Chiamato (epilogo)

- eventuale passaggio di valori (dipende dalla modalità di passaggio dei parametri - lo vedremo dopo)
- il valore calcolato dalla funzione viene trasferito al chiamante
- ripristina le informazioni di controllo (il vecchio valore di EP salvato come link dinamico)
- ripristina lo stato di esecuzione del chiamante
- trasferisce il controllo al chiamante

# Come si realizza?



- ✉ Partiamo dalla cosa più semplice: i blocchi
  - sostanzialmente delle procedure senza nome e senza parametri

# In-line block

- Record di attivazione -- Activation record
  - tipo di dati di sistema memorizzato nello stack
  - gestisce l'ambiente locale

## Esempio

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

Push AR con spazio per x, y  
Assegna i valori a x, y

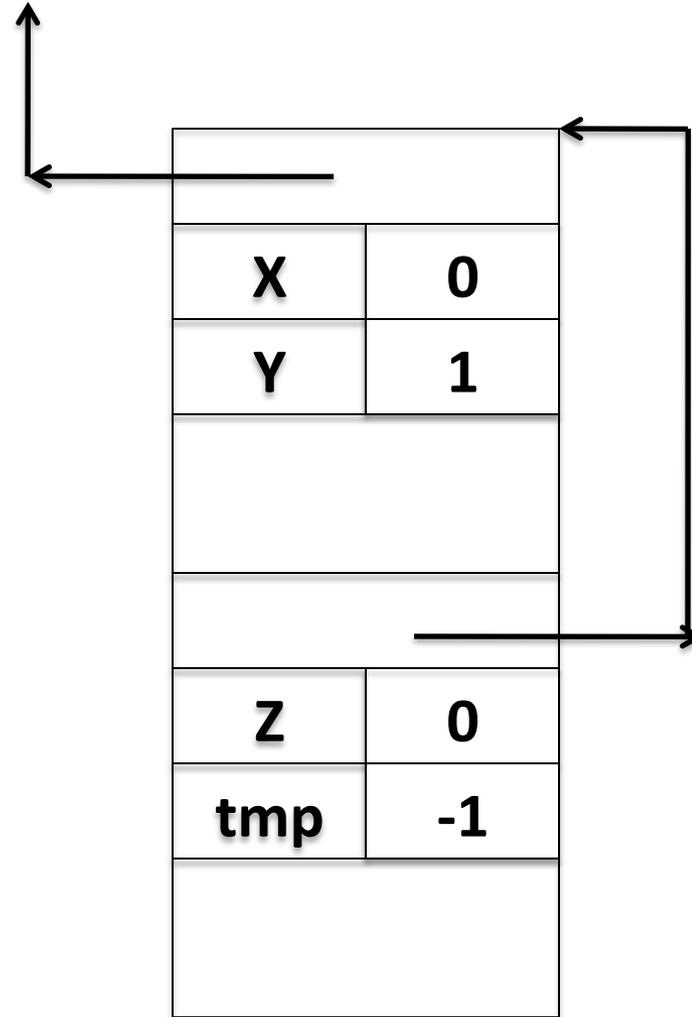
Push AR per blocco interno  
Assegna valore a z

Pop AR per blocco interno

Pop AR per blocco esterno

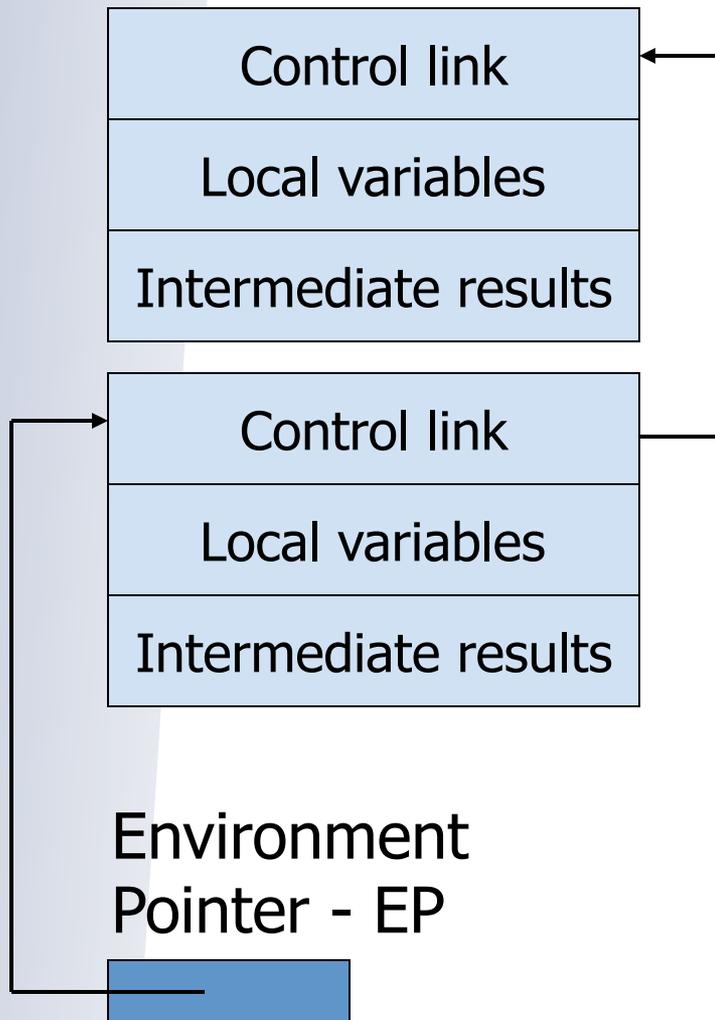
Occorre prevedere spazio per memorizzare i risultati intermedi

# STACK



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

# Record di attivazione per in-line block

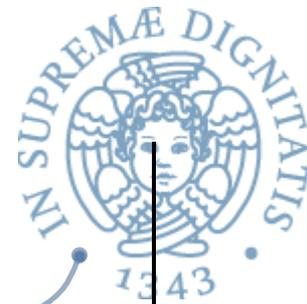


- 🦋 Control link
  - puntatore a AR precedente nello stack
- 🦋 Push AR
  - il valore di EP diviene il valore del control link del nuovo AR
  - modifica EP a puntare al nuovo AR
- 🦋 Pop record off stack
  - il valore del nuovo EP viene ottenuto seguendo il control link

# Record di attivazione

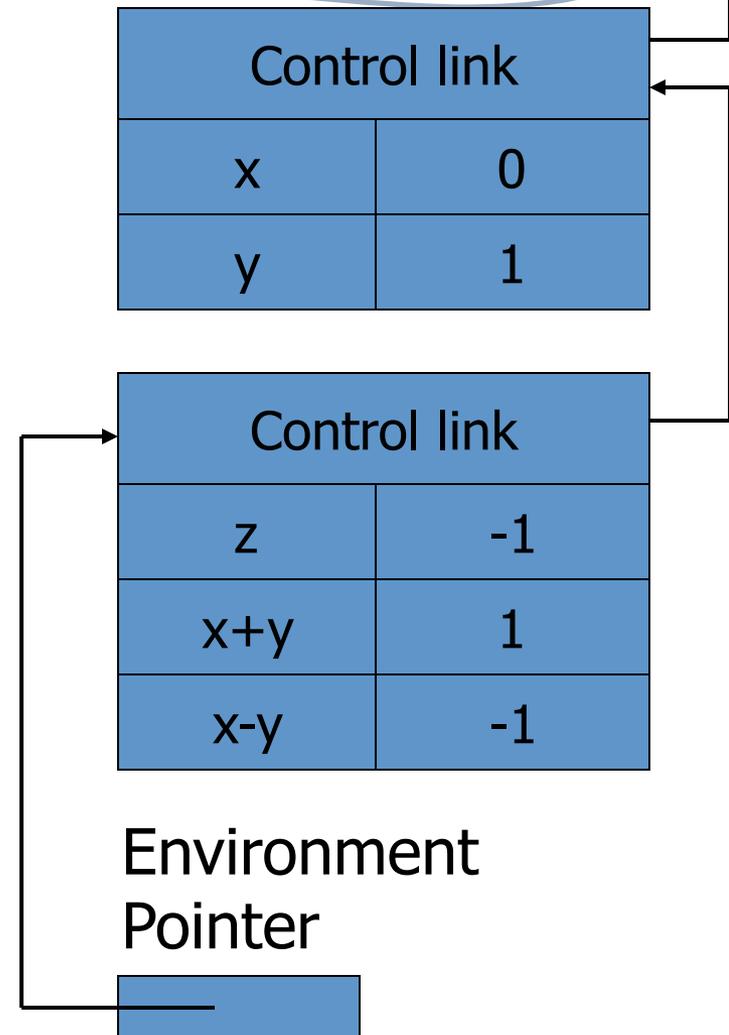


<b>Control link</b> <b>Puntatore di catena dinamica</b>
<b>Variabili locali</b>
<b>Risultati intermedi</b>



# Esempio completo

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





# E le regole di scope?

## 🦋 Variabili e ambiente

- x, y locali al blocco esterno
- z locale al blocco interno
- x, y non locali per il blocco interno

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

## 🦋 Static scope

- riferimenti non locali si risolvono nel più vicino blocco esterno

## 🦋 Dynamic scope

- riferimenti non locali si risolvono nell'AR precedente sullo stack

**Nel caso di in-line block le due nozioni coincidono**

# Analisi



- ✎ Il meccanismo dello stack dei record di attivazione è un meccanismo efficiente
- ✎ Per risolvere un riferimento locale basta accedere al record di attivazione in testa allo stack (tramite EP) e poi cercare il nome nell'ambiente locale memorizzato nel record di attivazione
- ✎ Maggiore efficienza se potessimo eliminare i nomi dal codice in esecuzione (dettagli in seguito)

# Funzioni e procedure

## Procedure (Algol)

```
procedure P (<pars>)  
begin  
    <local vars>  
    <proc body>  
end;
```

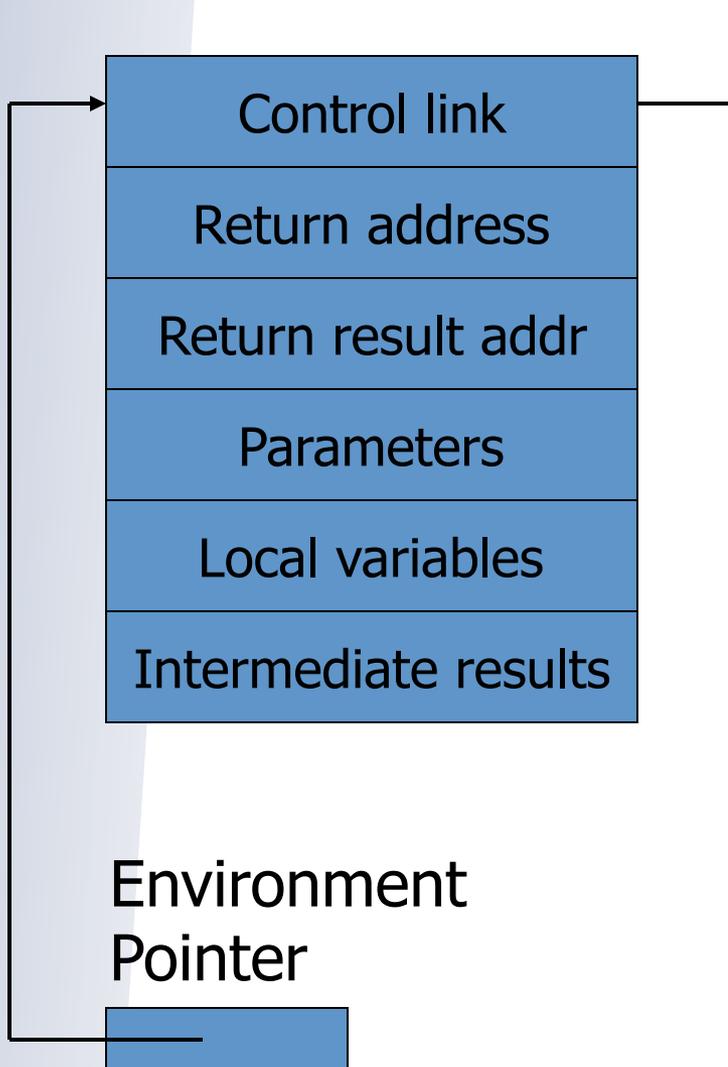
## Funzioni (C)

```
<type> function f(<pars>)  
{  
    <local vars>  
    <function body>  
}
```

🐼 Cosa ci deve stare nel record di attivazione?

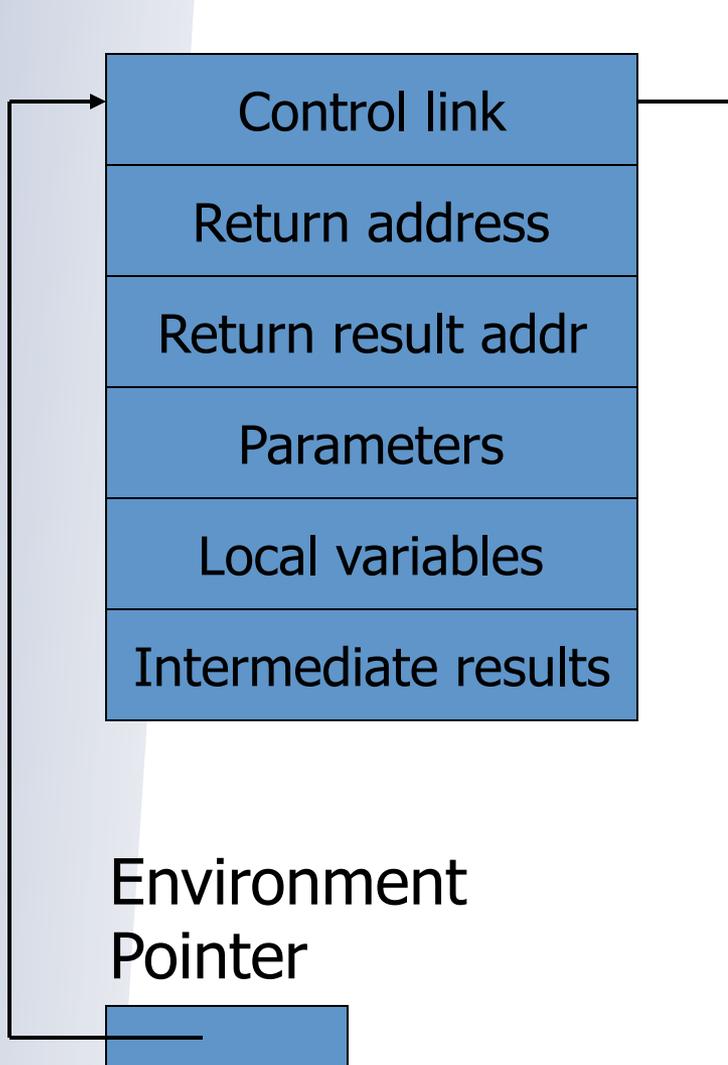
- parametri
- indirizzo di ritorno
- variabili locali, risultati intermedi
- valore restituito (caso part. di risultato intermedio)
- spazio per il valore restituito al momento del ritorno

# Funzioni: struttura AR



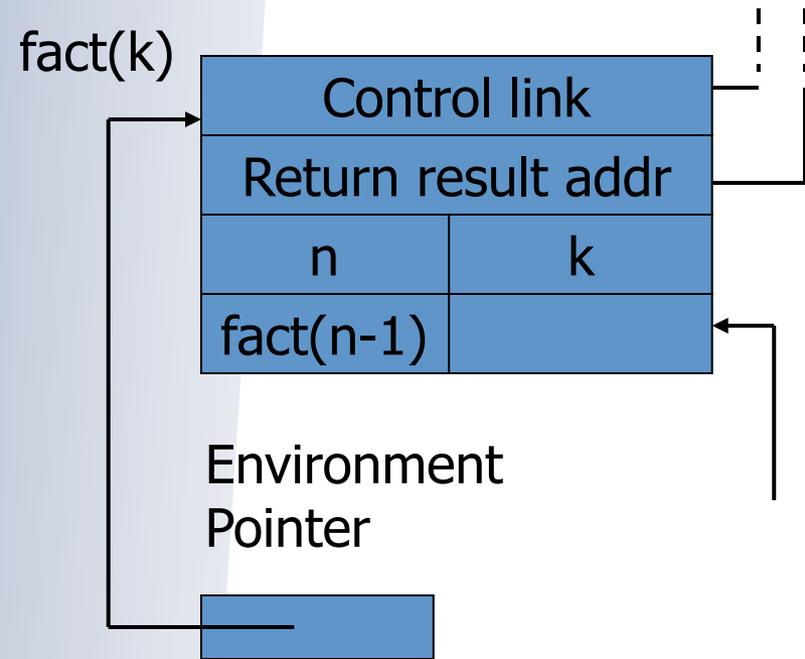
- ✎ Return address
  - indirizzo della istruzione da eseguire quando viene restituito il controllo al chiamante
- ✎ Return result address
  - indirizzo nell'AR del chiamante dove memorizzare il risultato
- ✎ Parameters
  - parametri della funzione

# Esempio



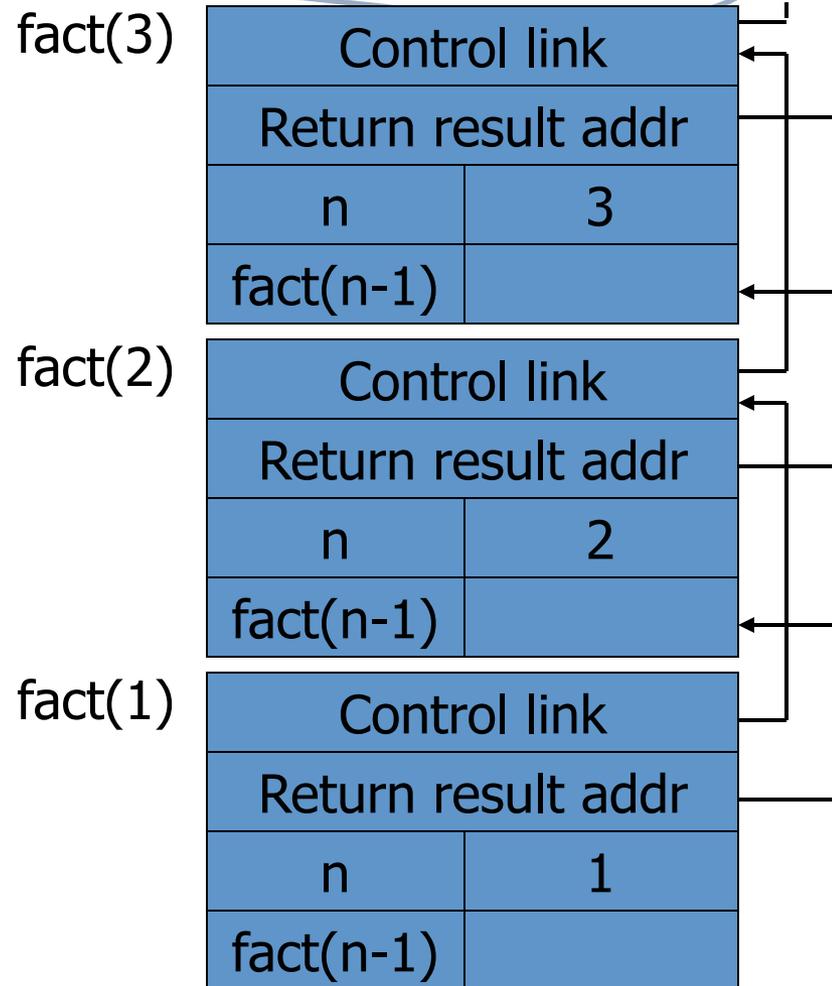
- 🦋 Il solito fattoriale  
$$\text{fact}(n) = \text{if } n \leq 1 \text{ then } 1$$
$$\text{else } n * \text{fact}(n-1)$$
- 🦋 Return result address
  - indirizzo dove memorizzare  $\text{fact}(n)$
- 🦋 Parameters
  - associazione tra  $n$  e il valore del parametro attuale
- 🦋 Intermediate results
  - spazio per memorizzare il valore di  $\text{fact}(n-1)$

# Call & ...



fact(n) = if  $n \leq 1$  then 1  
 else  $n * \text{fact}(n-1)$

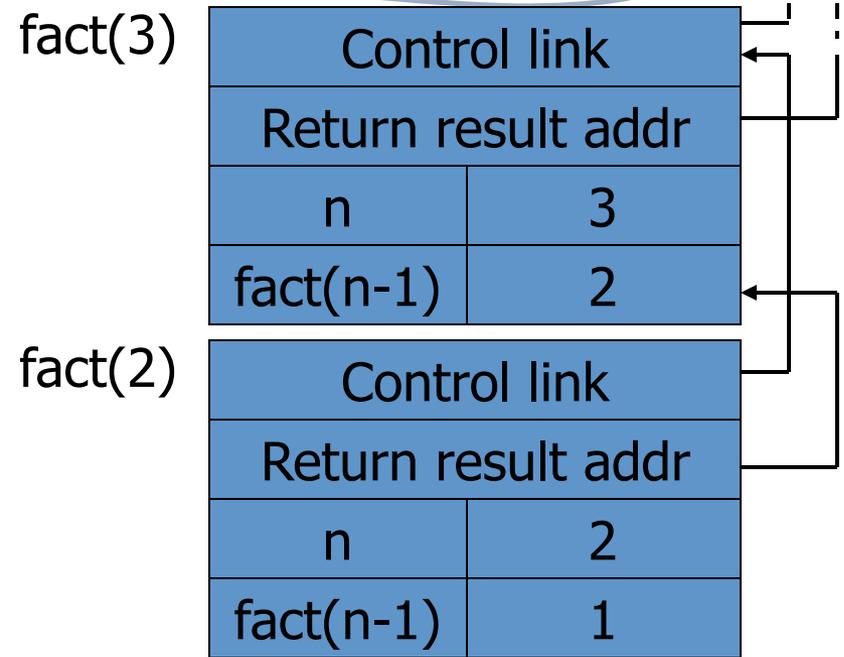
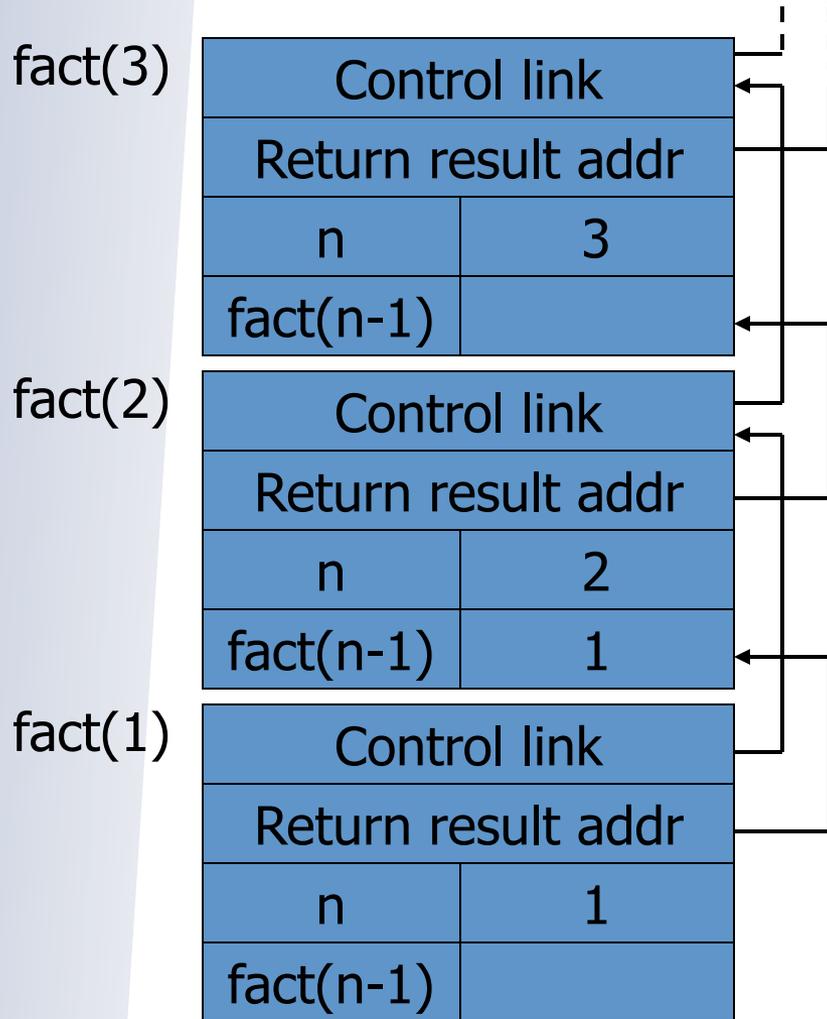
Per semplicità non inseriamo  
 il valore del return address



Continua →



# ..... & return



fact(n) = if  $n \leq 1$  then 1  
else  $n * \text{fact}(n-1)$

# Altri aspetti



- ✎ Passaggio dei parametri
  - per valore: copiare il valore del parametro attuale nello spazio previsto nel record di attivazione
  - per riferimento: copiare il valore del puntatore nel record di attivazione
- ✎ Variabili globali
  - le variabili globali sono memorizzate nel record di attivazione che sta in fondo allo stack (il primo a essere creato)
- ✎ Esamine questi aspetti con un semplice debugger!!

# Passaggio dei parametri



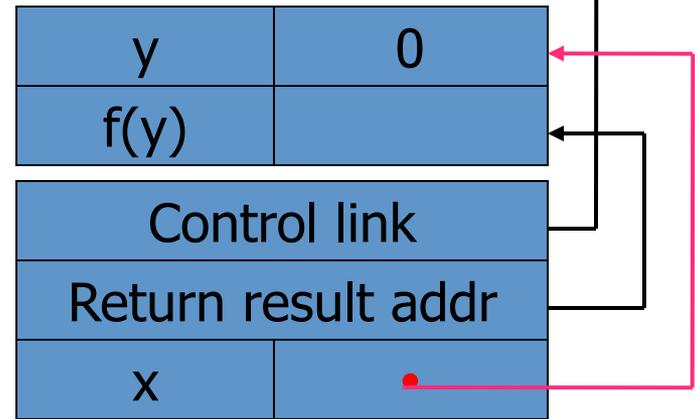
- ✎ L-value & R-value: Assegnamento  $y := x$ 
  - identificatore sulla sinistra dell'assegnamento denota la locazione e viene solitamente chiamato L-value
  - identificatore sulla destra fa riferimento al contenuto della locazione e viene chiamato R-value
- ✎ Per riferimento
  - memorizzare L-value (indirizzo di  $x$ ) nel record di attivazione
  - il corpo della funzione può modificare il parametro attuale
  - aliasing: parametro formale e parametro attuale
- ✎ Per valore
  - memorizzare R-value (contenuto di  $x$ ) nel record di attivazione
  - il corpo della funzione non può modificare il valore del parametro attuale
  - non abbiamo aliasing

# Esempio

```
function f (x) =
  { x = x+1; return x; }
var y = 0;
println (f(y)+y);
```

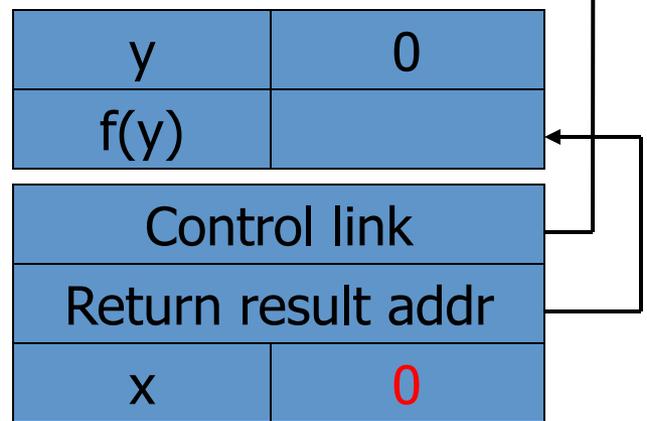
*pass-per-ref*

f(y)



*pass-per-val*

f(y)



Cosa stampa nei due casi?

# Variabili non locali

## Due alternative

- static scope (scoping statico)
- dynamic scope (scoping dinamico)

## Esempio

```
var x = 1;
function g(z) {
    return x+z; }
function f(y) {
    var x = y+1;
    return g(y*x); }
f(3);
```

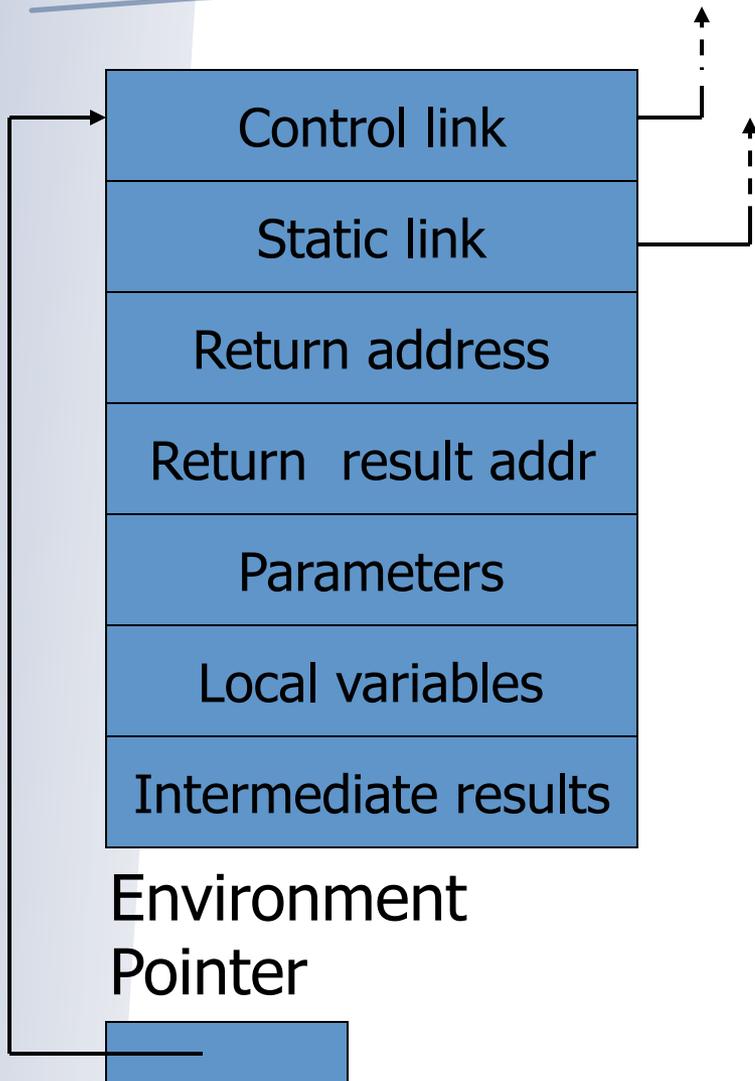
x	1
---	---

f(3)	y	3
	x	4

g(12)	z	12
-------	---	----

Quale è il riferimento corretto di x nel valutare x+z ?

# Scoping statico



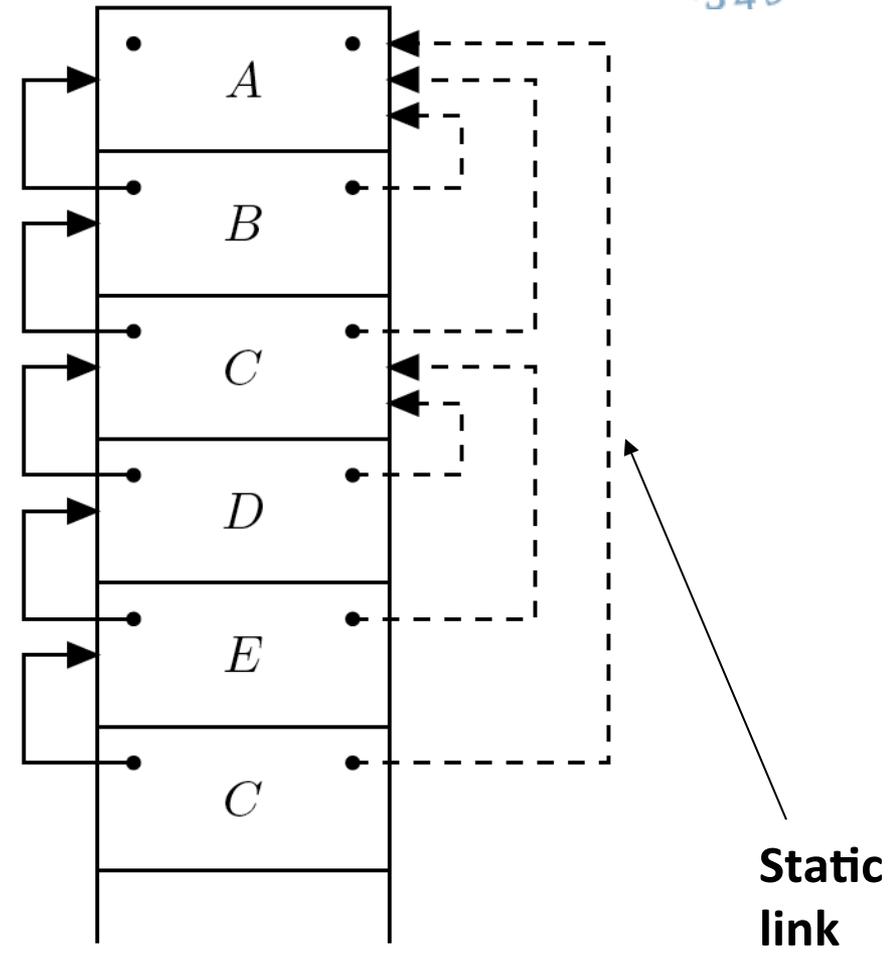
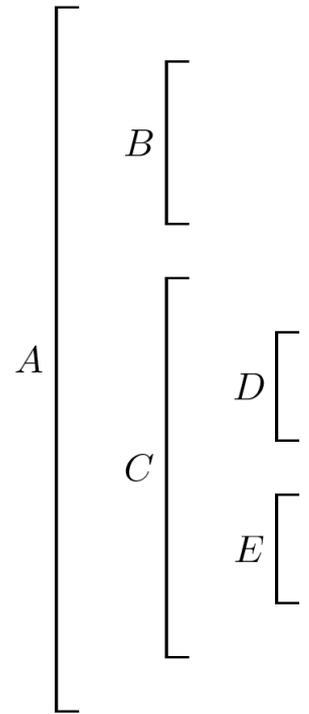
- Control link
  - o puntatore all'AR che era in testa alla pila
- Static link
  - o puntatore all'AR che contiene il blocco più vicino che racchiude il codice in esecuzione
- Analisi
  - o control link memorizza il flusso dinamico di esecuzione
  - o static link dipende dalla struttura sintattica del programma

# Static link



- Lo static link dell'AR di una funzione A è il puntatore al record di attivazione del blocco dove A è stata dichiarata
- La catena statica di un AR implementa la struttura sintattica dell'AR sulla catena dinamica
- Risolvere un riferimento non locale significa trovare l'istanza del record di attivazione dove il riferimento non locale è stato dichiarato*

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

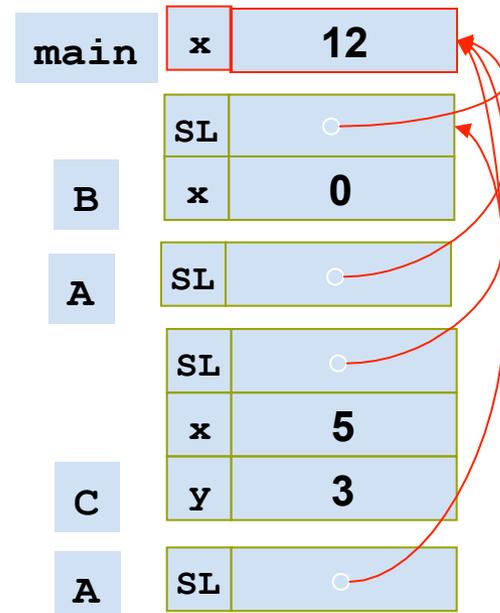
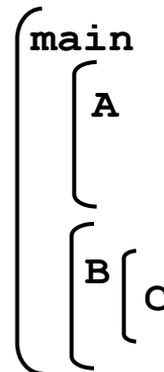


# Esempio

```

{ 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( );
}

```





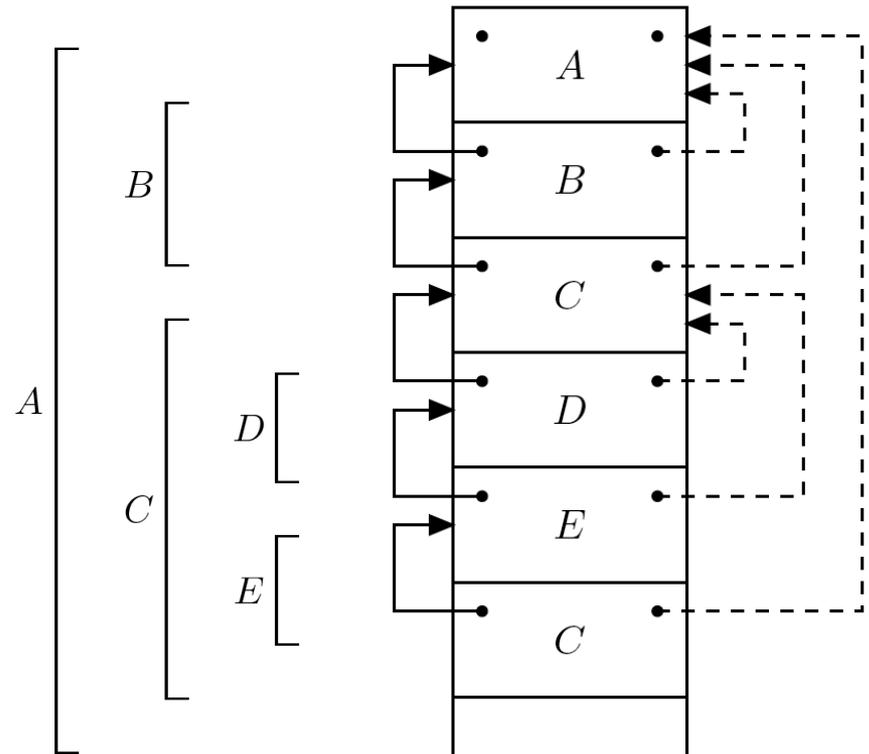
# Determinare la catena statica a run time

- Quali operazioni deve effettuare il supporto a tempo di esecuzione per determinare 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 staticamente)
  - proprio AR

## Il chiamante $C$ “conosce” l’annidamento dei blocchi

- quando  $C$  chiama  $P$ , sa se la definizione di  $P$  è
  - immediatamente inclusa in  $C$  ( $k=0$ );
  - in un blocco  $k$  passi fuori  $C$
  - 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  $k = 0$

- $C$  passa a  $P$  un puntatore al proprio AR

Se  $k > 0$

- $C$  risale la propria catena statica di  $k$  passi e passa a  $P$  il puntatore all’AR così determinato

# Static depth

- ✎ Si può determinare staticamente il valore dell'annidamento delle procedure
- ✎ Esempio

```
Main {  
    A {  
        B {  
        } B  
    } A  
  
    C {  
    } C  
} Main
```



# Static depth

- 👁️ **Static depth (SD)** = profondità statica della dichiarazione
- 👁️ SD può essere determinato staticamente: dipende solo dalla struttura sintattica del programma

```
Main {                -- SD = 0
  A {                 -- SD = 1
    B {               -- SD = 2
      } B
    } A
  } C                 --SD = 1
} C
} Main
```

# Chiamato esterno al chiamante



- Le regole dello scoping statico assicurano che affinché il chiamato sia visibile si deve trovare in un blocco esterno che includa il blocco del chiamante
- Questo implica che l'AR che contiene la dichiarazione del chiamato è già presente sullo stack
- Assumiamo che
  - $SD(\text{Chiamante}) = n$
  - $SD(\text{Chiamato}) = m$
  - distanza statica tra chiamante e chiamato  $n-m$
  - il chiamante deve fare  $n-m$  passi lungo la sua catena statica per definire il valore del puntatore della catena statica del chiamato

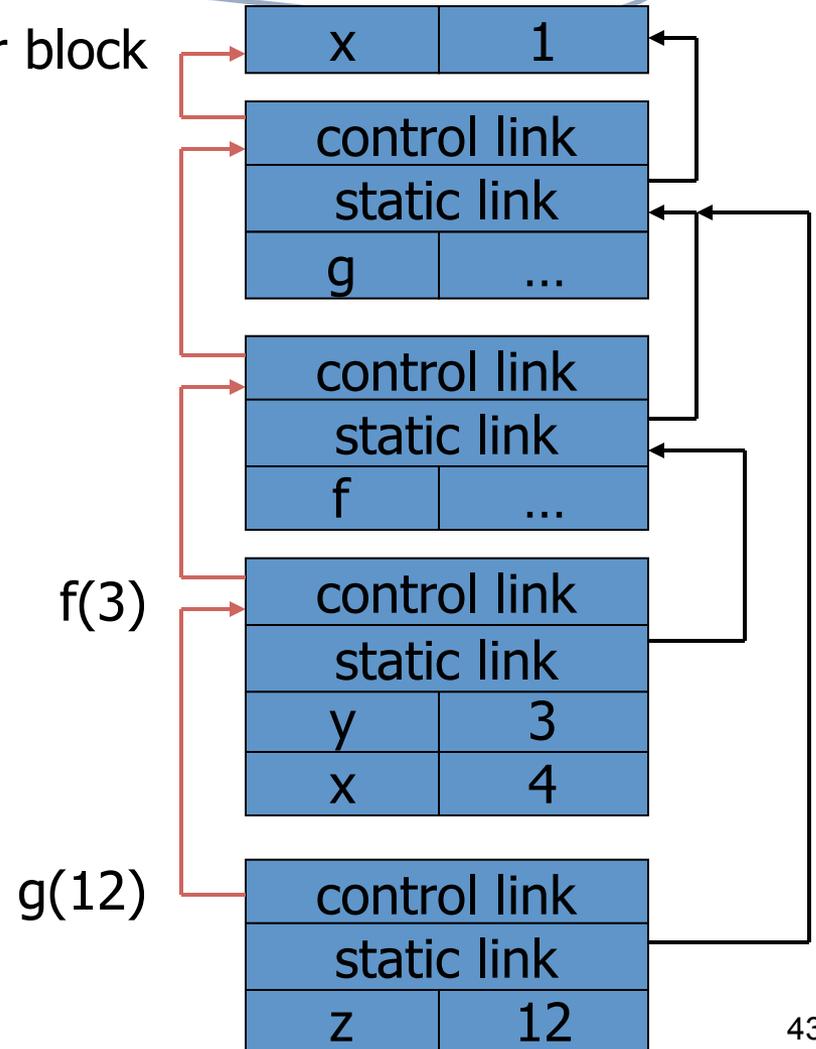
# Static scope: access links

```

var x = 1;
function g(z) = { return x+z; }
  function f(y) =
    { var x = y+1;
      return g(y*x); }
f(3);

```

outer block



# Funzioni come valori



- ✎ Nei **linguaggi funzionali** le funzioni tipicamente sono **valori esprimibili** (possono essere risultato della valutazione di espressioni)
- ✎ Consideriamo i seguenti due casi
  - funzione passata come parametro attuale (semplice)
  - funzione restituita come risultato di un'altra funzione: può essere utilizzata nel seguito della computazione (più complicato)



# Parametri funzionali

Haskell

```
int x = 4;
fun f(y) = x*y;
  fun g(h) = let
    int x=7
    in
    h(3) + x;
  g(f);
```

Pseudo-JavaScript

```
{ var x = 4;
  { function f(y) {return x*y};
    { function g(h) {
      var x = 7;
      return h(3) + x;
    };
    g(f);
  } } }
```

Due dichiarazioni per la variabile **x**

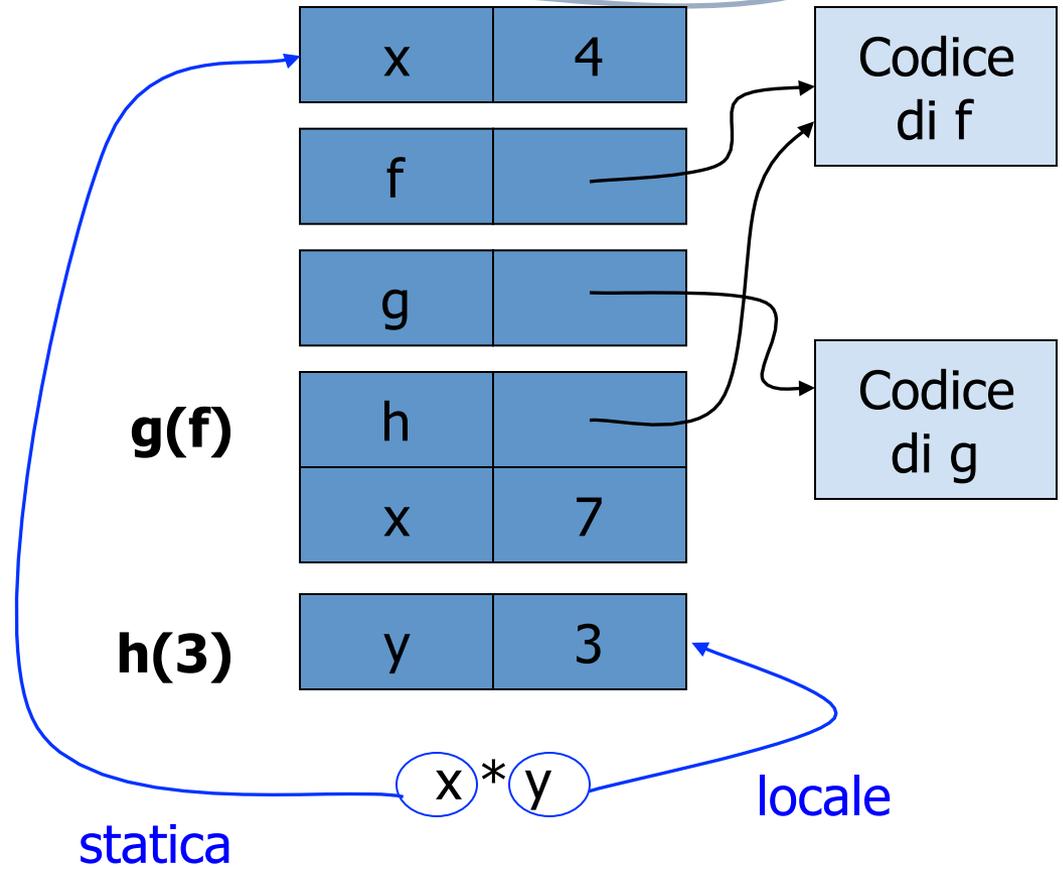
Quale deve essere usata nella chiamata **g(f)**?

# Parametri funzionali: static scope

```

int x = 4;
fun f(y) = x*y;
  fun g(h) =
    let
      int x=7
    in
      h(3) + x;
    g(f);

```



Come si determina?

# It does happen!



```
{ var x = 4;
  { function f(y) { return x*y; }
    { function g(h) {
      var x = 7;
      return h(3) + x; }
      g(f); }
    }
  }
```

Valutate questo codice JavaScript su [repl.it](https://repl.it)



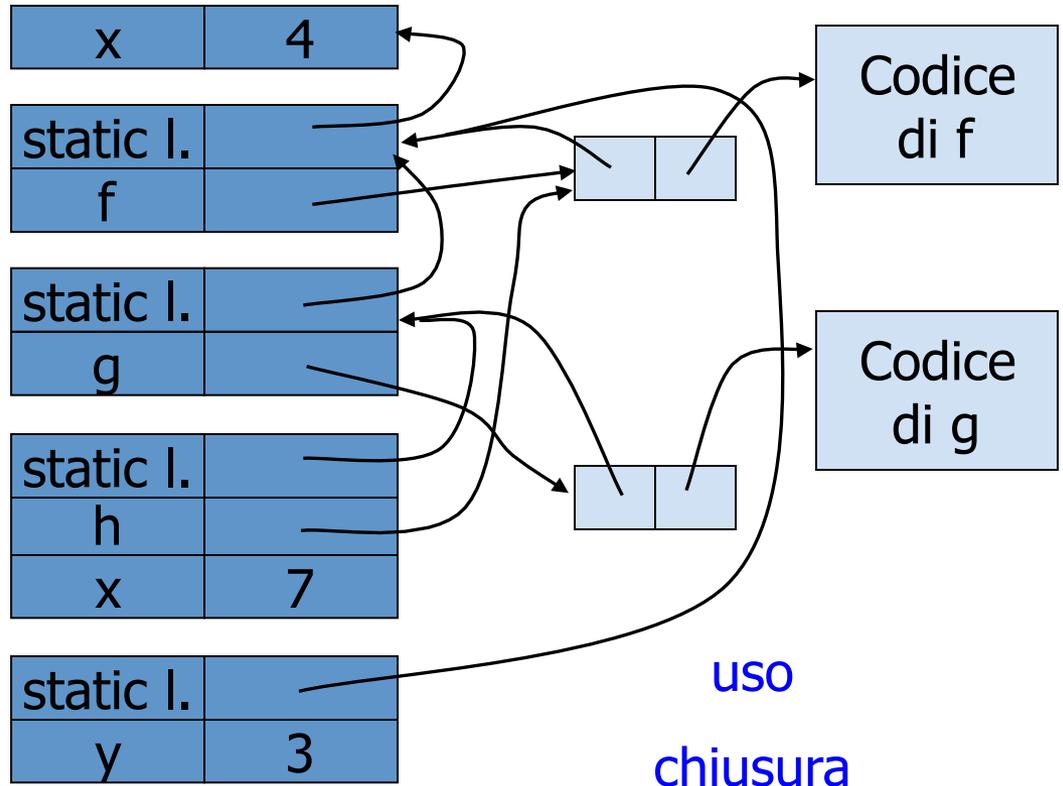
# Chiusure

- ✎ Il valore di una funzione trasmessa come parametro è una coppia denominata **chiusura**
  - $closure = \langle env\_dichiarazione, codice\_funzione \rangle$
- ✎ Quando il parametro formale (funzionale) viene invocato
  - si alloca sullo stack l'AR della funzione
  - si mette come valore del **puntatore di catena statica** il puntatore a *env\_dichiarazione*

# Struttura del run time

```

{ var x = 4;
  { function f(y) { return x*y };
    { function g(h) {
      int x = 7;
      return h(3) + x;
    };
    g(f);
  } } }
  
```



# Argomenti funzionali



- ✎ Si usano le **chiusure** per mantenere l'informazione sull'ambiente presente al momento della dichiarazione
- ✎ Si usa la chiusura per determinare il **puntatore di catena statica**



# Funzioni come risultato

- 🦋 Funzione che restituisce come valore una nuova funzione
  - bisogna congelare l'ambiente dove la funzione è "dichiarata"
- 🦋 Esempio

```
function compose(f, g)
  { return function(x) { return g(f(x)) } };
```
- 🦋 Funzione "dichiarata" dinamicamente
  - la funzione può avere variabili non locali
  - valore restituito è una chiusura **<env, code>**
  - **attenzione:** l'AR cui punta **env** non può essere distrutto finché la funzione può essere usata (**retention**)

# Funzioni con “stato”



## OCaml

- 🐼 L'ambiente non locale di una funzione può essere sfruttato per incapsulare una variabile, dotando la funzione di “stato”

```
# let mk_counter(init: int) = begin  
    let count = ref init in
```

Riferimento a  
variabile non locale,  
preservata tra  
diverse attivazioni

```
    let counter(inc: int) =  
        (count := !count + inc; !count)  
    in counter (* funzione come risultato *)
```

```
end;;
```

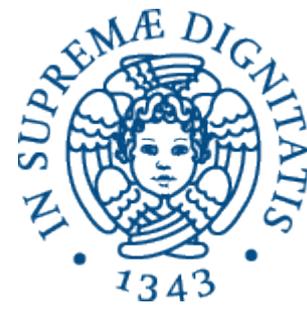
```
val mk_counter : int -> int -> int = <fun>
```

```
# let c = mk_counter 1;;
```

```
val c : int -> int = <fun>
```

```
# c(2) + c(2);; (* cosa restituisce? *)
```





---

# Scope dinamico

# Regole scope dinamico



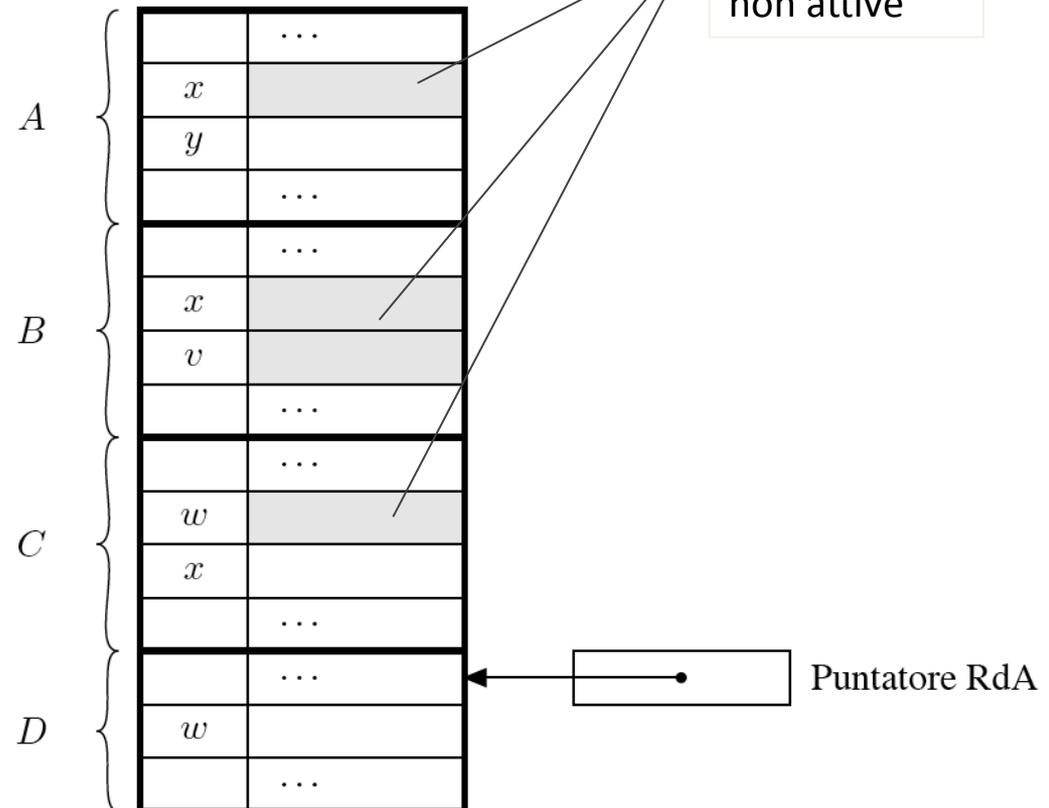
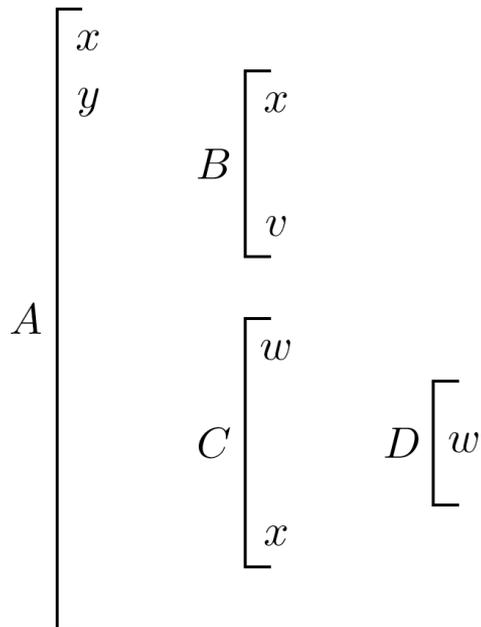
- ✎ Con scope dinamico l'associazione nomi-oggetti denotabili dipende
  - dal flusso del controllo a run time
  - dall'ordine con il quale i sottoprogrammi sono chiamati
- ✎ La regola generale è semplice: l'associazione corrente per un nome è quella determinata per ultima nell'esecuzione (non ancora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

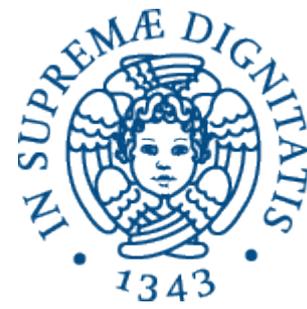
# Implementazione ovvia

🦋 Ricerca per nome risalendo la pila

🦋 Esempio

- chiamate A, B, C, D





---

# Record di attivazione: implementazione

# Record di attivazione



- ✎ La strutturazione dei vari campi del record di attivazione cambia a seconda del linguaggio e dell'implementazione
- ✎ Gli identificatori generalmente non vengono memorizzati nell'AR (se il linguaggio ha controllo statico dei tipi) ma sono sostituiti dal compilatore con un indirizzo relativo (offset) rispetto a una posizione fissa dell'AR

# Esempio

```
{ int x;  
  int arr[2];  
  char * s;  
  x = 4;  
  arr[0] = 10;  
  arr[1] = 11;  
  s = "bb";  
}
```

<b>x</b>	<b>4</b>
<b>arr[0]</b>	<b>10</b>
<b>arr[1]</b>	<b>11</b>
<b>s[0]</b>	<b>'b'</b>
<b>s[1]</b>	<b>'b'</b>
<b>s[2]</b>	<b>'\0'</b>

C-Standard: A C-string consists of an array of characters terminated by the null character '\0'

# Calcolo offset

x	4
arr[0]	10
arr[1]	11
s[0]	'b'
s[1]	'b'
s[2]	'\0'



compilation  
int = 2 byte

base

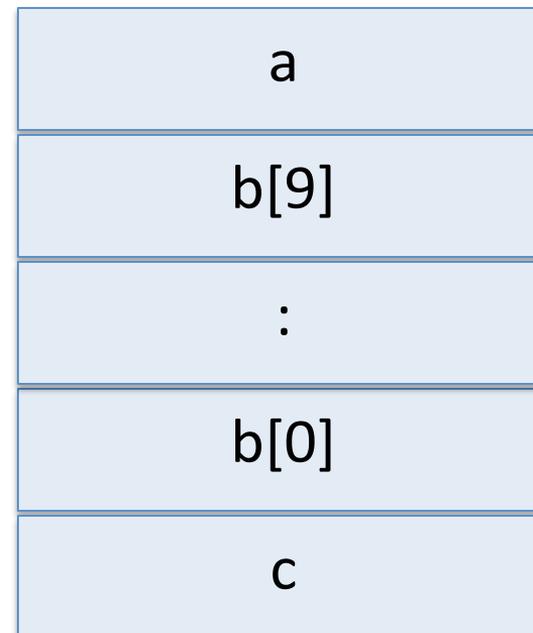
4
10
11
'b'
'b'
'\0'

**access(x) = base**  
**access(arr[1]) = base + 4byte**

# Allocazione di array

- 🦋 Array di dimensione fissa: facile!!
  - calcolo offset immediato a tempo di compilazione

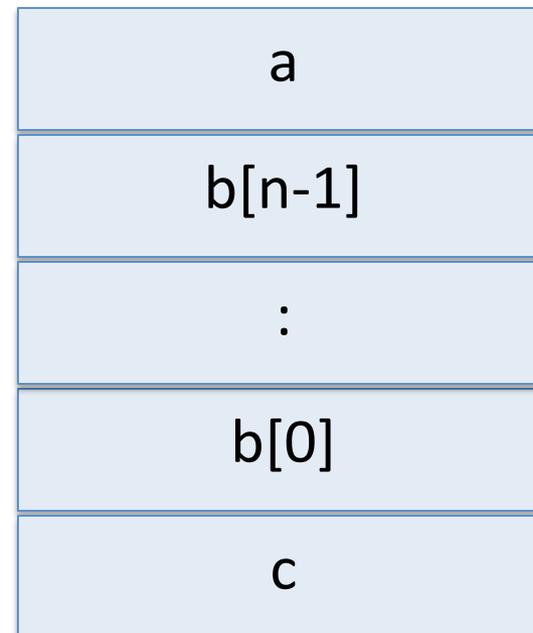
```
void foo( ) {  
  int a;  
  int b[10];  
  int c;  
}
```



# Allocazione di array

- 🦋 Array di dimensione variabile: più difficile!!
  - calcolo offset a tempo di compilazione?
    - ✓ è possibile determinarlo?

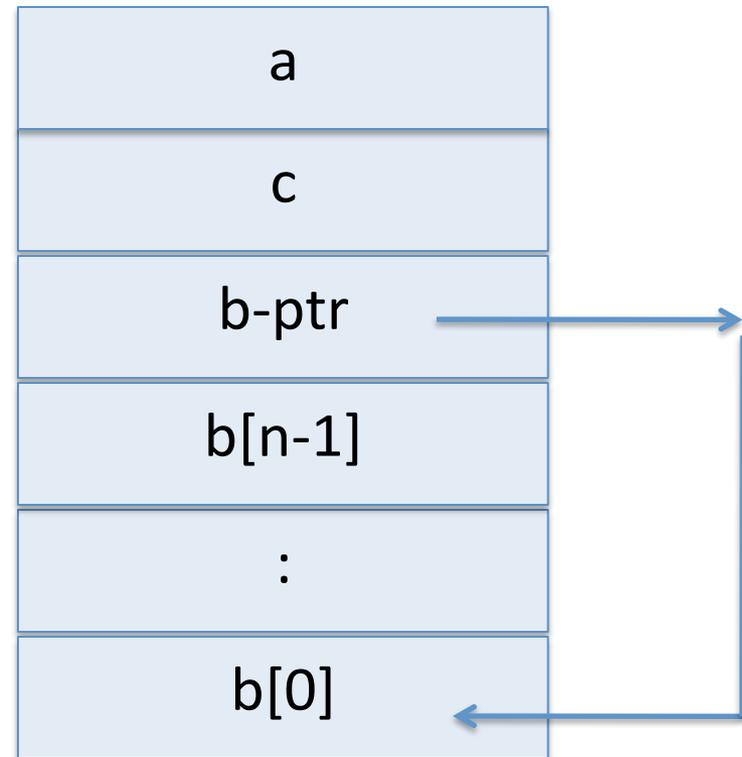
```
void foo(int n) {  
  int a;  
  int b[n];  
  int c;  
}
```



# Allocazione di array

- 🦋 Array di dimensione variabile
  - calcolo offset a tempo di compilazione

```
void foo(int n) {  
  int a;  
  int b[n];  
  int c;  
}
```



ponendo una dimensione limite



# Scope statico e analisi statica



# In brevissimo

- 🦋 Ambiente non locale con scope statico
  - il numero di passi che a tempo di esecuzione vanno fatti lungo la catena statica per trovare l'associazione (non locale) per l'identificatore "x" è uguale alla differenza fra le profondità di annidamento del blocco nel quale "x" è dichiarato e quello in cui è usato
- 🦋 Ogni *riferimento* a un identificatore *ide* nel codice può essere staticamente tradotto in una coppia (m,n) di numeri interi
  - m è la differenza fra le profondità di nesting dei blocchi (0 se *ide* si trova nell'ambiente locale)
  - n è la posizione relativa – offset – (partendo da 0) della dichiarazione di *ide* fra quelle contenute nel blocco

# Valutazione



- ✉ Efficienza nella rappresentazione
  - l'accesso diventa efficiente (non c'è più ricerca per nome)
- ✉ Si può economizzare nella rappresentazione degli ambienti locali che non necessitano più di memorizzare i nomi