
PROGRAMMAZIONE 2

20. Funzioni e procedure

Breve storia dei sotto-programmi

- 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 sotto-programmi

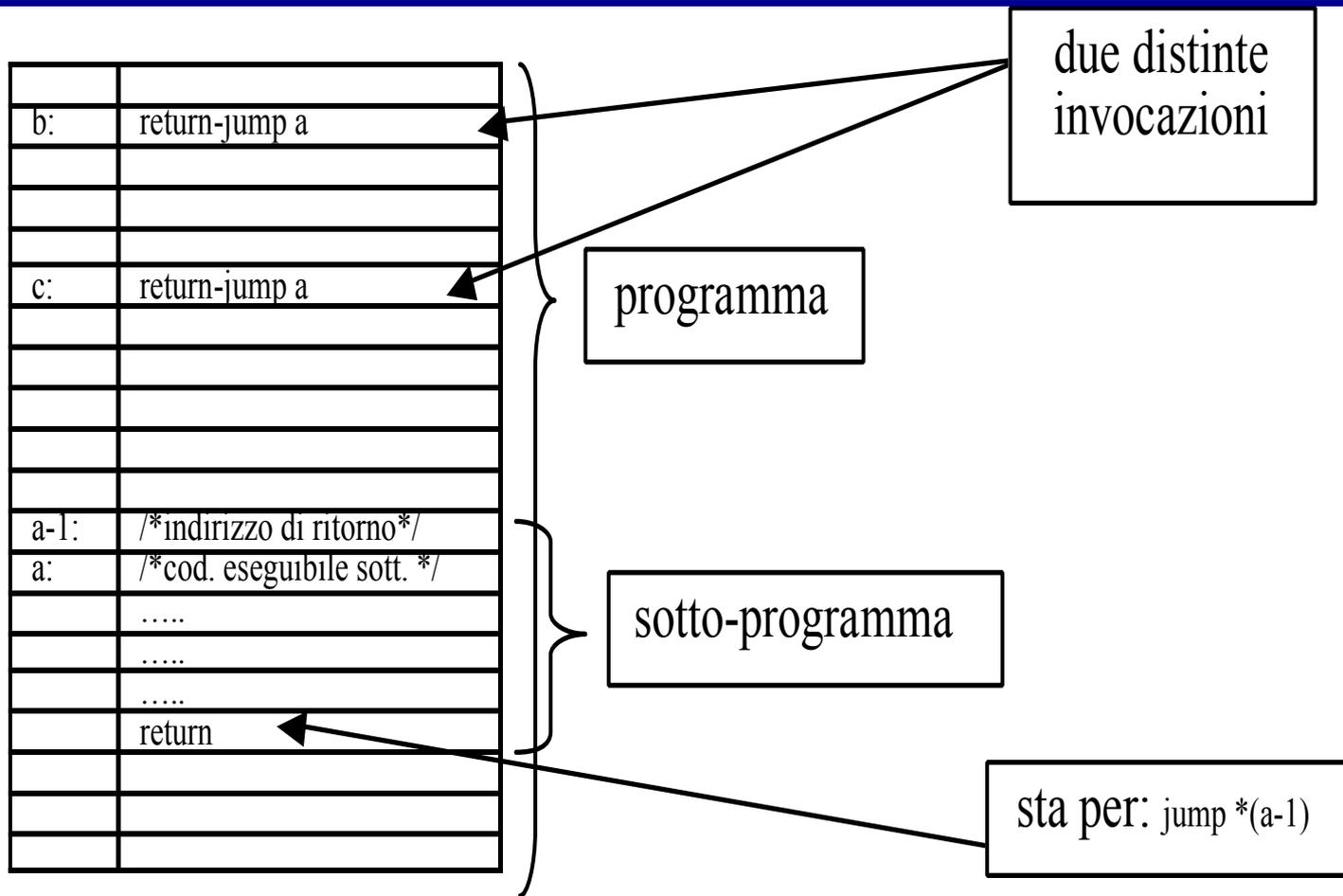
- **Astrazione sul controllo:** si riduce anche l'occupazione di memoria se esiste un meccanismo che permette al programma principale
 - di *trasferire il controllo* a una *unica copia del sotto-programma* memorizzata separatamente
 - di *riprendere il controllo* quando l'esecuzione del frammento è terminata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

Breve storia dei sotto-programmi

- *Ancora meglio se permettiamo astrazione via parametrizzazione*
 - *astrando 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 è trasferito alla cella **a** (**entry point della sub-routine**)
 - l'indirizzo dell'istruzione successiva del chiamante l'istruzione (**b + 1**) viene memorizzato in qualche posto noto, per esempio nella cella (**a - 1**) (punto di ritorno)
- quando nella **sub-routine** si esegue una operazione di **return**
 - il controllo ritorna all'istruzione (del programma chiamante) memorizzata nel punto di ritorno



Archeologia: FORTRAN

- *Una **sub-routine** 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 sub-routine à la FORTRAN

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

Sub-routine à la FORTRAN

- *Osservazione: non è compatibile con la ricorsione*
 - *la macro-espansione 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*

Sotto-programmi: attivazione

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

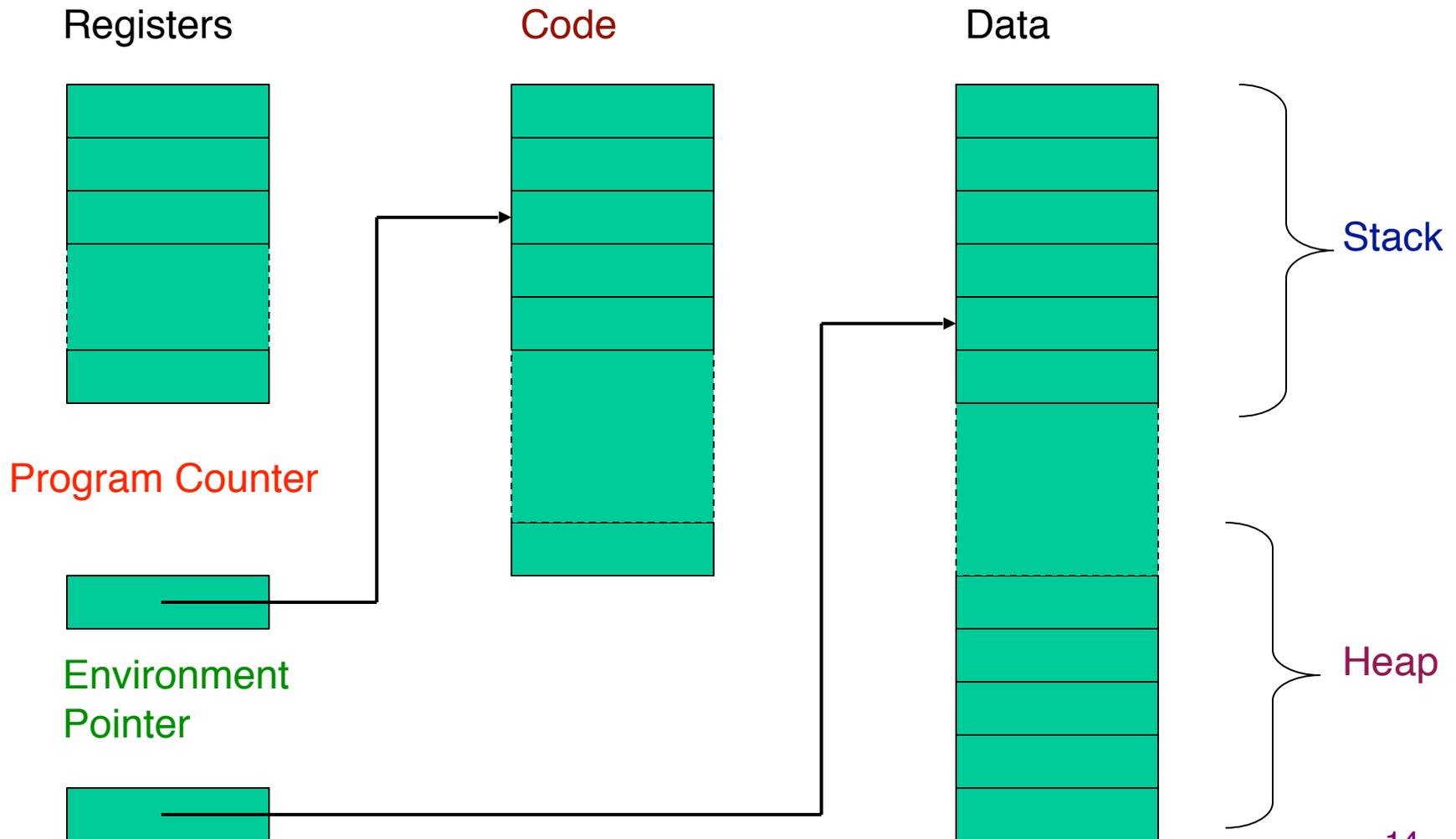
Le strutture di implementazione

- Invece delle informazioni staticamente associate al codice compilato di **FORTRAN**
 - punto di ritorno, parametri, ambiente e memoria localesi usano i record di attivazione (frame)
 - contenenti le stesse informazionima associati **dinamicamente** alle varie chiamate di sotto-programmi
- Dato che l'accesso ai **sotto-programmi** segue una **politica LIFO**
 - l'ultima attivazione creata nel tempo è la prima che ritornapossiamo organizzare i **record di attivazione (frame)** in una **pila**

Cosa è un vero sotto-programma

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

Modello di macchina hw



Meccanismo **call/return** di sotto-programma

- **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 sotto-programma



- **Chiamato (prologo)**
 - 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, procedure senza nome e senza parametri
 - seguono una politica LIFO

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
con spazio per **z**

Assegna valore a **z**

Pop **AR** per blocco interno

Pop **AR** per blocco esterno

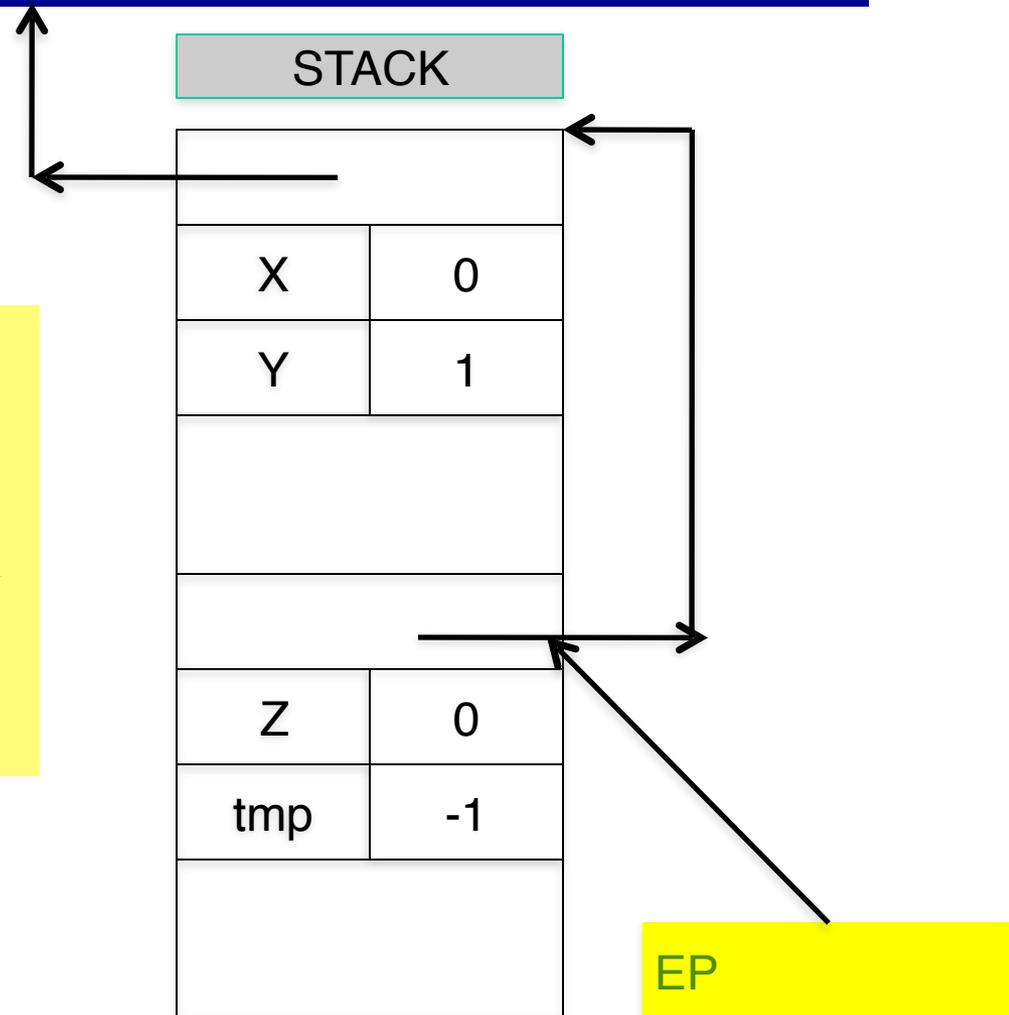
Occorre prevedere spazio per memorizzare i risultati intermedi

Esempio: blocco in-line

```

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

```

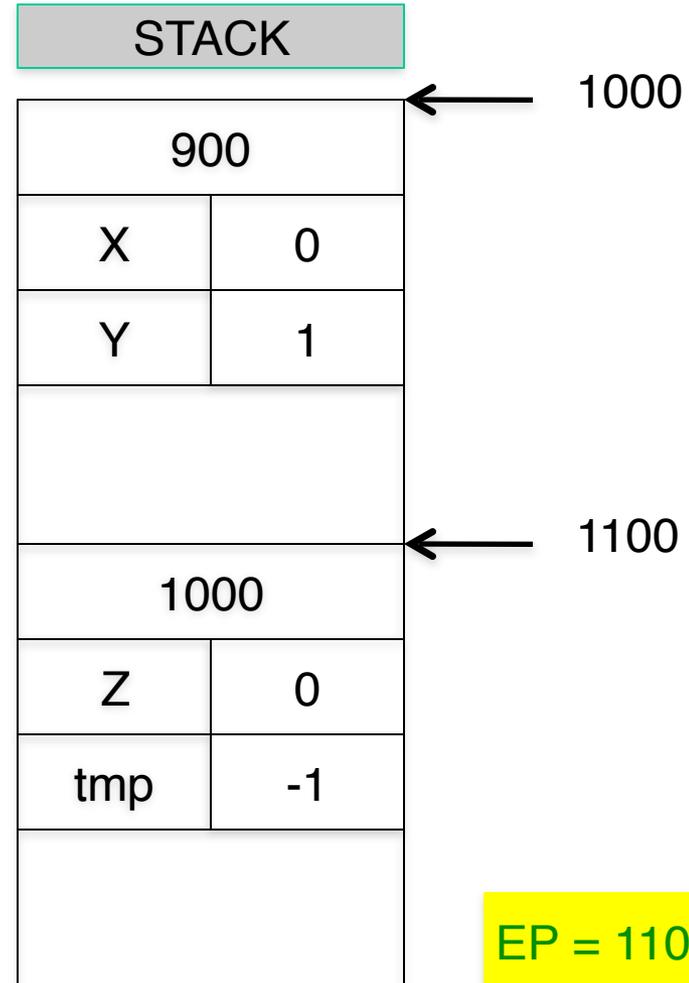


Esempio: blocco in-line

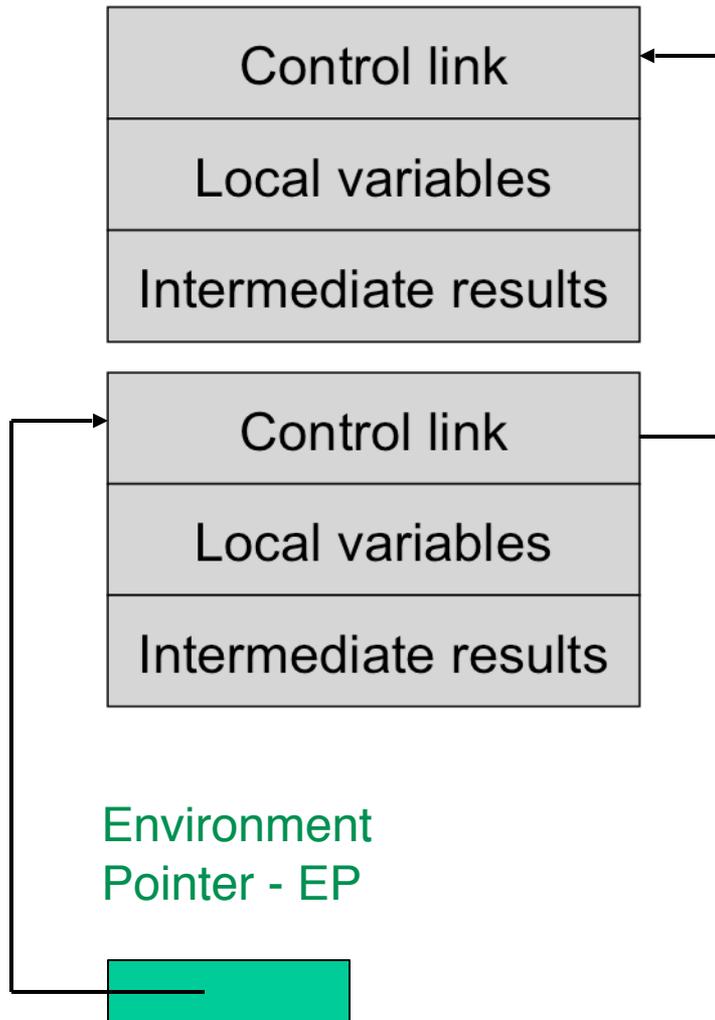
```

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

```



Record di attivazione: blocchi in-line



- **Control link**
 - puntatore (indirizzo base) 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

Control link
Puntatore di catena dinamica

Variabili locali

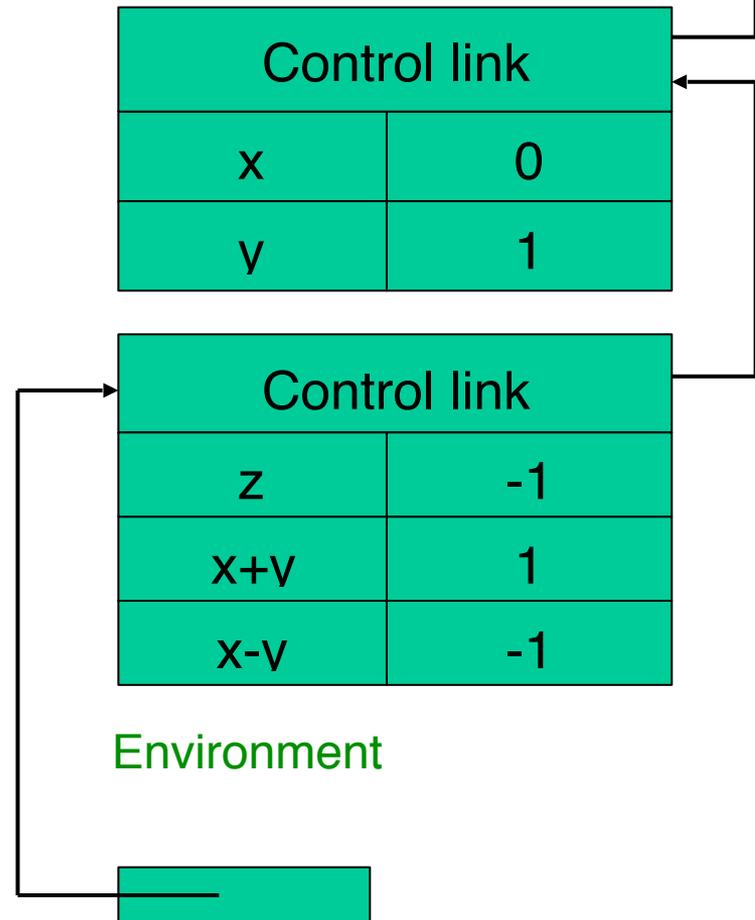
Risultati intermedi

Blocco in-line: 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
- Static scope
 - riferimenti non locali si risolvono nel più vicino blocco esterno
- Dynamic scope
 - riferimenti non locali si risolvono nell'AR precedente sullo stack

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

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 **referimento 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**
 - per risolvere un **referimento non locale?** Dipende dalla regola di scope
 - 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;
```

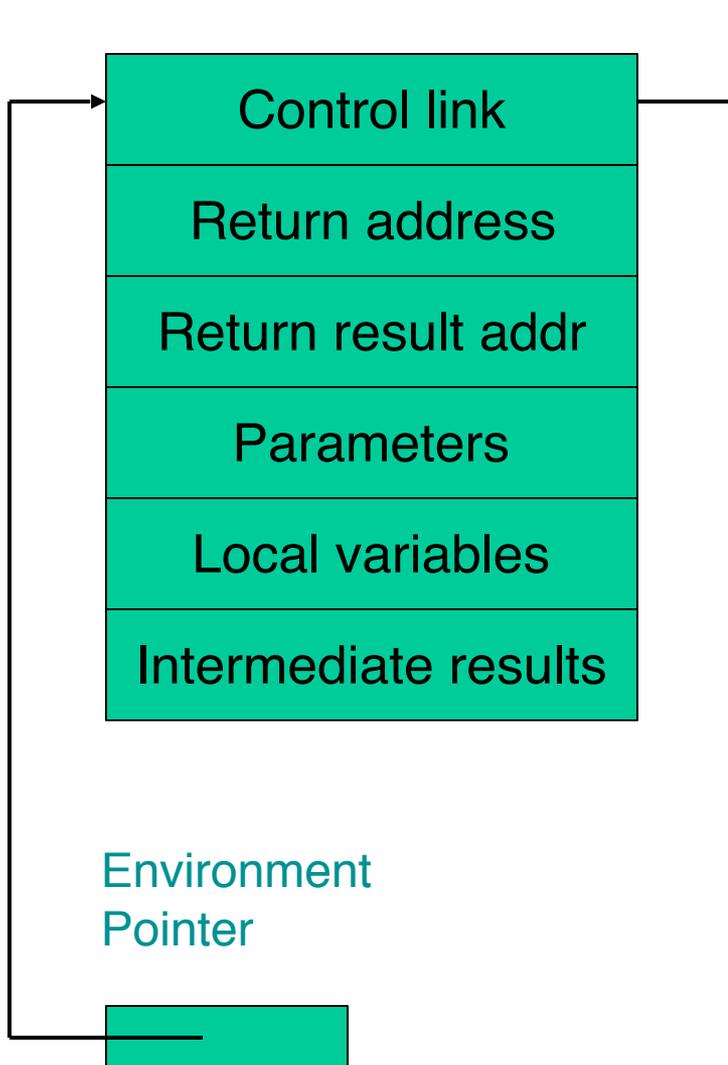
```
Funzioni C
<type> function f(<pars>)
  {
    <local vars>
    <function body>

  }
```

• *Cosa deve stare nel record di attivazione?*

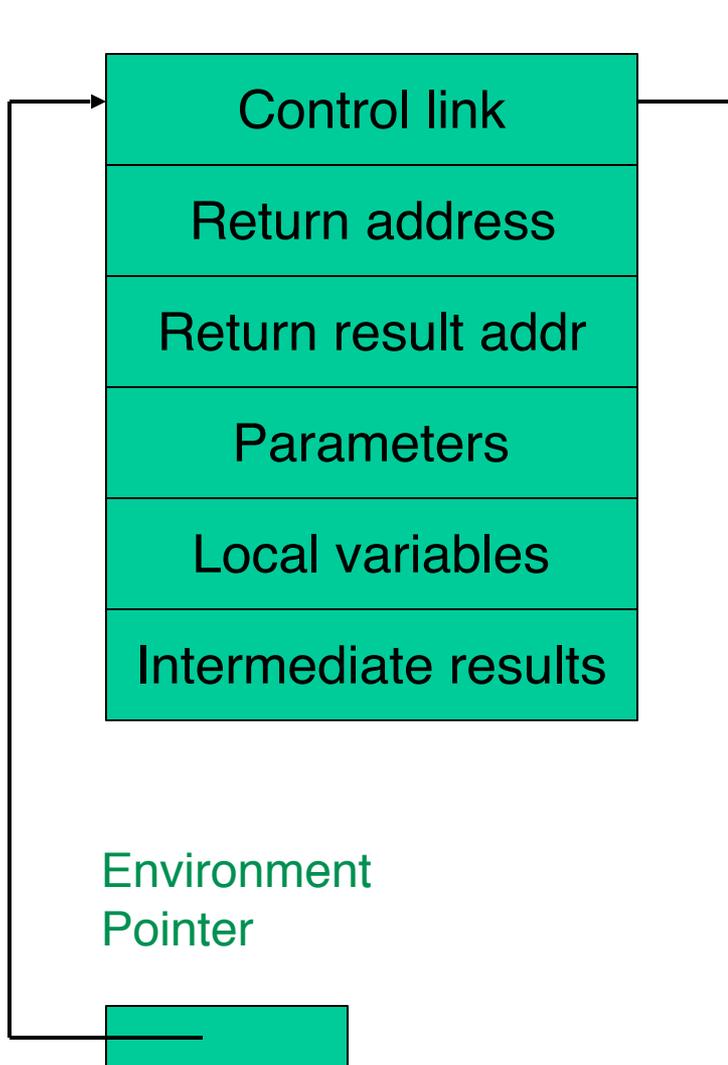
- parametri
- indirizzo di ritorno
- variabili locali, risultati intermedi
- valore restituito (caso part. di risultato intermedio)
- spazio per il valore restituito al momento del ritorno

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

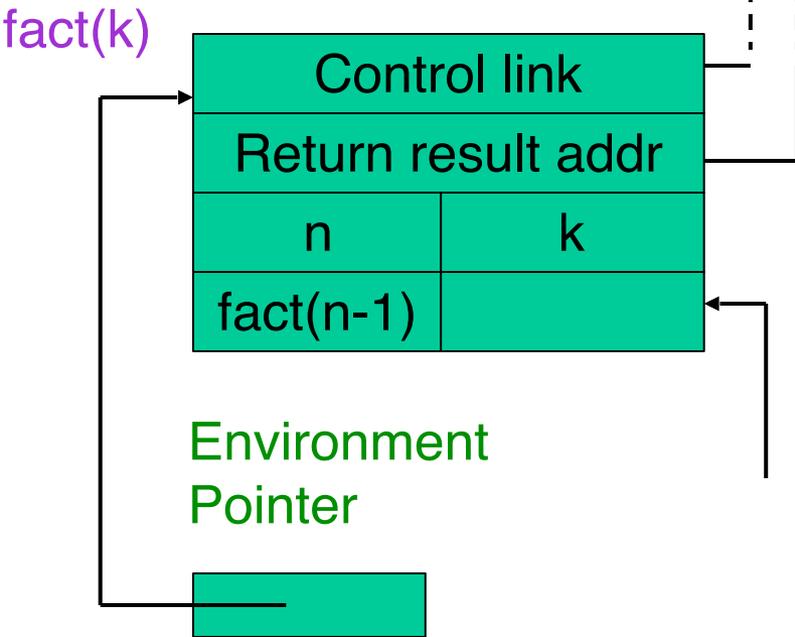
Esempio



- Il solito fattoriale


```
fact(n) = if n <= 1 then 1
          else n * fact(n-1)
```
- Return address
 - indirizzo dove memorizzare **fact(n)**
- Parameters
 - associazione tra **n** e il valore del **parametro attuale**
- Intermediate results
 - spazio per memorizzare il valore di **fact(n-1)**

Call & ...



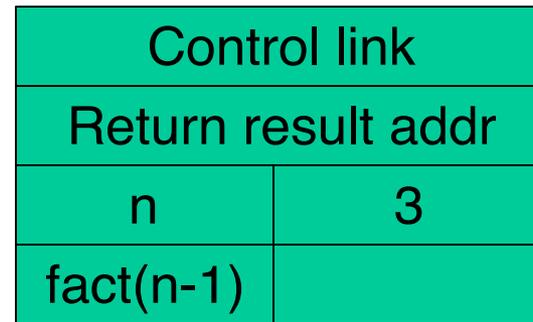
```

fact(n) =
if n <= 1 then 1
else n * fact(n-1)

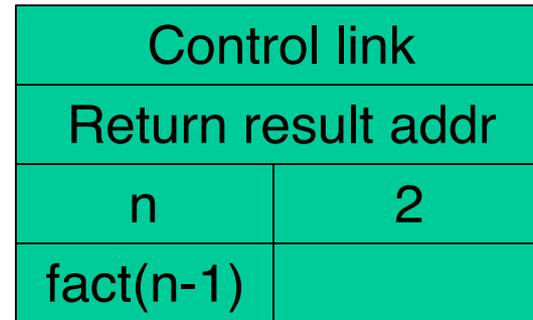
```

Per semplicità non inseriamo il valore del return address

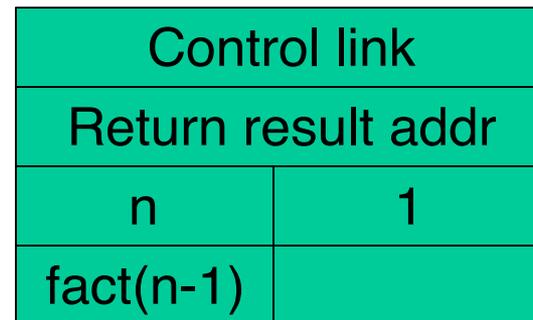
$fact(3)$



$fact(2)$

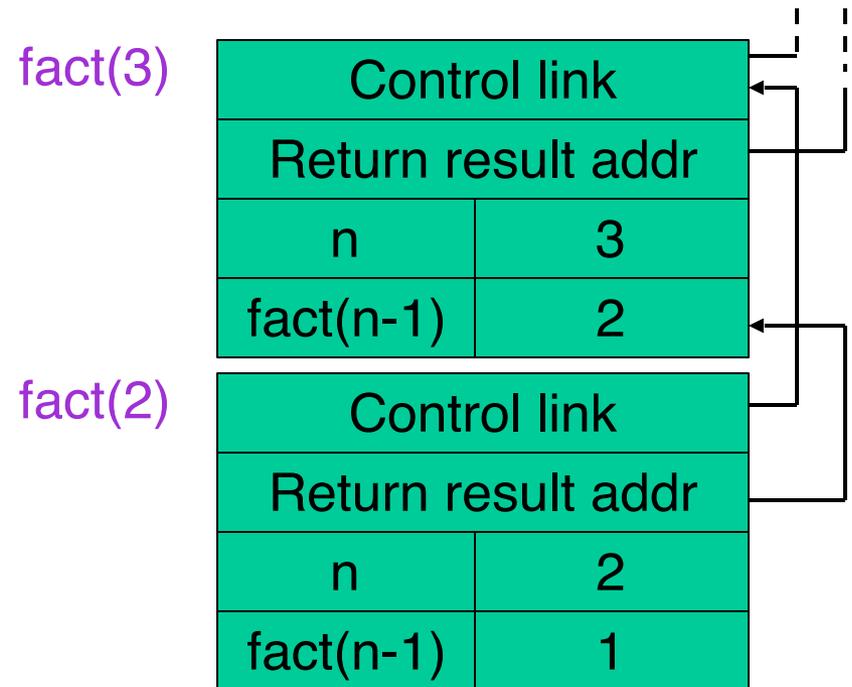
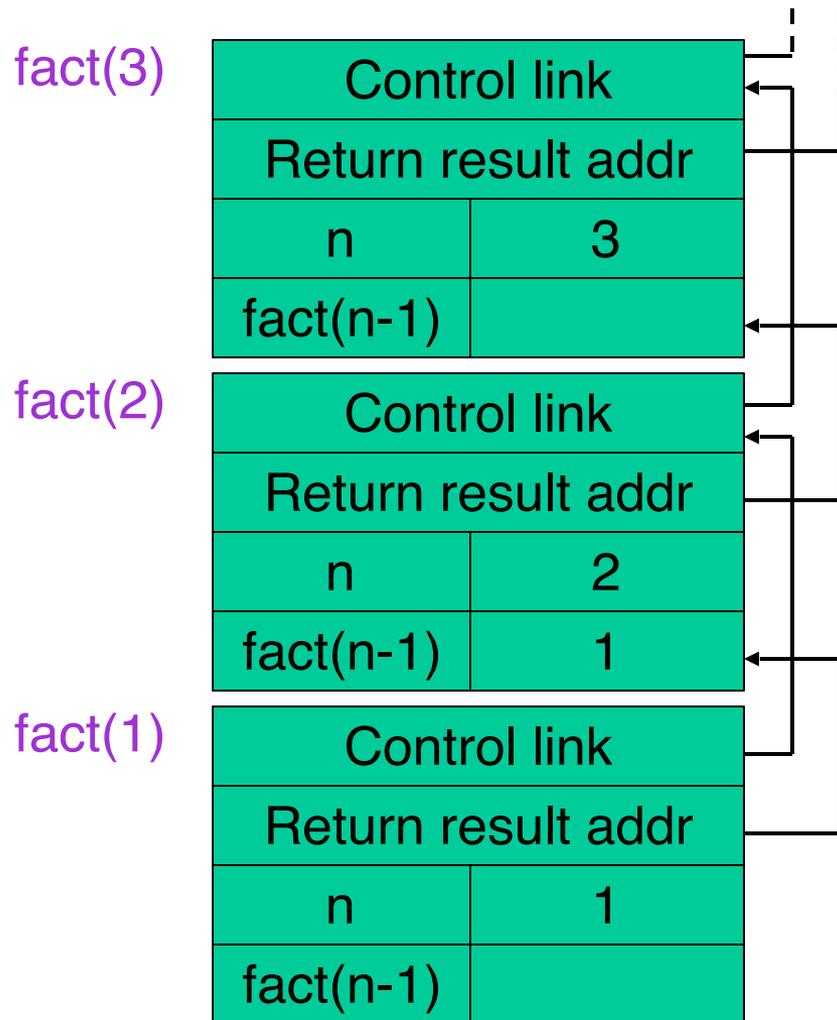


$fact(1)$



Continua →

..... & return



```

fact(n) =
if n <= 1 then 1
else n * fact(n-1)

```

Altri aspetti

- Variabili non locali?
- 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)

Passaggio dei parametri

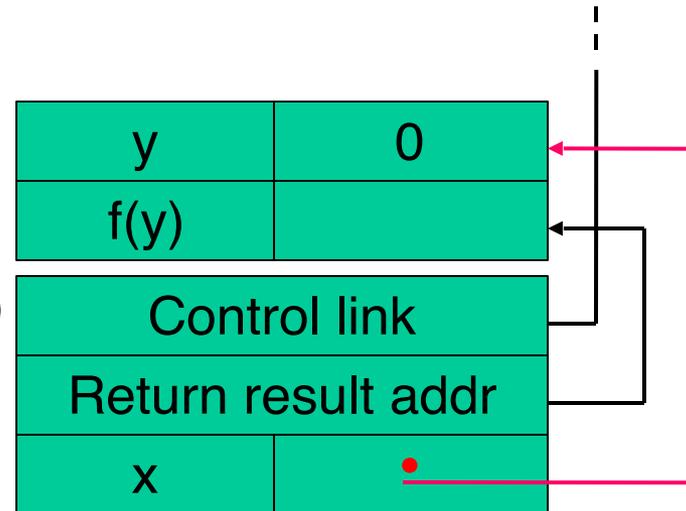
- **L-value** & **R-value**: Assegnamento $y := x$
 - *identificatore sulla sinistra dell'assegnamento denota la locazione e viene solitamente chiamato **L-value***
 - *identificatore sulla destra fa riferimento al contenuto della locazione e viene chiamato **R-value***
- **Per valore** memorizzare **R-value**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

Esempio

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

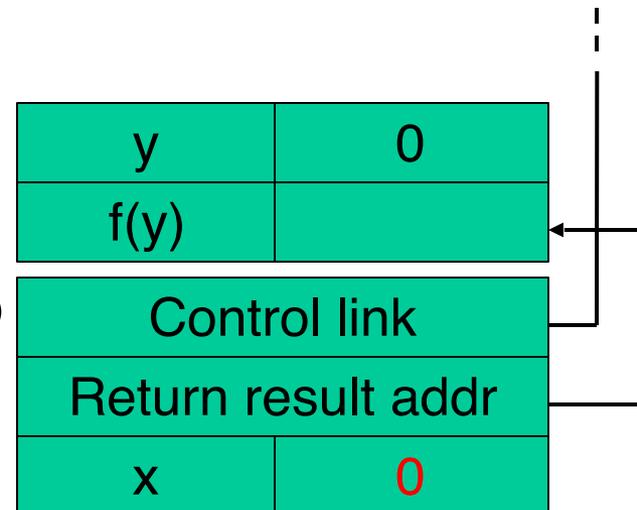
pass-per-ref

$f(y)$



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)

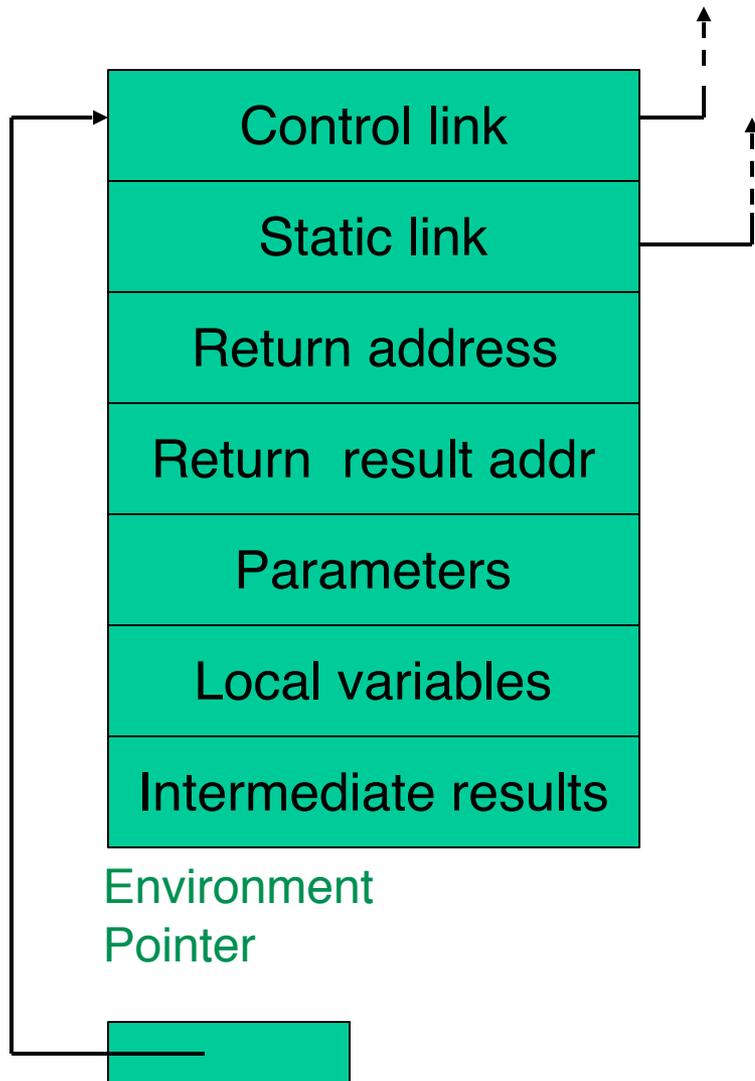
y	3
x	4

g(12)

z	12
---	----

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

Scoping statico: catena statica

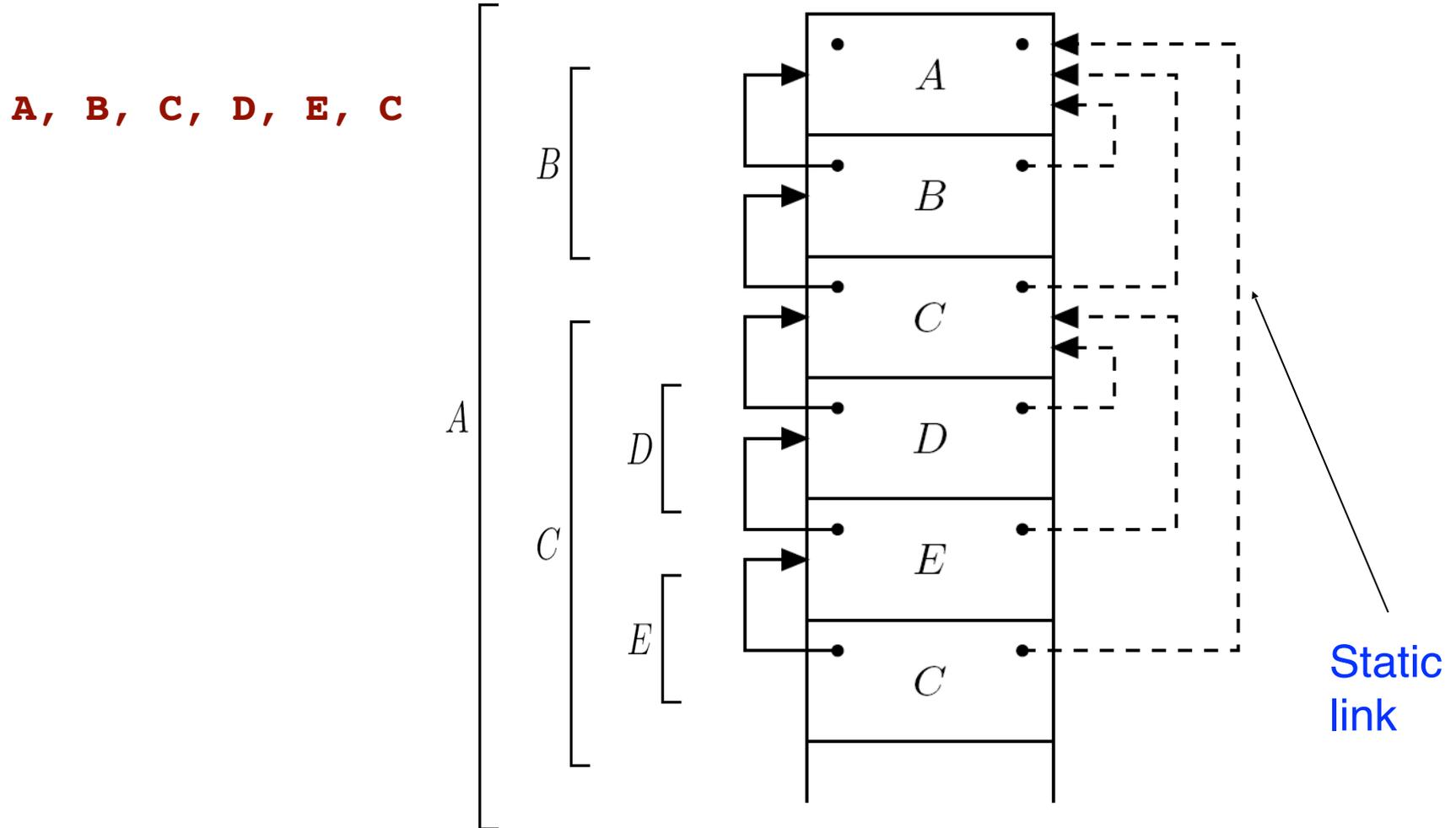


- **Control link**
 - puntatore all'AR che era in testa alla pila
- **Static link**
 - puntatore all'AR che contiene il blocco più vicino che racchiude la dichiarazione del codice in esecuzione
- **Analisi**
 - control link memorizza il flusso dinamico di esecuzione
 - static link dipende dalla struttura 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;B;C;D;E;C**

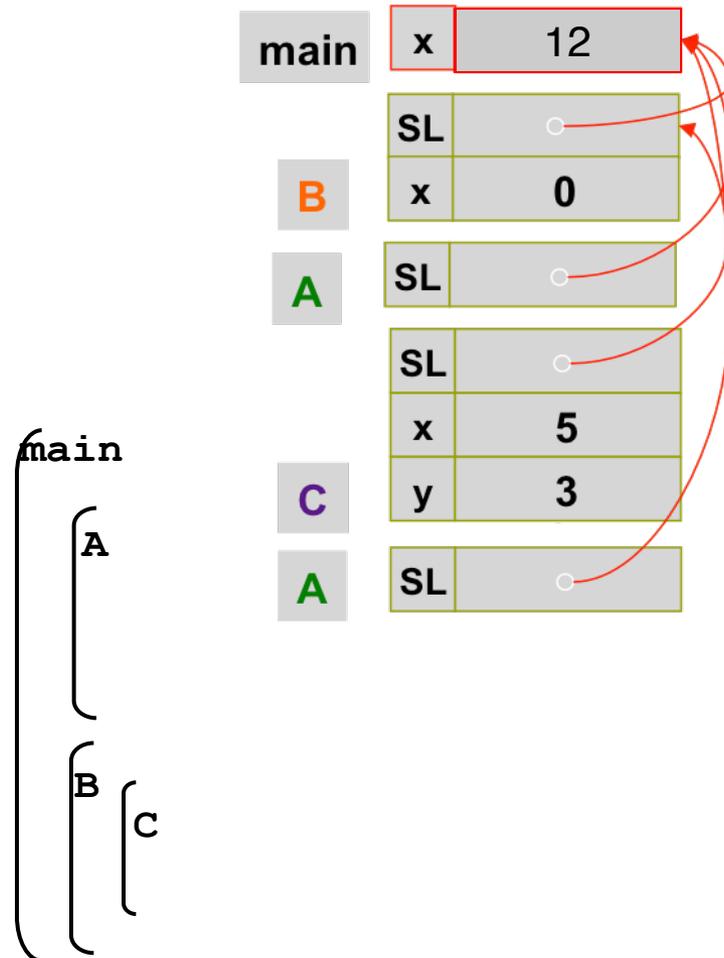


Esempio

```

{ 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**

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: **il chiamato deve essere dichiarato prima 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 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
    } A

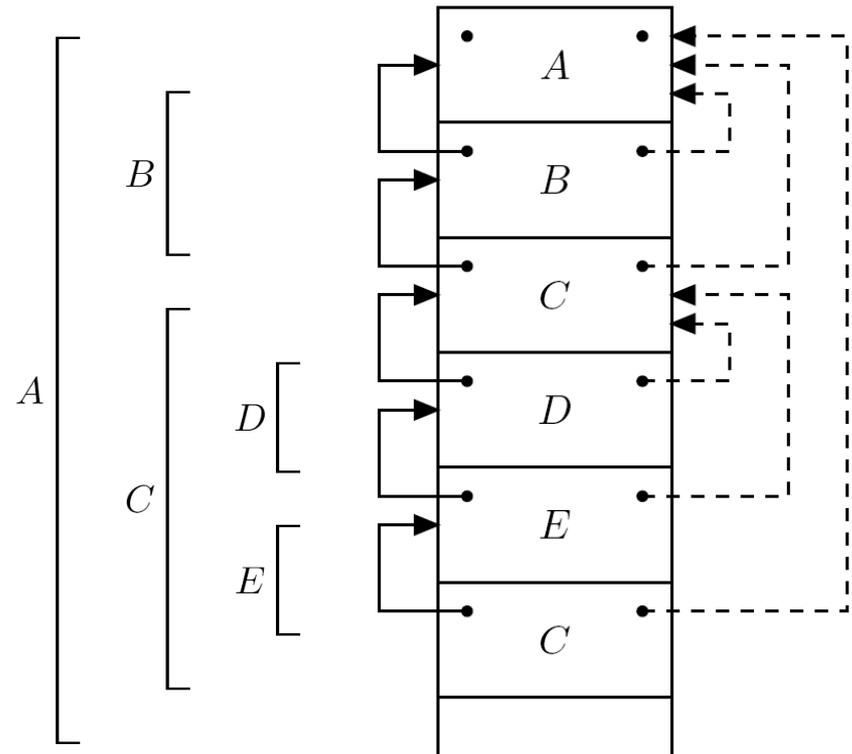
    C {                                -- SD = 1
    } C
} Main
```

Catena statica a run-time

- Il metodo **chiamante C** sa se la definizione del metodo **chiamato P** è
 - immediatamente inclusa in **C** ($k=0$);
 - in un blocco esterno k passi fuori **C**
 - nessun altro caso possibile (perché)?
- Se $k=0$
 - **C** passa a **P** un puntatore al proprio **AR**
- Altrimenti
 - **C** risale la **propria catena statica di k passi** e passa a **P** il puntatore all' **AR** così determinato

Esempio: catena statica a run-time

- nel caso a destra
 - chiamate: **A**, **B**, **C**, **D**, **E**, **C**
- con i dati di **catena statica**
 - **A**; (**B**,**0**); (**C**,**0**); (**D**,**1**); (**E**,**1**); (**C**,**0**)



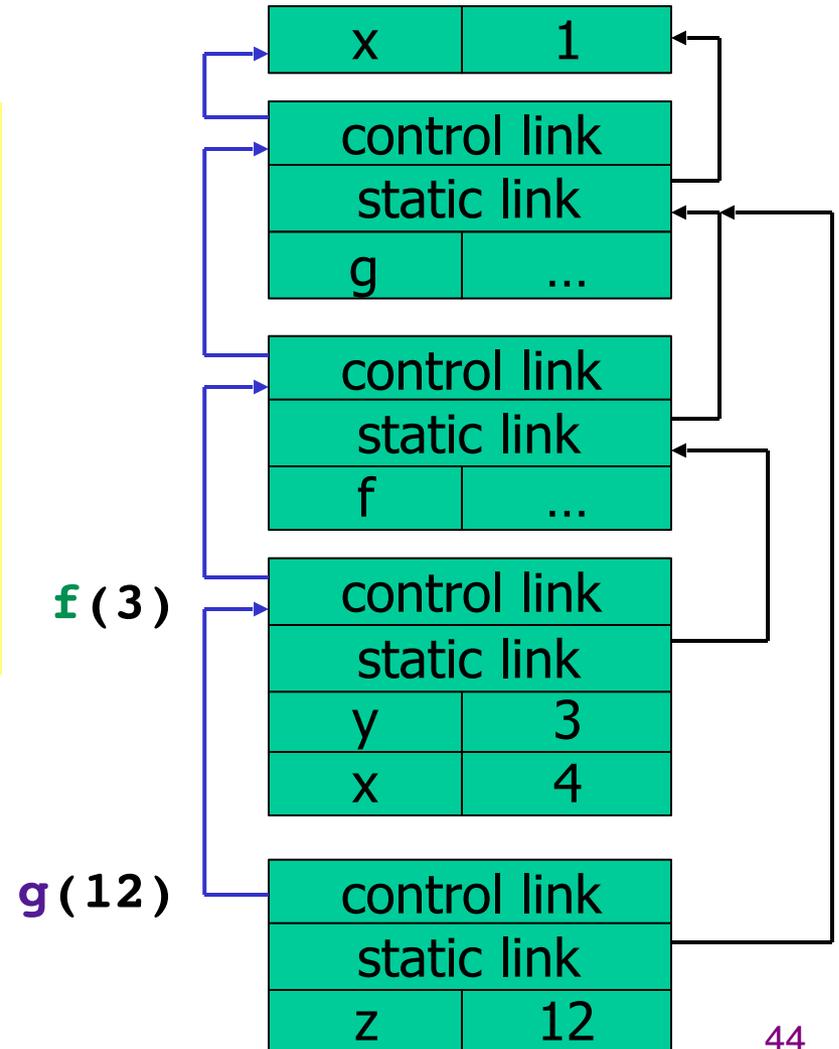
Static scope: catena statica

outer block

```

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

```



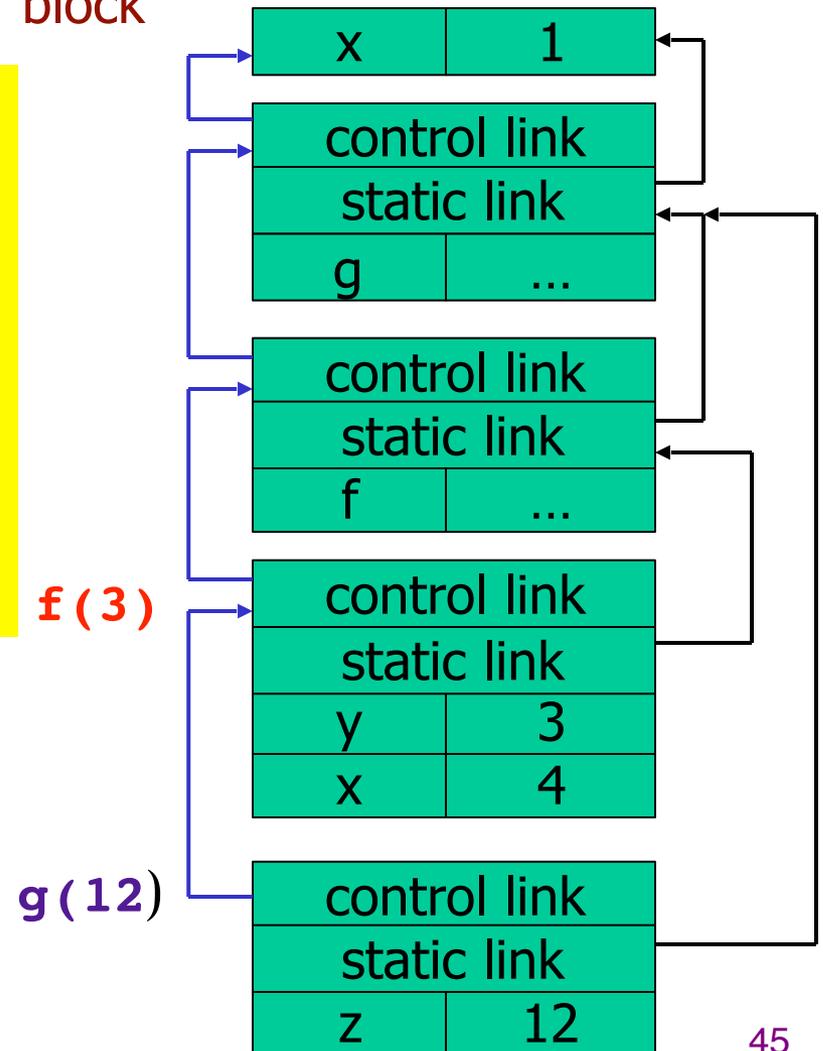
Static scope: catena statica

```

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

SD(g) = 1
SD(f(3)) = 3

outer block



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 è staticamente tradotto in una coppia **(m, n)** di interi
 - **m** è la differenza fra le profondità di nesting dei blocchi (0 se **ide** si trova nell'ambiente locale)
 - **n** 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*