

# *Macchine Astratte*

Luca Abeni

February 22, 2017

# ...E Macchine Astratte!

- L'algoritmo che esegue un programma non deve necessariamente essere implementato in hw...
- Implementazione software: **Macchina Astratta**
  - Insieme di algoritmi e strutture dati che permettono di **memorizzare** ed **eseguire** programmi
- Come per macchine fisiche (CPU), ad ogni macchina astratta è associato un suo linguaggio
  - $\mathcal{M}_{\mathcal{L}}$ : macchina astratta che capisce ed esegue il linguaggio  $\mathcal{L}$
  - $\mathcal{L}$  è il *linguaggio macchina* di  $\mathcal{M}_{\mathcal{L}}$
  - Programma: sequenza di istruzioni in  $\mathcal{L}$
- $\mathcal{M}_{\mathcal{L}}$  è un possibile modo per descrivere  $\mathcal{L}$

# Funzionamento di una Macchina Astratta

- Per eseguire un programma scritto in  $\mathcal{L}$ ,  $\mathcal{M}_{\mathcal{L}}$  deve:
  1. Eseguire operazioni elementari
    - In hw, ALU
  2. Controllare il flusso di esecuzione
    - Esecuzione non solo sequenziale (salti, cicli, etc...)
    - In hw, gestione PC
  3. Trasferire dati da / a memoria
    - Modalità di indirizzamento, ...
  4. Gestire la memoria
    - Allocazione dinamica, gestione stack, memoria dati / programmi, etc...

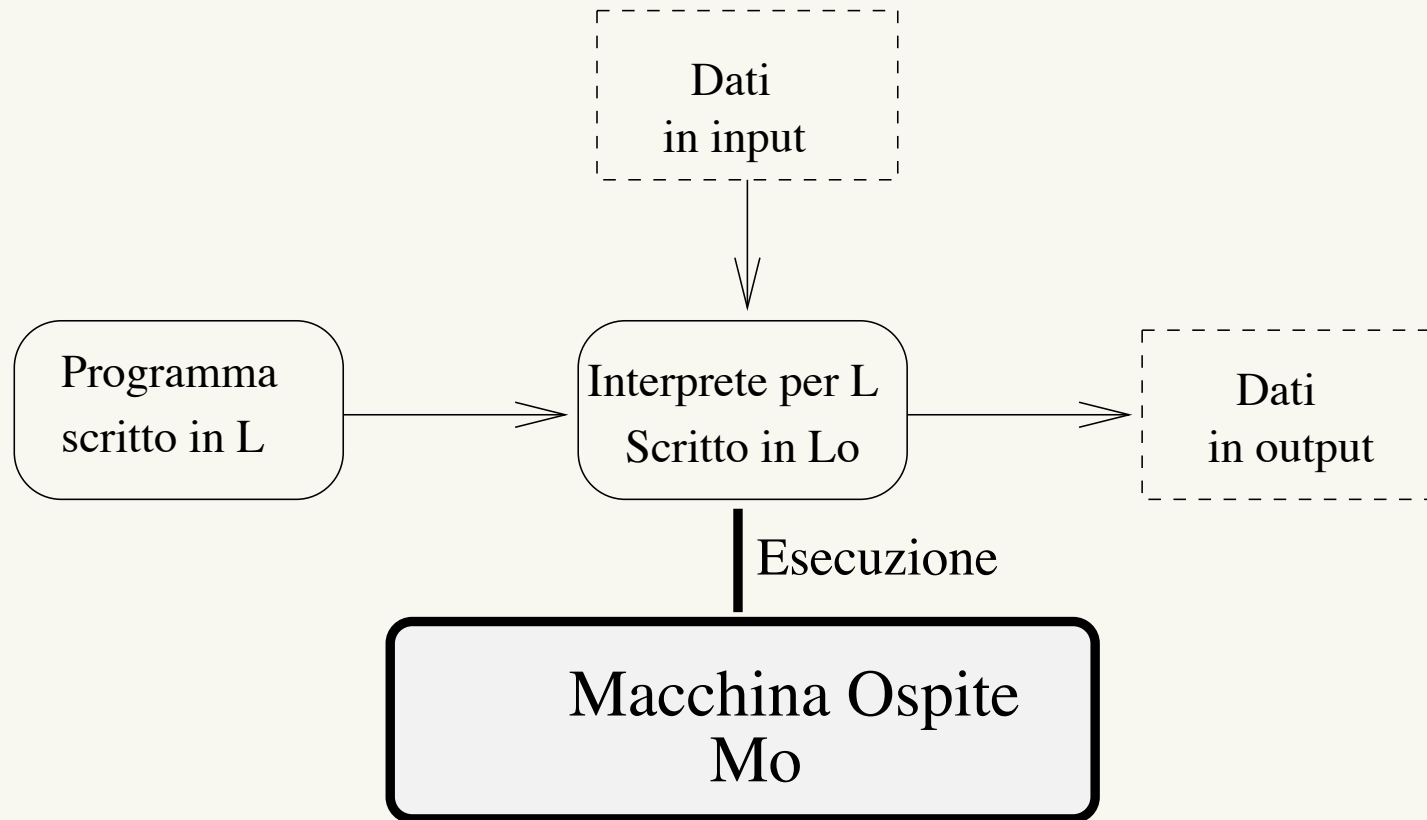
# Implementazione di un Linguaggio

- $\mathcal{M}_{\mathcal{L}}$  capisce il suo linguaggio macchina  $\mathcal{L}$ 
  - Un solo linguaggio macchina per macchina astratta
- $\mathcal{L}$  è capito/eseguito da varie (infinite) macchine astratte
  - Possono differire per implementazione, strutture dati, etc...
- Implementazione di un linguaggio  $\mathcal{L}$ : realizzazione di una macchina astratta  $\mathcal{M}_{\mathcal{L}}$  in grado di eseguire programmi scritti in tale linguaggio
  - Implementazione hw, sw o firmware

# Implementazione Software

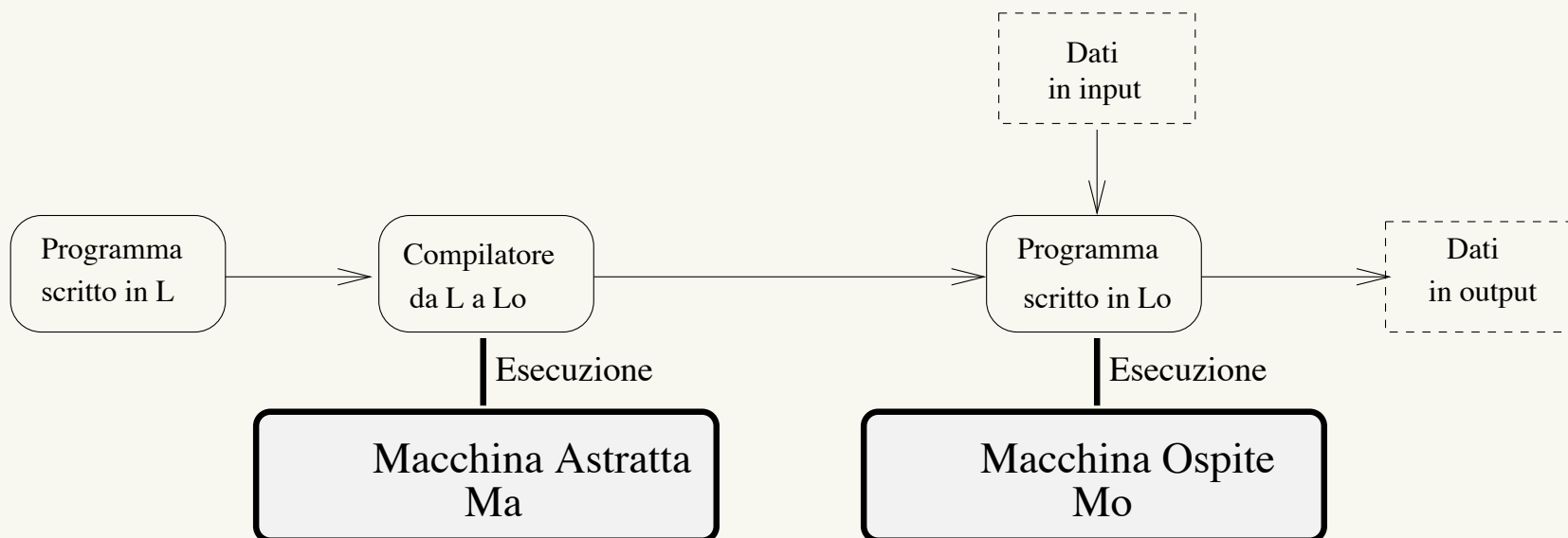
- Implementiamo in software  $\mathcal{M}_{\mathcal{L}}$  (esegue programmi scritti in  $\mathcal{L}$ )
- Il software esegue su una **Macchina Ospite**  $\mathcal{M}_{\mathcal{O}_{\mathcal{L}_0}}$  (con linguaggio macchina  $\mathcal{L}_0$ )
- Due tipologie di soluzione: *interpretativa* e *compilativa*
  - Interprete: programma scritto in  $\mathcal{L}_0$  che capisce ed esegue  $\mathcal{L}$ 
    - Implementa il ciclo  
fetch/decode/load/exec/save
  - Compilatore: programma che capisce traduce altri programmi da  $\mathcal{L}$  a  $\mathcal{L}_0$

# Implementazione Puramente Interpretativa



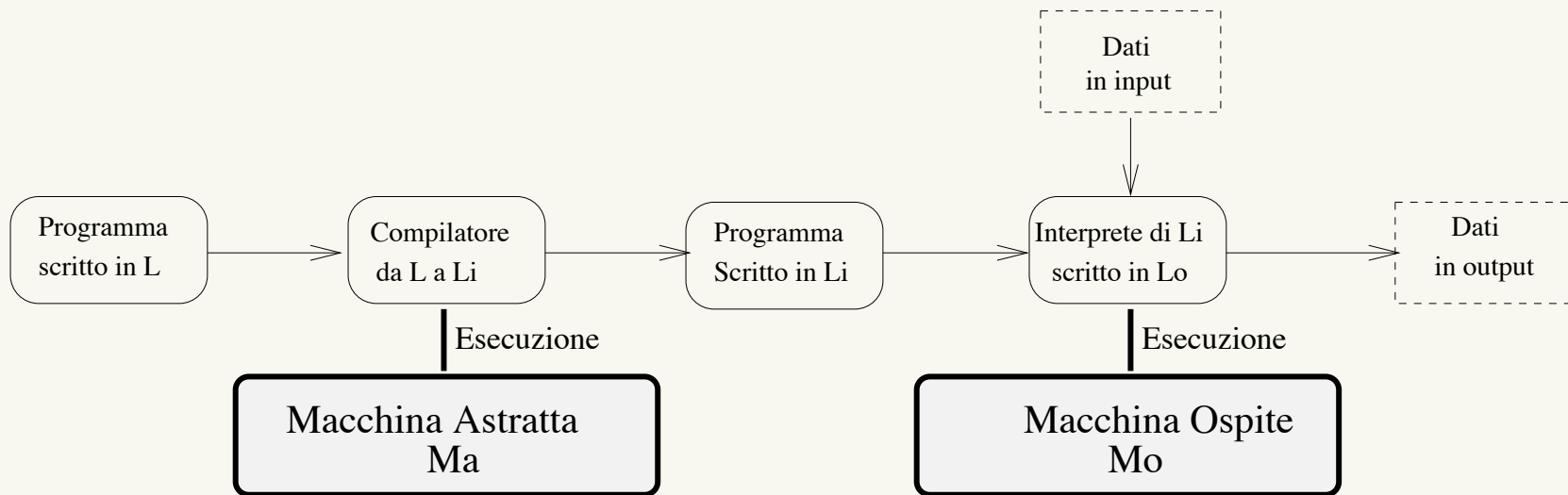
- Interprete: programma scritto in  $\mathcal{L}_0$  (esegue su  $\mathcal{M}_{O_{\mathcal{L}_0}}$ ) che capisce ed esegue programmi scritti in  $\mathcal{L}$
- Traduzione da  $\mathcal{L}_0$  a  $\mathcal{L}$  “istruzione per istruzione”

# Implementazione Puramente Compilativa



- Traduzione dell'intero programma da  $\mathcal{L}$  a  $\mathcal{L}_o$  *prima di eseguirlo*
- Traduzione è effettuata da un apposito programma, il **Compilatore**
  - Compilatore: non necessariamente scritto in  $\mathcal{L}_o$
  - Può eseguire su una macchina astratta  $\mathcal{M}_a$  diversa da  $\mathcal{M}_{o\mathcal{L}_o}$
  - **Cross-compilazione!**

# Implementazione “Ibrida”



- Implementazione non puramente compilativa ne' puramente interpretativa
- Compilatore traduce in un *linguaggio intermedio*  $\mathcal{L}_i$
- Interprete (o altro) esegue su  $\mathcal{M}_{O_{\mathcal{L}_o}}$  programma scritto in  $\mathcal{L}_i$
- Java: compilatore  $\rightarrow$  bytecode, poi JVM
- C: compilatore **generalmente** genera codice che ha bisogno di SO e runtime per eseguire



# Implementazioni Ibride - 2

- Dipendentemente dalle differenze e similitudini fra  $\mathcal{L}_i$  ed  $\mathcal{L}_o$  si parla di implementazioni **prevalentemente** compilative o interpretative
  - Java:  $\mathcal{L}_i == \text{bytecode} \rightarrow$  molto diverso da linguaggio macchina
    - Implementazione prevalentemente interpretativa
    - Ma alcune JVM sono basate su compilatori!!!
  - C:  $\mathcal{L}_i \approx$  linguaggio macchina
    - Ma non proprio identico: contiene chiamate a system call, funzioni di libreria, ...
    - Eccezione: compilatori “freestanding” (usati per compilare kernel)

# Compilatori vs Interpreti

- Meglio compilare o interpretare? Dipende...
- Implementazione puramente interpretativa:
  - Implementazione di  $\mathcal{M}_{\mathcal{L}}$  poco efficiente...
  - ... Ma più flessibile e portabile!
  - Debugging semplificato (interpretazione a run-time)
- Implementazione puramente compilativa:
  - Implementazione di  $\mathcal{M}_{\mathcal{L}}$  più efficiente...
  - ... Ma anche più complessa (“distanza” fra  $\mathcal{L}$  e  $\mathcal{L}_i/\mathcal{L}_o$ !)
  - Debugging generalmente più complesso
- In medio stat virtus...

# In Pratica...

- Come detto, è talvolta difficile stabilire se un linguaggio è *prevalentemente* compilato o interpretato...
- Comunque, proviamo
  - Linguaggi tipicamente implementati in modo compilativo:
    - C, C++, FORTRAN, Pascal, ADA
  - Linguaggi tipicamente implementati in modo interpretativo:
    - LISP, ML, Perl, Postscript, Pascal, Prolog, Smalltalk, Java
- Siamo così sicuri riguardo a Java? E LISP?

# Implementazione in Firmware

- Abbiamo visto come implementare macchine astratte in sw...
- ... Mentre una CPU è un'implementazione hw
- Esistono anche interpretazioni “intermedie”
  - Esempio: CPU microprogrammate
  - La macchina astratta che esegue il linguaggio macchina della CPU non è implementata in hw...
  - ... Ma è implementata da un microinterprete
  - Ciclo fetch/decode/load/execute/save implementato usando *microistruzioni* invisibili a normali utenti
  - Strutture dati e algoritmi MA realizzati da microprogrammi
- Alta velocità, flessibilità maggiore che hw puro

# Definizioni

- Definizione più formale di interpreti e compilatori...
- Prima di tutto, programma come funzione
  - Programma scritto in linguaggio  $\mathcal{L}$ :

$$P^{\mathcal{L}} : \mathcal{D} \rightarrow \mathcal{D}$$

- $\mathcal{D}$ : insieme dei dati di input ed output del programma
  - $P^{\mathcal{L}}(i) = o$ 
    - $i \in \mathcal{D}$ : input;  $o \in \mathcal{D}$ : output
- Interpreti e compilatori sono anche loro programmi...
  - Funzioni “un po’ particolari”, che ricevono in input altri programmi

# Definizione di Interprete

- Interprete per linguaggio  $\mathcal{L}$  scritto in linguaggio  $\mathcal{L}_o$ 
  - Implementa macchina astratta  $\mathcal{M}_{\mathcal{L}}$  su macchina astratta  $\mathcal{M}_{o\mathcal{L}_o}$
- Funzione  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_o} : (\mathcal{P}r^{\mathcal{L}} \times \mathcal{D}) \rightarrow \mathcal{D}$ 
  - $\mathcal{P}r^{\mathcal{L}}$ : insieme dei programmi  $P^{\mathcal{L}}$  scritti in linguaggio  $\mathcal{L}$
- $\forall i \in \mathcal{D}, \mathcal{I}_{\mathcal{L}}^{\mathcal{L}_o}(P^{\mathcal{L}}, i) = P^{\mathcal{L}}(i)$ 
  - Per qualsiasi input  $i$ , l'interprete applicato a  $P^{\mathcal{L}}$  ed  $i$  ritorna lo stesso risultato ritornato da  $P^{\mathcal{L}}$  applicato ad  $i$
  - $P^{\mathcal{L}}(i)$  è calcolata dalla macchina astratta  $\mathcal{M}_{\mathcal{L}}$ , mentre  $\mathcal{I}_{\mathcal{L}}^{\mathcal{L}_o}(P^{\mathcal{L}}, i)$  è calcolata dalla macchina astratta ospite  $\mathcal{M}_{o\mathcal{L}_o}$

# Definizione di Compilatore

- Compilatore da linguaggio  $\mathcal{L}$  a linguaggio  $\mathcal{L}_o$  (scritto in linguaggio  $\mathcal{L}_a$ )
  - Implementa  $\mathcal{M}_{\mathcal{L}}$  su  $\mathcal{M}_{o_{\mathcal{L}_o}}$ ...
  - ...trasformando programmi  $P^{\mathcal{L}} \in \mathcal{P}r^{\mathcal{L}}$  in programmi  $P^{\mathcal{L}_o} \in \mathcal{P}r^{\mathcal{L}_o}$
- Funzione  $\mathcal{C}_{\mathcal{L}, \mathcal{L}_o}^{\mathcal{L}_a} : \mathcal{P}r^{\mathcal{L}} \rightarrow \mathcal{P}r^{\mathcal{L}_o}$
- $\forall i \in \mathcal{D}, P_C^{\mathcal{L}_o} = \mathcal{C}_{\mathcal{L}, \mathcal{L}_o}^{\mathcal{L}_a}(P^{\mathcal{L}}) \Rightarrow P_C^{\mathcal{L}_o}(i) = P^{\mathcal{L}}(i)$ 
  - Per qualsiasi input  $i$ , il programma compilato  $P_C^{\mathcal{L}_o} = \mathcal{C}_{\mathcal{L}, \mathcal{L}_o}^{\mathcal{L}_a}(P^{\mathcal{L}})$  applicato ad  $i$  ritorna lo stesso risultato ritornato da  $P^{\mathcal{L}}$  applicato ad  $i$
  - $P^{\mathcal{L}}(i)$ : calcolata dalla macchina astratta  $\mathcal{M}_{\mathcal{L}}$ ;  
 $P_C^{\mathcal{L}_o}$ : calcolata dalla macchina ospite  $\mathcal{M}_{o_{\mathcal{L}_o}}$
  - Compilatore  $\mathcal{C}_{\mathcal{L}, \mathcal{L}_o}^{\mathcal{L}_a}(P^{\mathcal{L}})$ : calcolata da  $\mathcal{M}_a$

# Capitolo 6

I nomi e l'ambiente





# Nomi

- **nome**

- sequenza di caratteri usata per *denotare* qualche cos'altro

- `const pi = 3.14;`

- `int x;`

- `void f(){...};`

oggetto denotato:

la costante 3.14

una variabile

la definizione di f

nomi

- Nei linguaggi i nomi sono spesso **identificatori** (token alfa-numerici)
- L'**uso** di un nome serve ad indicare l'oggetto denotato
  - oggetti simbolici più facili da ricordare
  - astrazione (sia sui dati che sul controllo)

# Oggetti denotabili

- Oggetto *denotabile*
  - quando può essergli associato un nome
- Nomi definiti dall'utente
  - variabili, parametri formali, procedure (in senso lato), tipi definiti dall'utente, etichette, moduli, costanti definite dall'utente, eccezioni
- Nomi definiti dal linguaggio
  - tipi primitivi, operazioni primitive, costanti predefinite.
- Terminologia:
  - *Legame* (binding), o *associazione*, tra nome e oggetto

# Ambiente

## Ambiente:

insieme delle associazioni fra nomi e oggetti denotabili esistenti a run-time in uno specifico punto del programma ed in uno specifico momento dell'esecuzione

## Dichiarazione:

meccanismo (implicito o esplicito) col quale si crea un'associazione in ambiente

```
int x;  
int f () {  
    return 0;  
}  
type T = int;
```

# Ambiente, 2

Lo stesso nome può denotare oggetti distinti  
in punti diversi del programma

## Aliasing

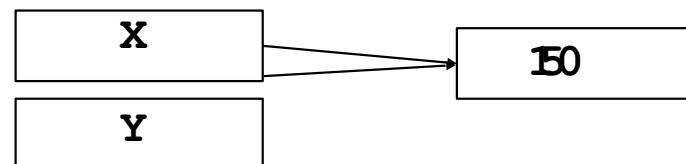
nomi diversi denotano lo stesso oggetto

passaggio per riferimento

puntatori

ecc.

```
int *X, *Y;           // X,Y puntatori a interi
X = (int *) malloc (sizeof (int));
                        // allocata la memoria puntata
*X = 5;                // * dereferenzia
Y=X;                   // Y punta alla stesso oggetto di X
*Y=10;
write(*X);
```



# Blocchi

- Nei linguaggi moderni l'ambiente è *strutturato*
- **Blocco:**
  - regione testuale del programma, identificata da un segnale di inizio ed uno di fine, che può contenere dichiarazioni *locali* a quella regione
    - `begin...end`                      Algol, Pascal
    - `{...}`                                C, Java
    - `(...)`                                Lisp
    - `let...in...end`                    ML
  - anonimo (o in-line)
  - associato ad una procedura

# Suddividiamo l'ambiente

- L'ambiente (in uno specifico blocco) può essere suddiviso in
  - **ambiente locale**: associazioni create all'ingresso nel blocco
    - variabili locali
    - parametri formali
  - **ambiente non locale** : associazioni ereditate da altri blocchi
  - **ambiente globale**: quella parte di ambiente non locale relativo alle associazioni comuni a tutti i blocchi
    - dichiarazioni esplicite di variabili globali
    - dichiarazioni del blocco più esterno
    - associazioni esportate da moduli ecc.

# Operazioni sull'ambiente

- **Creazione** associazione nome-oggetto denotato (naming)
  - dichiarazione locale in blocco
- **Riferimento** oggetto denotato mediante il suo nome (referencing)
  - uso di un nome
- **Disattivazione** associazione nome-oggetto denotato
  - entrata in un blocco con dichiarazione che maschera

**Riattivazione** associazione nome-oggetto denotato

  - uscita da blocco con dichiarazione che maschera
- **Distruzione** associazione nome-oggetto denotato (unnaming)
  - uscita da blocco con dichiarazione locale

# Alcuni eventi fondamentali

1. Creazione di un oggetto
2. Creazione di un legame per l'oggetto
3. Riferimento all'oggetto, tramite il legame
4. Disattivazione di un legame
5. Riattivazione di un legame
6. Distruzione di un legame
7. Distruzione di un oggetto

Il tempo tra 1 e 7 è la **vita** (o il tempo di vita: *lifetime*)  
**dell'oggetto**

Il tempo tra 2 e 6 è la **vita dell'associazione**



# Regole di scope

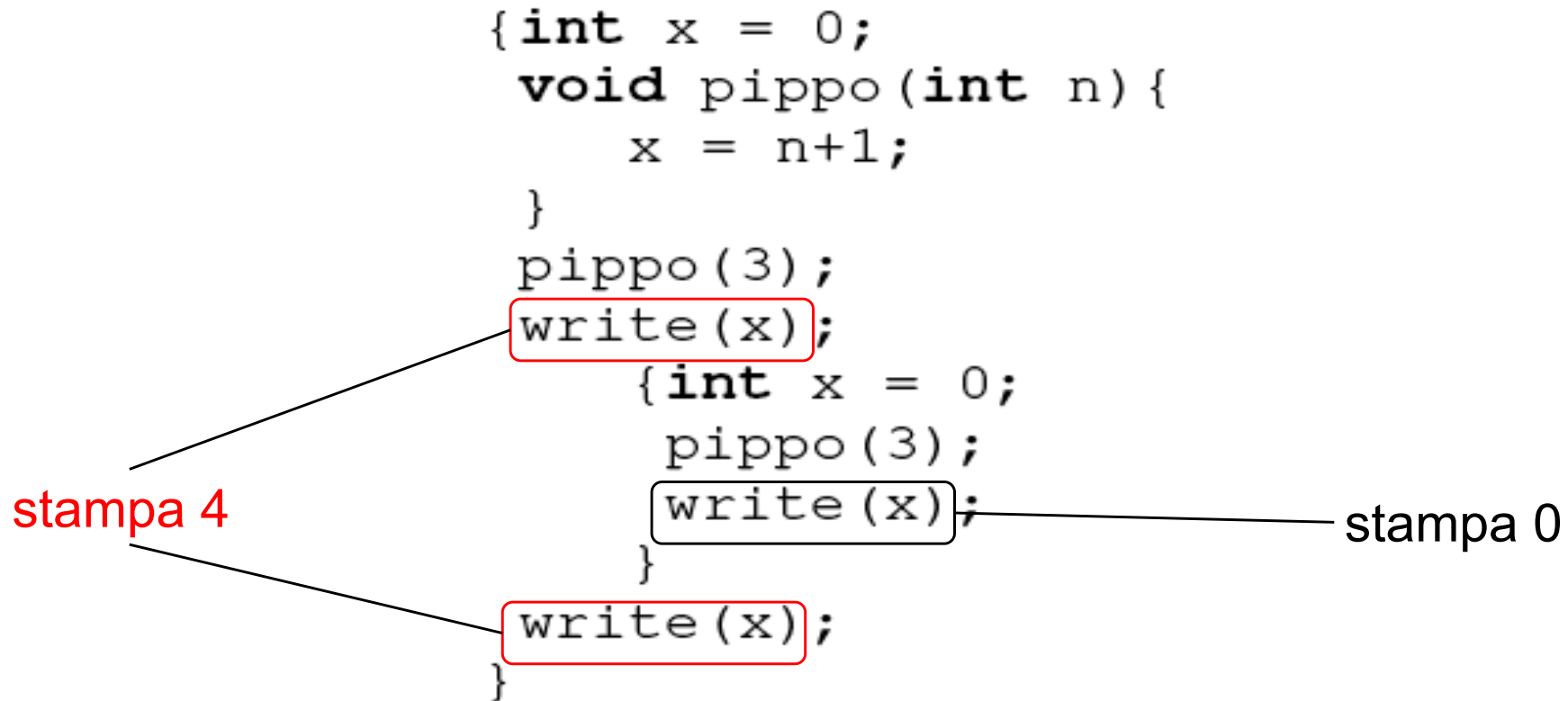
- Come deve essere interpretata la regola di visibilità?

*Una dichiarazione locale ad un blocco è visibile in quel blocco e in tutti i blocchi in esso annidati, a meno che non intervenga in tali blocchi una nuova dichiarazione dello stesso nome (che nasconde, o maschera, la precedente)*

- in presenza di procedure  
cioè di blocchi che sono eseguiti in posizioni diverse dalla loro definizione
- in presenza ambiente non locale (e non globale)

# Scope statico

- Un nome non locale è risolto nel blocco che *testualmente* lo racchiude



# Scope dinamico

- Un nome non locale è risolto nel blocco attivato *più di recente* e non ancora disattivato

```
    {int x = 0;
      void pippo(int n) {
        x = n+1;
      }
      pippo(3);
      write(x);
      {int x = 0;
        pippo(3);
        write(x);
      }
      write(x);
    }
```

stampa 4

# Scope statico vs dinamico

- **Scope statico** (*scoping statico, statically scoped, lexical scope*).
  - informazione completa dal testo del programma
  - le associazioni sono note a tempo di compilazione
  - principi di indipendenza
  - più complesso da implementare ma più efficiente
  - Algol, Pascal, C, Java, ...
- **Scope dinamico** (*scoping dinamico, dynamically scoped*).
  - informazione derivata dall'esecuzione
  - spesso causa di programmi meno ``leggibili''
  - più semplice da implementare, ma meno efficiente
  - Lisp (alcune versioni), Perl
- Differiscono solo in presenza congiunta di
  - ambiente non locale e non globale
  - procedure

# Determinare l'ambiente

- **L'ambiente** è dunque determinato da
  - regola di **scope** (statico o dinamico)
  - regole specifiche, p.e.
    - quando è visibile una dichiarazione nel blocco in cui compare?

discuteremo più avanti

- regole per il **passaggio dei parametri**
- regole di **binding** (shallow o deep)
  - intervengono quando una procedura P è passata come parametro ad un'altra procedura mediante il formale X

# Capitolo 7

La gestione della memoria



# Tipi di allocazione della memoria

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

# Allocazione statica

- Un oggetto ha un indirizzo assoluto che è mantenuto per tutta l'esecuzione del programma
- Solitamente sono allocati staticamente:
  - variabili globali
  - variabili locali sottoprogrammi (senza ricorsione)
  - costanti determinabili a tempo di compilazione
  - tabelle usate dal supporto a run-time (per type checking, garbage collection, ecc.)
- Spesso usate zone protette di memoria



# L'allocazione statica non permette ricorsione

FORTRAN: Programma sintatticamente **illegale**: ricorsione non ammessa

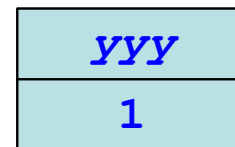
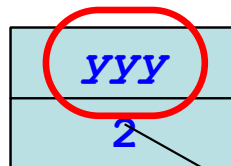
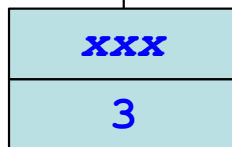
```
SUBROUTINE ERROR(N)
    IF (N.LE.1) RETURN
yyy    CALL ERROR(N-1)
    PRINT N
END
```

**Supponiamo legale:  
eseguiamo nel modello  
di memoria statica**

xxx CALL ERROR(3)

Unica area statica

IndRit  
N



OUTPUT

L'indirizzo di ritorno  
originale è perduto

1 1 1 1 1 1 ...

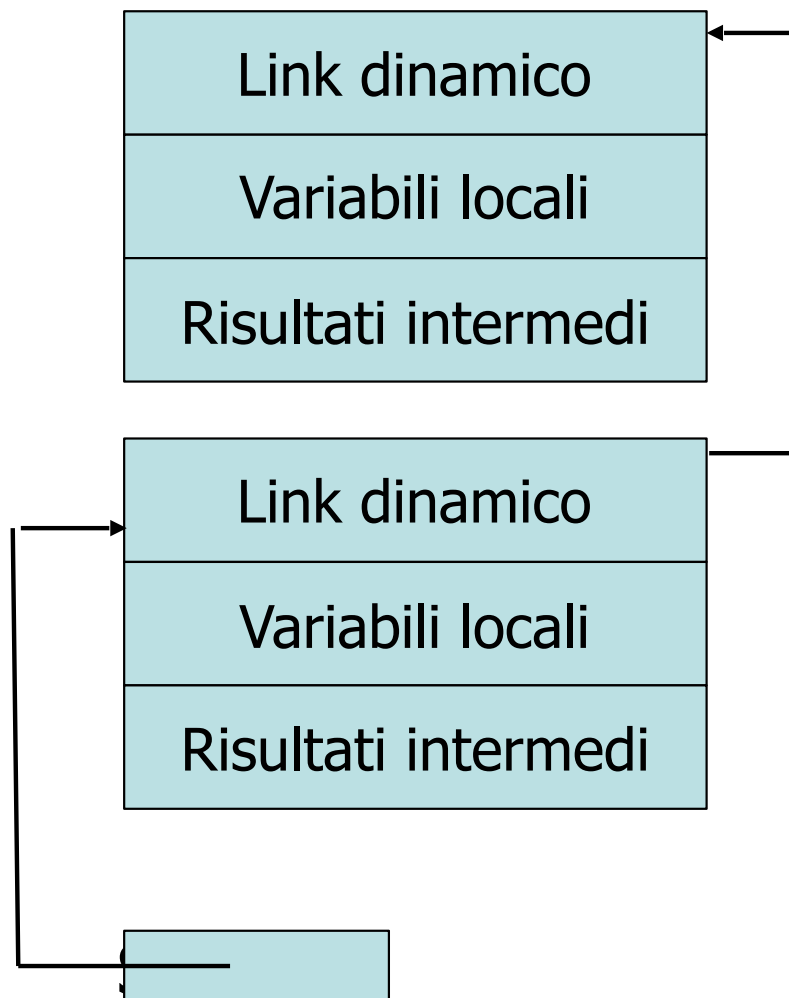
# Allocazione dinamica: pila

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

# Allocazione dinamica con pila

- La gestione della pila è compiuta mediante:
  - sequenza di chiamata
  - prologo
  - epilogo
  - sequenza di ritorno
- Indirizzo di un RdA non è noto a compile-time.
- Il Puntatore RdA (o SP) punta al RdA del blocco attivo
- Le info contenute in un RdA sono accessibili per offset rispetto allo SP:
  - $\text{indirizzo-info} = \text{contenuto(SP)} + \text{offset}$
  - offset determinabile staticamente
  - Somma eseguita con unica istruzione macchina **load** o **store**

# Record di attivazione per blocchi in-line



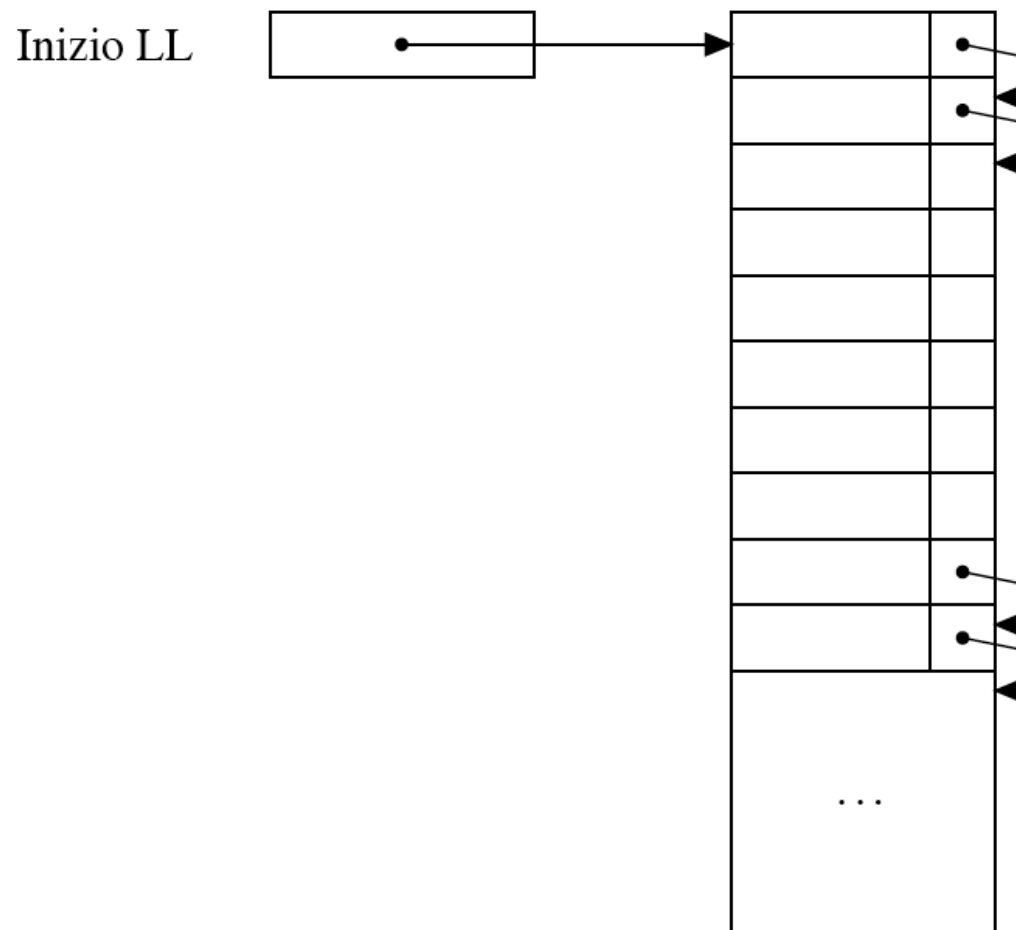
- Link dinamico (o **control link**)
  - puntatore al precedente record sullo stack
- Ingresso nel blocco: **Push**
  - link dinamico del nuovo Rda := SP
  - SP aggiornato a nuovo Rda
- Uscita dal blocco: **Pop**
  - Elimina Rda puntato da SP
  - SP := link dinamico del Rda tolto dallo stack

# Allocazione dinamica con heap

- **Heap:** regione di memoria i cui (sotto) blocchi possono essere allocati e deallocati in momenti arbitrari
- Necessario quando il linguaggio permette
  - allocazione esplicita di memoria a run-time
  - oggetti di dimensioni variabili
  - oggetti con vita non LIFO
- La gestione dello heap non è banale
  - gestione efficiente dello spazio: frammentazione
  - velocità di accesso

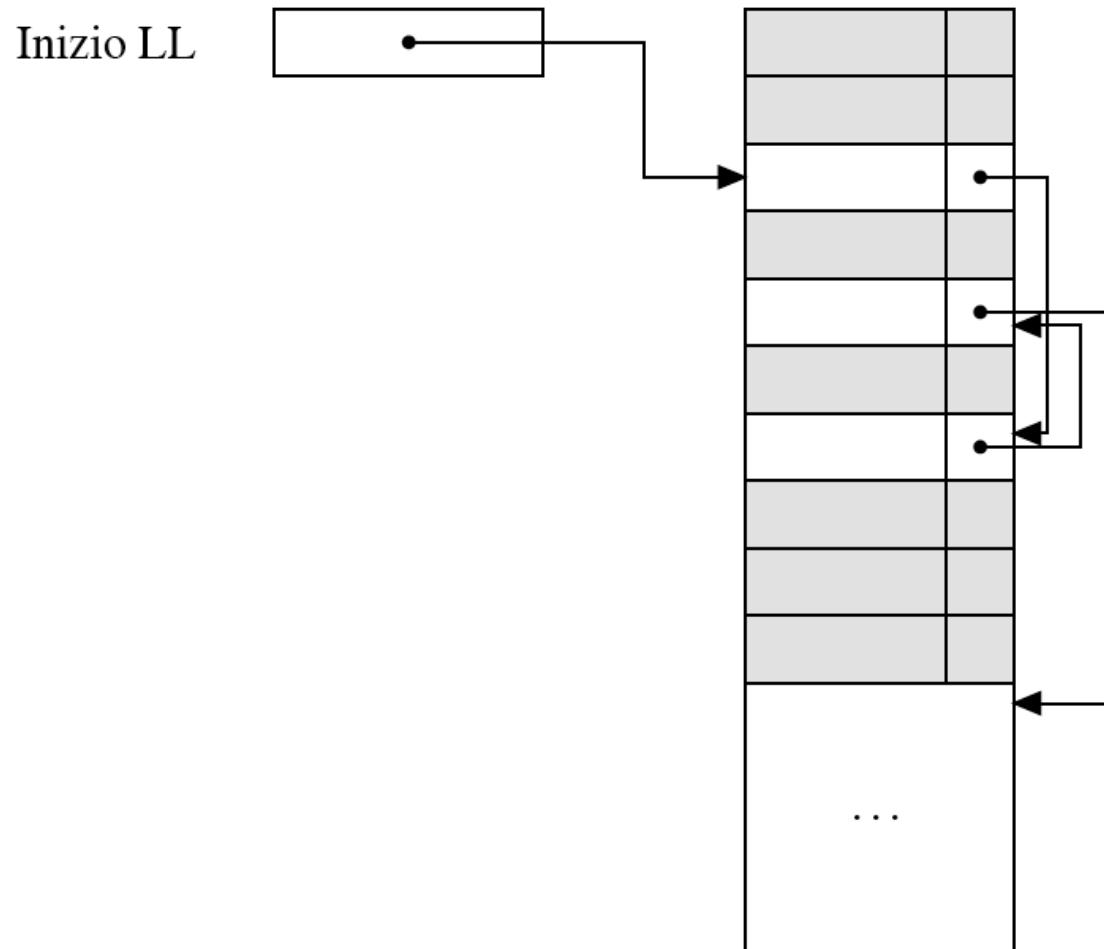
# Heap: blocchi di dimensione fissa

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



# Heap: blocchi di dimensione fissa

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



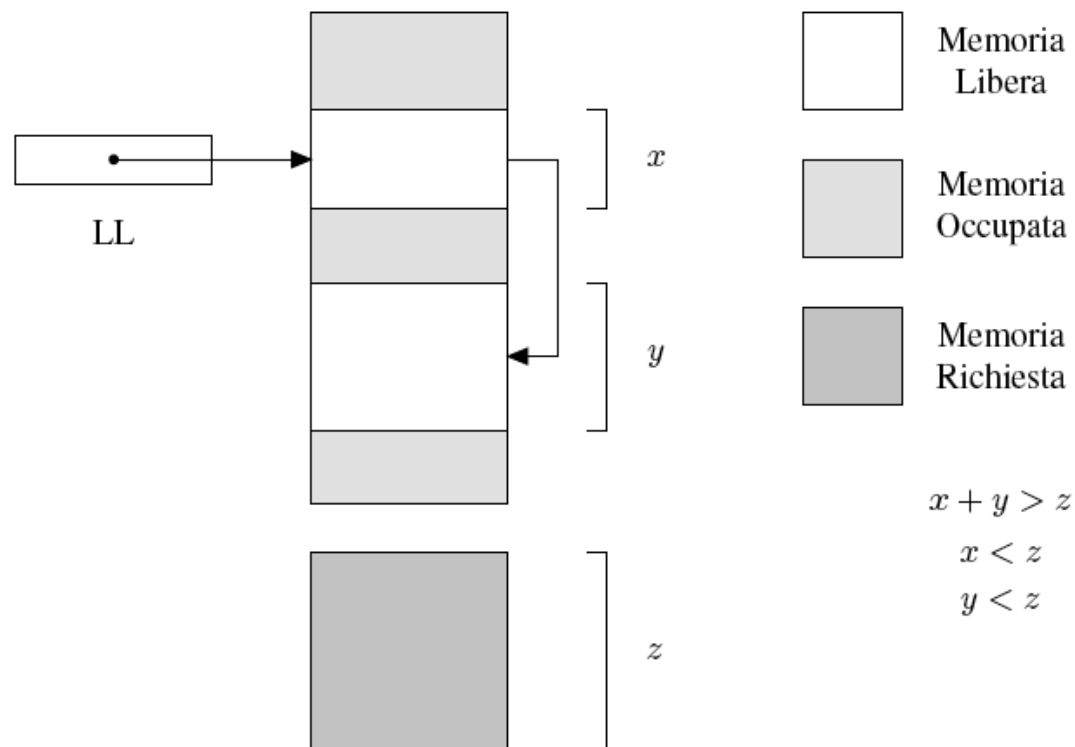
# Heap: blocchi di dimensione variabile

- Inizialmente unico blocco nello heap
- Allocazione: determinazione di un blocco libero della dimensione opportuna
- Deallocazione: restituzione alla lista libera
- Problemi:
  - le operazioni devono essere efficienti
  - evitare lo spreco di memoria
    - frammentazione interna
    - frammentazione esterna



# Frammentazione

- Frammentazione **interna**: lo spazio richiesto è  $X$ ,
  - viene allocato un blocco di dimensione  $Y > X$ ,
  - lo spazio  $Y-X$  è sprecato
- Frammentazione **esterna**: ci sarebbe lo spazio necessario ma è inusabile perché suddiviso in “pezzi” troppo piccoli



# Gestione della lista libera: unica lista

- Inizialmente un solo blocco, della dimensione dello heap
- Ad ogni richiesta di allocazione: cerca blocco di dimensione opportuna
  - **first fit**: primo blocco grande abbastanza
  - **best fit**: quello di dimensione più piccola, grande abbastanza
- Se il blocco scelto è molto più grande di quello che serve viene diviso in due e la parte inutilizzata è aggiunta alla LL
- Quando un blocco è de-allocato, viene restituito alla LL (se un blocco adiacente è libero i due blocchi sono ``fusi'' in un unico blocco).

# Gestione heap

- **First fit** o **Best Fit** ? Solita situazione conflittuale:
  - First fit: più veloce, occupazione memoria peggiore
  - Best fit: più lento, occupazione memoria migliore
- Con unica LL costo allocazione lineare nel numero di blocchi liberi. Per migliorare liste libere multiple: La ripartizione dei blocchi fra le varie liste può essere
  - **statica**
  - **dinamica:**
    - Buddy system: k liste; la lista k ha blocchi di dimensione  $2^k$ 
      - » se richiesta allocazione per blocco di  $2^k$  e tale dimensione non è disponibile, blocco di  $2^{k+1}$  diviso in 2
      - » se un blocco di  $2^k$  e' de-allocato è riunito alla sua altra metà (*buddy*), se disponibile
      - »
    - Fibonacci simile, ma si usano numeri di Fibonacci

# Implementazione delle regole di scope

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

# Tentiamo di ridurre i costi: il *display*

- Si può ridurre il costo derivante dalla scansione della CS ad una costante usando il *display*:
- La catena statica viene rappresentata mediante un array (detto *display*):
  - $i$ -esimo elemento dell'array = puntatore all'RdA del sottoprogramma di livello di annidamento  $i$ , attivo per ultimo
- Se il sottoprogramma corrente è annidato a livello  $i$ , un oggetto che è in uno scope esterno di  $h$  livelli può essere trovato guardando il punt a RdA nel *display* alla posizione  $j = i - h$

# Come si determina il display

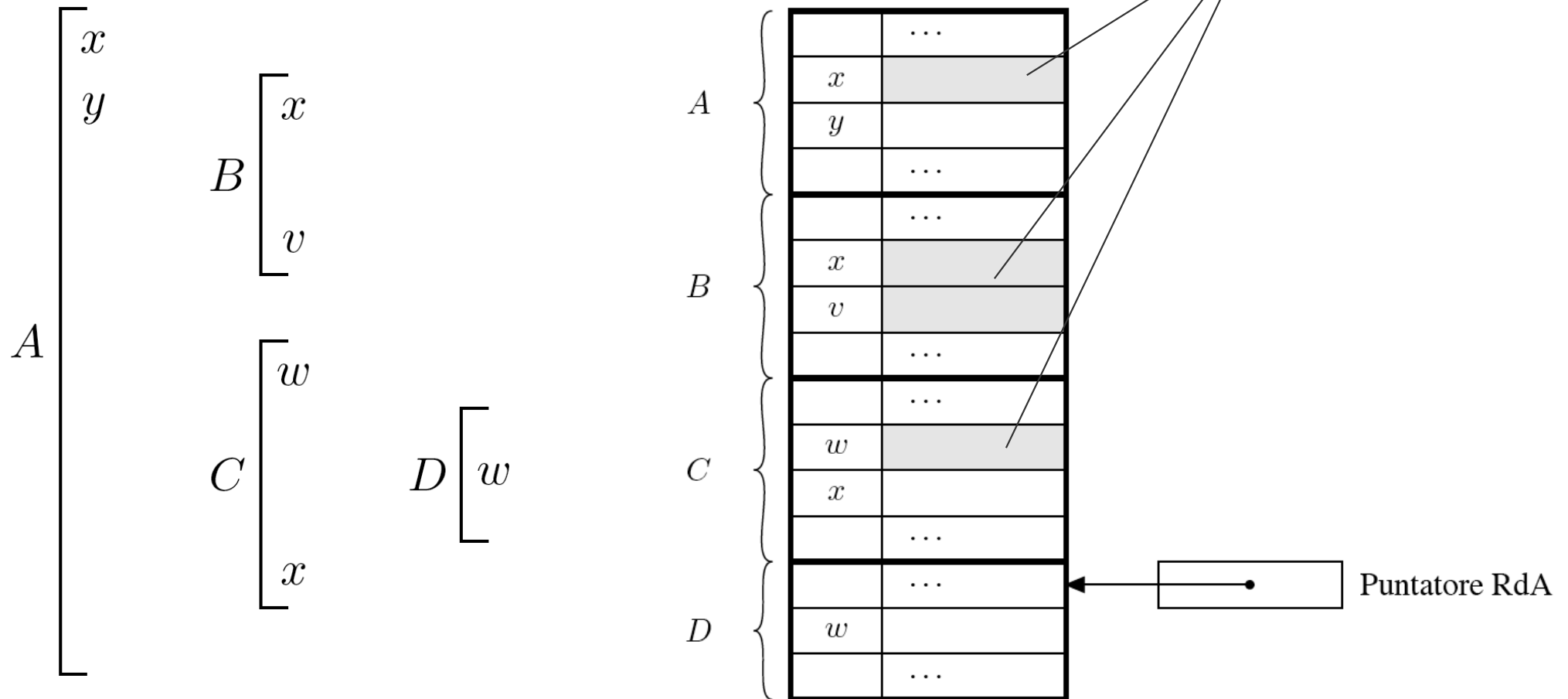
- È il **chiamato** a maneggiare il display.
  - Quando Ch chiama P a livello di annidamento **j**, P salva il valore di **Display[j]** nel proprio RdA e vi mette una copia del proprio (nuovo) punt a RdA.
- Funziona: soliti due casi
  - P dichiarata immediatamente in Ch (**k**=0); Ch e P condividono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

# Scope dinamico

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

# Implementazione ovvia

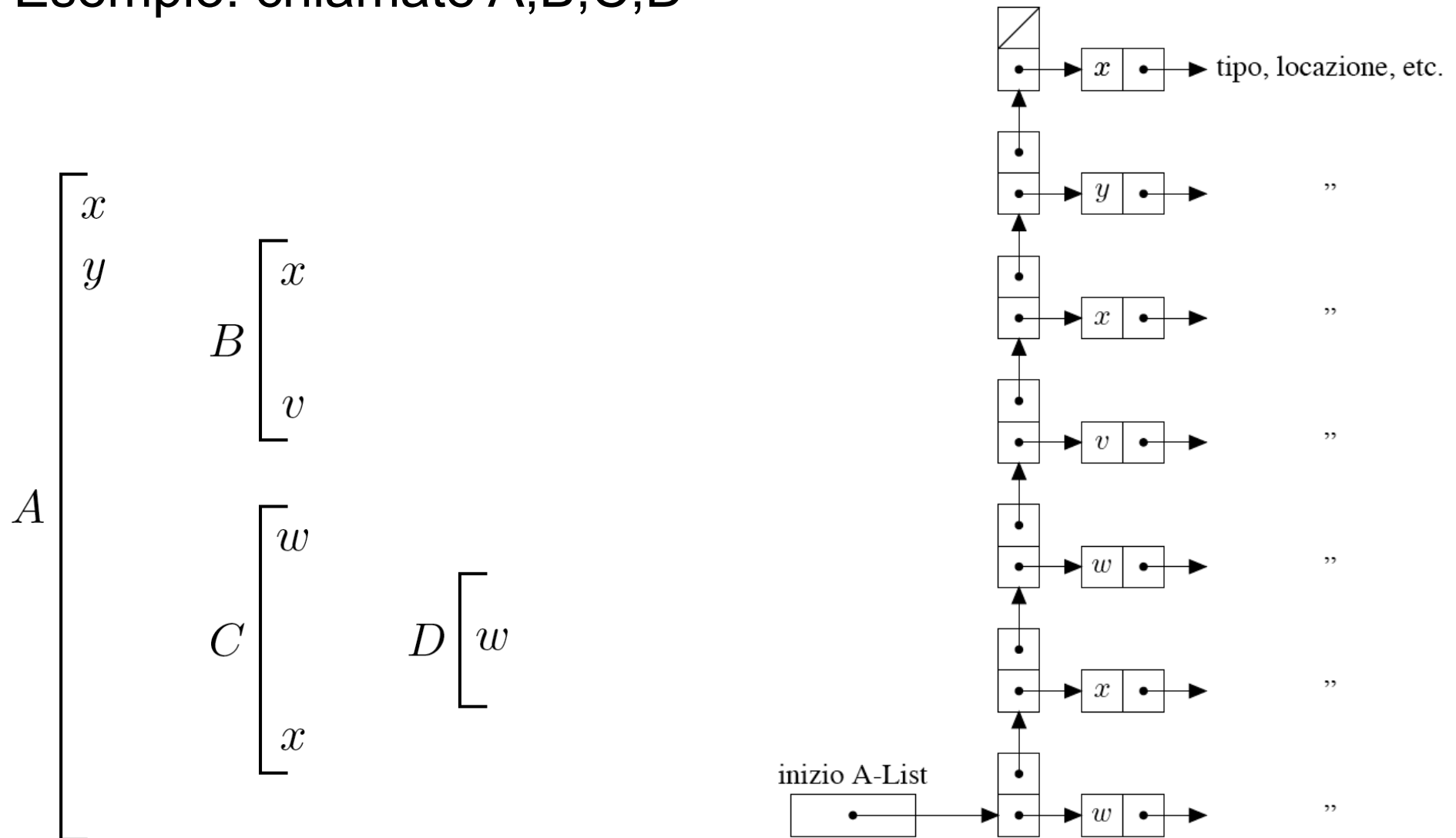
- Memorizzare i nomi negli RdA
- Ricerca per nome risalendo la pila
- Esempio: chiamate A,B,C,D





# Variante: A-list

- Le associazioni sono memorizzate in una struttura apposita, manipolata come una pila
- Esempio: chiamate A,B,C,D



# Costi delle A-list

- Molto semplice da implementare
- Occupazione memoria:
  - nomi presenti esplicitamente
- Costo di gestione
  - ingresso/uscita da blocco
    - inserzione/rimozione di blocchi sulla pila
- Tempo di accesso
  - sempre lineare nella profondità della A-list
- Possiamo ridurre il tempo d'accesso medio, aumentando il tempo di ingresso/uscita da blocco...

# Tabella centrale dei riferimenti, CRT

- Evita le lunghe scansioni delle A-list
- Una tabella mantiene tutti i nomi distinti del programma
  - se nomi noti staticamente accesso in tempo costante
  - altrimenti, accesso hash
- Ad ogni nome è associata la lista delle associazioni di quel nome
  - la più recente è la prima
  - le altre (disattivate) seguono
- Tempo di accesso costante

# Costi della CRT

- Gestione più complessa di A-list
- Meno occupazione di memoria:
  - se nomi noti staticamente, i nomi non sono necessari
  - in ogni caso, ogni nome memorizzato una sola volta
- Costo di gestione
  - ingresso/uscita da blocco
    - manipolazione di tutte le liste dei nomi presenti nel blocco
- Tempo di accesso
  - costante (due accessi indiretti)
- Possiamo ridurre il tempo d'accesso medio, aumentando il tempo di ingresso/uscita da blocco...

# Capitolo 8

Strutturare il controllo



# Controllo del flusso

- Espressioni
  - Notazioni
  - Valutazione
  - Problemi
- Comandi
  - Assegnamento
  - Sequenziale
  - Condizionale
- Comandi iterativi
- Ricorsione

# Variabile

- In matematica un'incognita che può assumere i valori di un insieme predefinito
  - non è modificabile !
- Nei linguaggi imperativi: (Pascal, C, Ada, ...):  
variabile modificabile
  - un contenitore di valori che ha un nome

X      2

- il valore nel contenitore può essere modificato

# Assegnamento

- Comando che modifica il valore di una variabile (modificabile)
- Normalmente la valutazione di un assegnamento non restituisce un valore ma produce un effetto collaterale (in C restituisce valore)

$X := 2$

$X = X + 1$

Diverso ruolo di  $X$  e  $X$ :

- $X$  è un l-value (denota una locazione)
- $X$  è un r-value (può essere contenuto in una locazione)

- In generale

exp1 Opass exp2



# Modelli di variabile diversi

- Linguaggi funzionali (Lisp, ML, Haskell, Smalltalk): una variabile denota un valore e non è modificabile
- Linguaggi logici: una variabile è modificabile solo entro certi limiti (istanziamento)
- Clu: modello a oggetti, chiamato anche modello a riferimento
  - Una variabile è un riferimento ad un valore, che ha un nome
  - Analogo al puntatore ma senza le possibilità di manipolazione esplicita delle locazioni
- Java:
  - variabile modificabile per i tipi primitivi (interi, booleani ecc.)
  - modello a riferimento per i tipi classe

# Operatori di assegnamento

- $X := X+1$ 
  - doppio accesso alla locazione di  $a$  (a meno di ottimizzazione del compilatore)
  - poco chiaro; in alcuni casi può causare errori

$A[\text{index}(i)] := A[\text{index}(i)] + 1$   
( $\text{index}(i)$  potrebbe causare un side effect)


No

$j := \text{index}(i)$   
 $A[j] := A[j] + 1;$

Meglio

- { Alcuni linguaggi usano opportuni operatori di assegnamento }

# Operatori di assegnamento

- $X := X + 1$    $X += 1$  (Pascal)  
 $X += 1$  (C)
- In C 10 diversi operatori di assegnamento, incremento/decremento prefissi e postfissi:  
++e   --e   e++   e--   .....
- L'incremento di un puntatore in C tiene conto della dimensione degli oggetti puntati

# Ambiente e memoria

- Nei linguaggi imperativi sono presenti tre importanti domini semantici:
  - Valori Denotabili (quelli a cui si può dare un nome)
  - Valori Memorizzabili (si possono memorizzare)
  - Valori Esprimibili (risultato della valutazione di una exp.)
- La semantica dei linguaggi imperativi usa
  - Ambiente: Nomi ----> Valori Denotabili
  - Memoria: Locazioni ---> Valori Memorizzabili
  - Permettono di rappresentare l'aliasing
- I linguaggi funzionali usano solo l'ambiente

# Iterazione

- **Iterazione** e **ricorsione** sono i due meccanismi che permettono di ottenere formalismi di calcolo Turing completi. Senza di essi avremmo automi a stati finiti
- Iterazione
  - **indeterminata**: cicli controllati logicamente  
(`while`, `repeat`, ...)
  - **determinata** cicli controllati numericamente  
(`do`, `for` ... ) con numero di ripetizioni del ciclo determinate al momento dell'inizio del ciclo

# Iterazione indeterminata

```
while condizione do comando
```

- Introdotto in Algol-W, rimasto in Pascal e in molti altri linguaggi, piu' semplice semanticamente del `for`
- In Pascal anche versione post-test:

```
repeat comando untill condizione
```

equivalente a

```
comando;
```

```
while not condizione do comando;
```

# Iterazione indeterminata

- Indeterminata perché il numero di iterazioni non è noto a priori
- L'iterazione indeterminata permette il potere espressivo delle MdT
- È di facile implementazione usando l'istruzione di salto condizionato della macchina fisica

# Iterazione determinata

```
FOR indice : = inizio TO fine BY passo DO  
    ...  
END
```

- non si possono modificare `indice`, `inizio`, `fine`, `passo` all'interno del loop
- è **determinato** (al momento dell'inizio dell'esecuzione del ciclo) il numero di ripetizioni del ciclo
- il potere espressivo è minore rispetto all'iterazione indeterminata: non si possono esprimere computazioni che non terminano
- in molti linguaggi (ad esempio C) il `for` non è un costrutto di iterazione determinata



# Iterazione controllata numericamente

```
FOR indice : = inizio TO fine  
                BY passo DO ... END
```

I vari linguaggi differiscono nei seguenti aspetti:

1. Possibilità di **modificare gli indici** primo, ultimo, passo nel loop (se sì, non si tratta di iterazione determinata)
2. **Numero di iterazioni** (dove avviene il controllo `indice < fine`)
3. Incremento **negativo**
4. **Valore** di `indice` al termine del ciclo
5. Possibilità di **salto** dall'esterno all'interno

# Ricorsione

- Modo alternativo all'iterazione per ottenere il potere espressivo delle MdT
- Intuizione: una funzione (procedura) è ricorsiva se definita in termini di se stessa.
- Esempio (abusato): il fattoriale

- Corrisponde alla definizione

fattoriale  
fattoriale

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

# Ricorsione e iterazione

- La ricorsione è possibile in ogni linguaggio che permetta
  - funzioni (o procedure) che possono chiamare se stesse
  - gestione dinamica della memoria (pila)
- Ogni programma ricorsivo (iterativo) può essere tradotto in uno equivalente iterativo (ricorsivo)
  - ricorsione più naturale con linguaggi funzionali e logici
  - iterazione più naturale con linguaggi imperativi
- In caso di implementazioni naif ricorsione meno efficiente di iterazione tuttavia
  - optimizing compiler può produrre codice efficiente
  - tail-recursion ...

# Ricorsione in coda (tail recursion)

- Una chiamata di  $g$  in  $f$  si dice “in coda” (o tail call) se  $f$  restituisce il valore restituito da  $g$  senza ulteriore computazione.

- $f$  è tail recursive se contiene solo chiamate in coda

```
function tail_rec (n: integer): integer  
begin ... ; x:= tail_rec(n-1) end
```

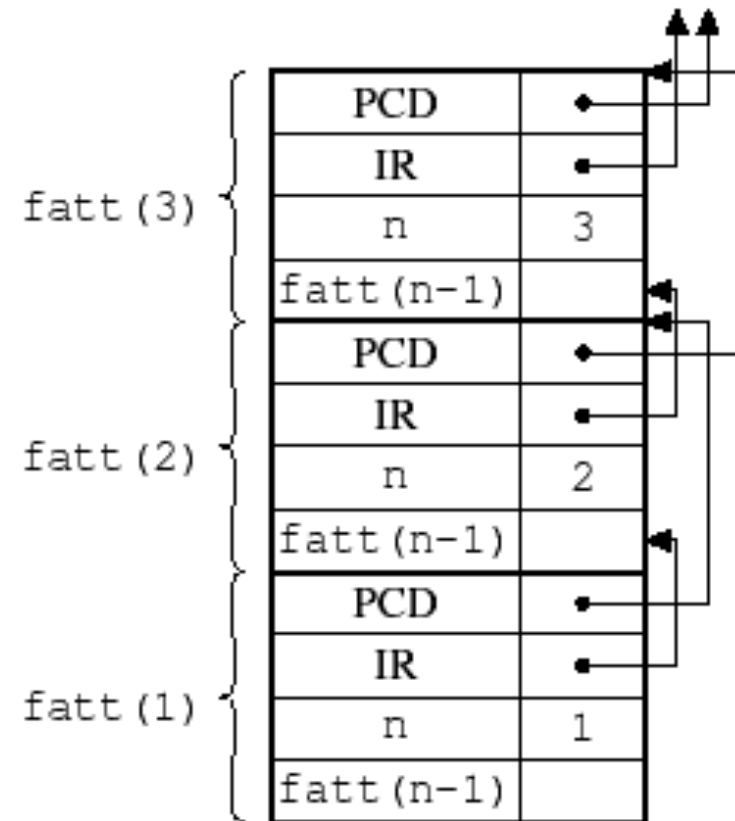
```
function non_tail_rec (n: integer): integer  
begin ... ; x:= non_tail_rec(n-1); y:= g(x) end
```

- Non serve allocazione dinamica della memoria con pila: basta un unico RdA !
- Più efficiente
- Possibile la generazione di codice tail-recursive usando continuation passing style

# Esempio: il caso del fattoriale

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

Situazione dei RdA  
Dopo la chiamata di f(3) e le  
successive chiamate ricorsive



# Una versione tail-recursive del fattoriale

- Cosa accade con la seguente funzione ?

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

- Abbiamo aggiunto un parametro per memorizzare ``il resto della computazione’’
- Basta un unico RdA
  - Dopo ogni chiamata il RdA può essere eliminato

# Una versione più efficiente per Fibonacci

- La versione tail-recursive

```
int fibrc (int n, int res1, int res2){  
    if (n == 0)  
        return res2;  
    else  
        if (n == 1)  
            return res2;  
        else  
            return fibrc(n-1, res2, res1+res2);  
}
```

- Complessità
  - in tempo lineare in n
  - in spazio costante (un soloRdA)

# Capitolo 9

Astrazione sul controllo:  
sottoprogrammi ed eccezioni



# Argomenti

- Astrazione sul controllo
- Modalità di passaggio dei parametri
- Funzioni di ordine superiore
  - funzioni come parametro
  - funzioni come risultato
- Gestori delle eccezioni

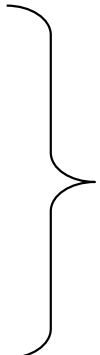
# Astrazione

- identificare proprietà importanti di cosa si vuole descrivere
- concentrarsi sulle questioni rilevanti e ignorare le altre
- cosa è rilevante dipende dallo scopo del progetto

# Astrazione sul controllo

- Sottoprogrammi, blocchi, parametri

```
double P (int x) {  
    double z;  
    /* CORPO DELLA FUNZIONE  
    return expr;  
}
```

- Specifica P
  - Scrivi P
  - Usa P
- 
- senza conoscere  
il contesto

# Parametri

- Terminologia:

- dichiarazione/definizione

```
int f (int n) {return n+1;}
```



Parametro formale

- uso/chiamata

```
x = f(y+3);
```



Parametro attuale

# Una funzione comunica col chiamante

- Valore restituito

```
int f() {return 1; }
```

- Parametri

- **main** → **proc**
- **main** ← **proc**
- **main** ↔ **proc**

- Ambiente non locale

# Modalità di passaggio dei parametri

- Due modi principali:
  - per valore:
    - il valore dell'attuale è assegnato al formale, che si comporta come una variabile locale
    - pragmatica:  $\text{main} \rightarrow \text{proc}$
    - attuale qualsiasi; modifiche al formale non passano all'attuale
  - per riferimento (per variabile)
    - è passato un riferimento (indirizzo) all'attuale; i riferimenti al formale sono riferimenti all'attuale (*aliasing*)
    - pragmatica:  $\text{main} \leftrightarrow \text{proc}$
    - attuale: variabile; modifiche al formale passano all'attuale

# Passaggio per valore

```
void foo (int x) { x = x+1; }  
...  
y = 1;  
foo(y+1);
```



qui y vale 1

- Il formale `x` è una var locale (sulla pila)
- Alla chiamata, l'attuale `y+1` è valutato ed il valore è assegnato al formale `x`
- Nessun legame tra `x` nel corpo di `foo` e `y` nel chiamante
- Al ritorno da `foo`, `x` viene distrutto (tolto dalla pila)
- Non è possibile trasmettere info da `foo` al chiamante mediante il parametro
- Costoso per dati grandi: copia
- Java, Scheme, Pascal (default), C e Java (unico modo);

# Passaggio per riferimento (per variabile)

```
void foo (reference int x){ x = x+1;}  
...  
y = 1;  
foo(y);
```



qui y vale 2

- Viene passato un riferimento (indirizzo; puntatore)
- Il formale `x` è un alias di `y`
- L'attuale deve essere un L-valore ("una variabile")
- Al ritorno da `foo`, viene distrutto il (solo) legame tra `x` e l'indirizzo di `y`
- Trasmissione bidirezionale tra chiamante e chiamato
- Efficiente nel passaggio, ma indirezionale nel corpo
- Pascal (var); in C simulato passando un puntatore...



# Passaggio per risultato

- Duale del passaggio per valore. Pragmatica: `main`  $\leftarrow$  `proc`

```
void foo (result int x) {x = 8;}
```

```
...
```

```
y = 1;  
foo(y);
```



qui y vale 8

- Il formale `x` è una var locale (sulla pila)
- Al ritorno da `foo`, il valore di `x` è assegnato all'attuale `y`
- Nessun legame tra `x` nel corpo di `foo` e `y` nel chiamante
- Al ritorno da `foo`, `x` viene distrutto (tolto dalla pila)
- Non è possibile trasmettere info dal chiamante a `foo` mediante il parametro
- Costoso per dati grandi: copia
- Ada: `out`

# Passaggio per valore/risultato

- Insieme valore+risultato. Pragmatica: main  $\leftrightarrow$  proc

```
void foo (value-result int x)
    { x = x+1; }
```

...

```
y = 8;
foo(y);
```



qui y vale 9

- Il formale `x` è a tutti gli effetti una var locale (sulla pila)
- Alla chiamata, il valore dell'attuale è assegnato al formale
- Al ritorno, il valore del formale è assegnato all'attuale
- Nessun legame tra `x` nel corpo di `foo` e `y` nel chiamante
- Al ritorno da `foo`, `x` viene distrutto (tolto dalla pila)
- Costoso per dati grandi: copia
- Ada: `in out` (ma solo per dati piccoli; per dati grandi passa riferimento)

# Valore e riferimento: morale

- Passaggio per valore:
  - semantica semplice: il corpo non ha necessità di conoscere come la procedura verrà chiamata (*trasparenza referenziale*)
  - implementazione abbastanza semplice
  - potenzialmente costoso il passaggio; efficiente il riferimento al parametro formale
  - necessità di altri meccanismi per comunicare main ← proc
- Passaggio per riferimento:
  - semantica complessa; *aliasing*
  - implementazione semplice
  - efficiente il passaggio; un po' più costoso il riferimento al parametro formale (un indiretto)

# Passaggio per nome

- Regola di copia:  
una **chiamata** alla procedura **P** è la stessa cosa che eseguire il **corpo** di **P** dopo aver **sostituito** i parametri attuali al posto dei parametri formali
- “Macro espansione”, realizzata in modo semanticamente corretto
- Apparentemente semplice...

# Capitolo 10

Strutturare i dati



# Tipo di dato

**Tipo:** Collezione di valori (omogenei ed effettivamente presentati) dotata di un insieme di operazioni per manipolare tali valori

Cosa è un tipo e cosa no dipende fortemente dal linguaggio di programmazione

# A cosa servono i tipi?

- Livello di progetto: Organizzano l'informazione
  - Tipi diversi per concetti diversi
  - Costituiscono “commenti” sull'uso inteso degli identificatori
- Livello di programma: Identificano e prevengono errori
  - I tipi (e non i commenti) sono controllabili automaticamente
  - Costituiscono un “controllo dimensionale”:
    - 3+“pippo” **deve** essere sbagliato
- Livello di implementazione: Permettono alcune ottimizzazioni
  - Bool richiede meno bit di real
  - Precalcolo degli offset di accesso a record/struct

# Array

- Collezioni di dati omogenei:
  - funzione da un tipo indice al **tipo degli elementi**
  - indice: in genere discreto
  - elemento: “qualsiasi tipo” (raramente un tipo funzionale)
- Dichiarazioni
  - C: `int vet[30];`                      tipo indice: tra 0 e 29
  - Pascal: `var vett : array [0..29] of integer;`
- Array multidimensionali
  - funzione da tipo indice a tipo array
  - in Pascal le seguenti sono equivalenti

```
var mat : array [0..29, 'a'..'z'] of real;
var mat : array [0..29] of array ['a'..'z']
of real;
```
  - ma non sono equivalenti in Ada: la seconda permette *slicing*
  - C fonde array e puntatori



# Memorizzazione degli array

- Elementi memorizzati in locazioni contigue:
  - ordine di riga:  $V[1,1]; V[1,2]; \dots; V[1,10]; V[2,1]; \dots$ 
    - maggiormente usato;
    - il subarray di un array di array (“la terza riga”) vive in locazioni contigue.
  - ordine di colonna:  $V[1,1]; V[2,1]; V[3,1]; \dots; V[10,1]; V[1,2]; \dots$
- Ordine rilevante per efficienza in sistemi con cache

# Array: calcolo indirizzi

- Calcolo locazione corrispondente a  $A[i,j,k]$  (per riga)
  - $A$  : array[ $l1..u1, l2..u2, l3..u3$ ] of elem\_type;  
     $S3$ : dimensione di (un elemento di) elem\_type  
     $S2 = (u3-l3+1)*S3$       dimensione di una riga  
     $S1 = (u2-l2+1)*S2$       dimensione di un piano
  - locazione di  $A[i,j,k]$  è:  
     $\alpha$       indirizzo di inizio di  $A$   
     $+ (i-l1)*S1$       = ind di inizio del piano di  $A[i,j,k]$   
     $+ (j-l2)*S2$       = ind di inizio della riga di  $A[i,j,k]$   
     $+ (k-l3)*S3$       = ind di  $A[i,j,k]$
  - se la shape è nota a tempo di compilazione, riorganizza:
    - $i*S1+j*S2+k*S3 + \alpha$
  - se l'indice inizia sempre da zero ( $l1=l2=l3=0$ ) nessuna precomputazione

# Equivalenza e compatibilità tra tipi

- Due tipi  $T$  e  $S$  sono equivalenti se “sono lo stesso tipo” (ogni oggetto di tipo  $T$  è anche un oggetto di tipo  $S$  e viceversa).
- $T$  è compatibile con  $S$  quando oggetti di  $T$  possono essere usati in un contesto dove ci si attende valori  $S$

# Equivalenza per nome

- Due tipi sono equivalenti se hanno lo stesso nome
- Usata in Pascal, Ada, Java
- Equivalenza per nome loose (lasca) (Pascal, Modula-2)
  - una dichiarazione di un alias di tipo non genera un nuovo tipo, ma solo un nuovo nome:

```
type A = record .... end;  
type B = A;
```

- A e B sono due nomi dello stesso tipo.

# Equivalenza strutturale

- Due tipi sono equivalenti se hanno la stessa struttura:

**Definizione 8.3 (Equivalenza strutturale)** *L'equivalenza strutturale fra tipi è la (minima) relazione d'equivalenza che soddisfa le seguenti tre proprietà:*

- *un nome di tipo è equivalente a se stesso;*
  - *se un tipo  $T$  è introdotto con una definizione `type T = espressione`,  $T$  è equivalente a `espressione`;*
  - *se due tipi sono costruiti applicando lo stesso costruttore di tipo a tipi equivalenti, allora essi sono equivalenti.*
- **Equivalenza controllata per riscrittura:**
    - un tipo complesso riscritto nei suoi componenti elementari
  - **Equivalenza strutturale:** a basso livello, non rispetta l'astrazione che il programmatore inserisce col nome:

# Compatibilità

- T è compatibile con S quando oggetti di T possono essere usati in un contesto dove ci si attende valori S
  - Esempio: `int n; float r; r = r + n;`
- La definizione dipende in modo cruciale dal linguaggio! T è compatibile con S se
  - T e S sono equivalenti;
  - I valori di T sono un sottinsieme dei valori di S (intervallo);
  - tutte le operazioni sui valori di S sono possibili anche sui valori di T (“estensione” di record);
  - i valori di T corrispondono in modo canonico ad alcuni valori di S (int e float);
  - I valori di T possono essere fatti corrispondere ad alcuni valori di S (float e int con troncamento);

# Conversione di tipo

- Se  $T$  compatibile con  $S$  occorre comunque una qualche conversione di tipo. Due meccanismi principali
  - Conversione implicita (detta anche coercizione, coercion): la macchina astratta inserisce la conversione, senza che ve ne sia traccia a livello linguistico;
  - Conversione esplicita, o cast, quando la conversione è indicata nel testo programma.

# Coercizione

- La coercizione serve per indicare una situazione di compatibilità e per indicare cosa deve fare l'implementazione.
- Tre possibilità. I tipi sono diversi ma:
  - con stessi valori e stessa rappresentazione. Esempio: tipi strutturalmente uguali, nomi diversi
    - conversione solo a compile time; no codice
  - valori diversi, ma stessa rappresentazione nell'intersezione. Esempio: intervalli e interi
    - codice per controllo dinamico sull'appartenenza all'intersezione
  - valori e rappresentazione diversi. Esempio: interi e reali.
    - codice per la conversione



# Cast

- In determinati contesti il programmatore deve inserire esplicite conversioni di tipo (cast in C e Java)
  - annotazioni nel linguaggio che specificano che un valore di un tipo deve essere convertito in un altro tipo.

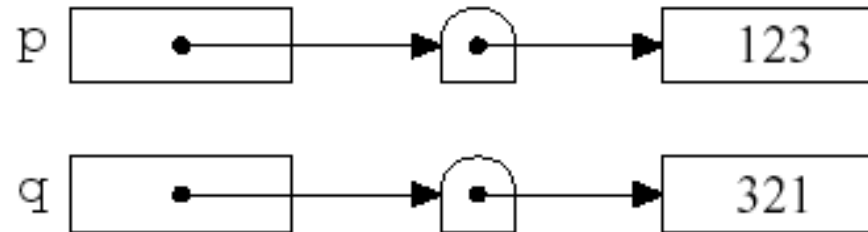
`s s = (s) t`

```
r = (float) n;  
n = (int) r;
```

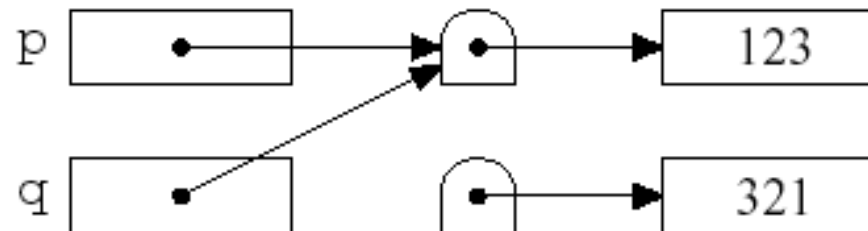
- Tre casi analoghi a quelli delle coercizioni
- Non ogni conversione esplicita consentita
  - solo quelle le quali il linguaggio conosce come implementare la conversione.
  - si può sempre inserire un cast laddove esiste una compatibilità (utile per documentazione)
- Linguaggi moderni tendono a favorire i cast rispetto coercizioni

# Tombstones

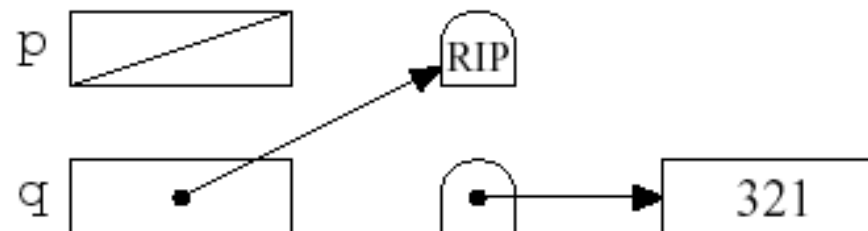
```
p=malloc();  
q=malloc();  
*p=123;  
*q=321;
```



```
q=p;
```

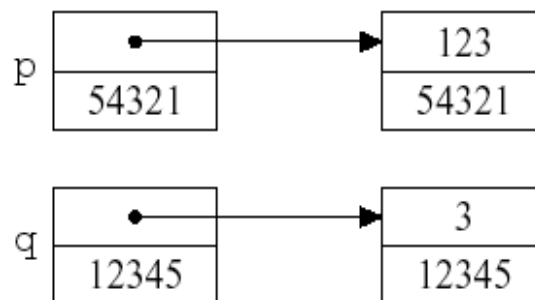


```
free(p);
```

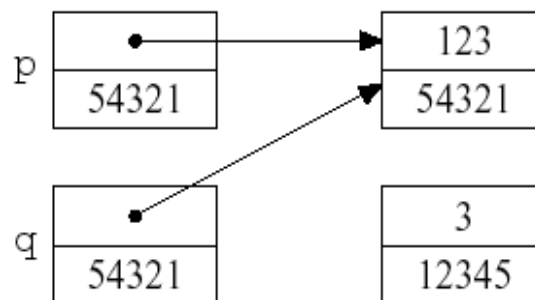


# Locks and keys

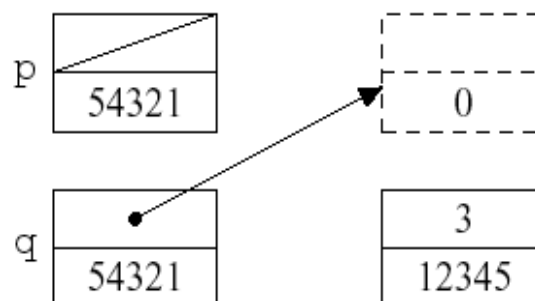
```
p=malloc();  
q=malloc();  
*p=123;  
*q=3;
```



```
q=p;
```



```
free(p);
```



potenzialmente  
riallocato

# Garbage collection

- L'utente alloca liberamente memoria
- Non è permesso deallocare memoria
- Il sistema periodicamente recupera la memoria allocata e non più utilizzabile
  - non utilizzabile = senza un cammino valido di accesso

# Garbage collection: mark and sweep

1. Marca tutti gli oggetti sullo heap come **unused**
  2. Partendo dai puntatori fuori dello heap, visita tutte le strutture concatenate, marcando ogni oggetto come
  3. Recupera dallo heap tutti gli oggetti rimasti **unused**
- 
- Uso spazio per inizio/fine blocco su heap; riconoscere i puntatori
  - Uso spazio per la pila della visita in (2)
    - quando il GC gira lo spazio è limitato! : pointer reversal (Schorr and Waite)
  - Stop-the-world effect: quando lo spazio viene recuperato l'utente sperimenta un sensibile rallentamento nella reazione del sistema
    - GC incrementali (pe Java)

# Capitolo 11

## Astrazione sui dati

# Astrazione e modularità

- **Componente**
  - Unità di programma
    - funzione, struttura dati, modulo
- **Interfaccia**
  - Tipi e operazioni definiti in un componente che sono visibili *fuori* del componente stesso
- **Specifica**
  - Funzionamento “inteso” del componente, espresso mediante proprietà osservabili attraverso l'interfaccia
- **Implementazione**
  - Strutture dati e funzioni definiti *dentro* al componente, non necessariamente visibili da fuori

# Esempio: Tipo di dato

- Componente

- Coda a priorità: struttura dati che restituisce elementi in ordine decrescente di priorità

- Interfaccia

- Tipo `PrioQueue`
- Operazioni `empty : PrioQueue`  
`insert : ElemType * PrioQueue → PrioQueue`  
`deletemax : PrioQueue → ElemType * PrioQueue`

- Specifica

- `insert` aggiunge all'insieme di elementi memorizzati
- `deletemax` restituisce l'elemento a max priorità e la coda degli elementi rimanenti



# Quale supporto linguistico per l'astrazione ?

- Astrazione sul controllo
  - Nascondi la realizzazione nel corpo di procedure
- Astrazione sui dati
  - Nascondi decisioni sulla rappresentazione delle strutture dati e sull'implementazione delle operazioni
  - Esempio: una coda a priorità realizzata mediante
    - un albero binario di ricerca
    - un vettore parzialmente ordinato
- Quale supporto linguistico è fornito da un linguaggio a questo “nascondimento” dell'informazione (*information hiding*) ?

# Principio di incapsulamento

- Indipendenza dalla rappresentazione

Due implementazioni corrette di un tipo (astratto) non sono distinguibili dai clienti di quel tipo

- Le implementazioni sono modificabili senza con ciò interferire con alcun cliente
- Perché il cliente non ha alcun modo per accedere all'implementazione

# Moduli

- Costrutto generale per lo *information hiding*
  - disponibile in linguaggi imperativi, funzionali, ecc.
- Due parti
  - Interfaccia:  
Un insieme di nomi e relativi tipi
  - Implementazione:  
Dichiarazioni (di tipi e funzioni) per ogni nome dell'interfaccia  
Dichiarazioni aggiuntive nascoste (al cliente)
- Esempi:
  - moduli di Modula, packages di Ada, strutture di ML, ...

# Capitolo 12

Il paradigma orientato agli oggetti

# Verso un nuovo *paradigma*

- i principi di astrazione richiedono di “trattare assieme” i dati e le operazioni su di essi
- tuttavia:
  - gli ADT sono rigidi
  - permettono solo difficilmente estensioni
- Una diversa prospettiva (e nuovi supporti linguistici)...

# Obiettivi

- Costrutti per
  - Information hiding e incapsulamento
  - riuso del codice (ereditarietà)
  - compatibilità tra tipi (sottotipi)
  - selezione dinamica delle operazioni

# Oggetti e classi

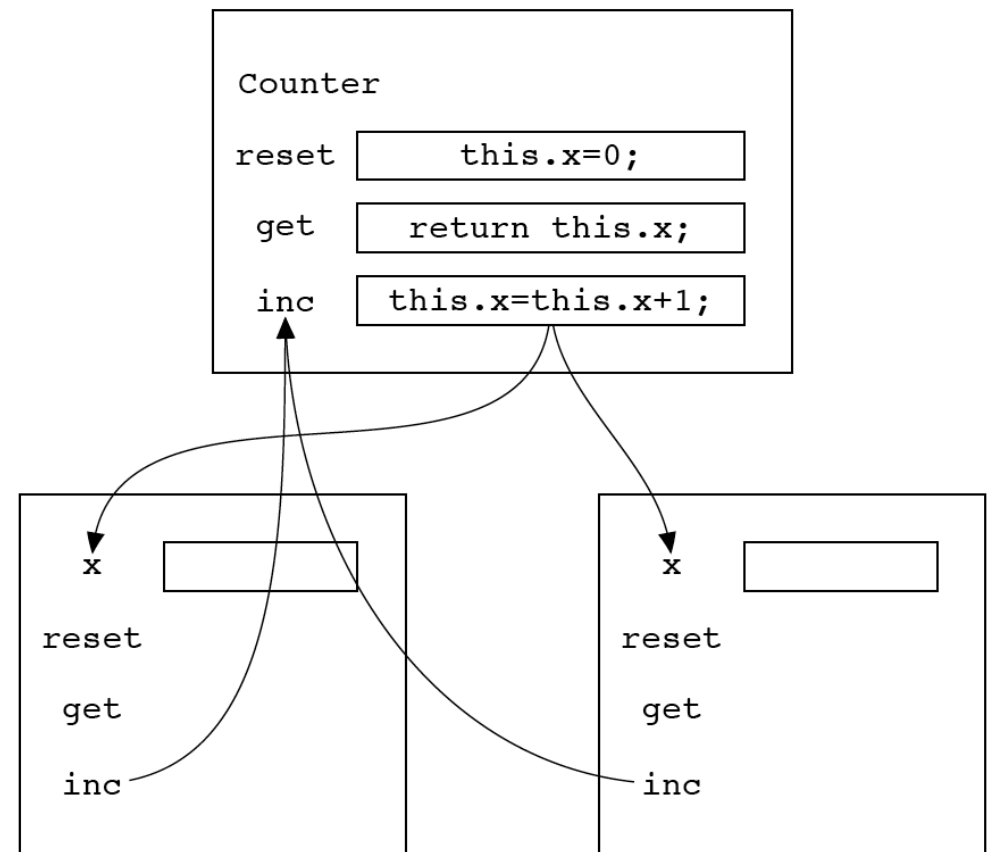
- Un oggetto è una capsula che contiene
  - dati nascosti: variabili, valori (talvolta anche operazioni)
  - operazioni pubbliche: operazioni, dette *metodi*
- Un programma orientato agli oggetti
  - mandare messaggi agli oggetti
  - un oggetto risponde al msg
  - stato confinato negli oggetti
- *Principi di organizzazione* permettono di raggruppare gli oggetti con la stessa struttura (p.e. le classi). Gli stessi principi consentono *estensibilità* e *riuso*.
- Astrazione sui dati e sul controllo, information hiding e incapsulamento sono presenti nel progetto sin dall'inizio.

dati nascosti	
msg <sub>1</sub>	metodo <sub>1</sub>
...	...
msg <sub>n</sub>	metodo <sub>n</sub>

# Classi

- Una classe è un modello di un insieme di oggetti:
  - quali dati
  - nome, segnatura e visibilità delle operazioni (metodi)
- Oggetti creati dinamicamente per *istanziatura* di una classe, mediante *costruttori*

```
class Counter{  
    private int x;  
    public void reset () {  
        x = 0;  
    }  
    public int get () {  
        return x;  
    }  
    public void inc () {  
        x = x+1;  
    }  
}
```





# Incapsulamento

- Le classi garantiscono incapsulamento
  - opportuni *modificatori* assicurano che determinati campi sono
    - pubblici
    - privati
    - protetti
  - la parte pubblica costituisce l' “*interfaccia*” della classe
  - la parte privata è l'implementazione
- Gli altri meccanismi assicurano che l'incapsulamento sia compatibile con l'estensibilità e la modificabilità del codice

# Sottotipi

```
class Counter{
    private int x;
    public void reset() {
        x = 0;
    }
    public int get() {
        return x;
    }
    public void inc() {
        x = x+1;
    }
}
```

```
class NamedCounter extending Counter{
    private String nome;
    public void set_name(String n){
        nome = n;
    }
    public String get_name(){
        return nome;
    }
}
```

- **NamedCounter** è una sottoclasse (o classe derivata) di **Counter**:
  - ogni istanza di **NamedCounter** risponde a tutti i metodi di **Counter** (più altri)
  - In quanto tipi, **NamedCounter** è *compatibile* con **Counter**
  - **NamedCounter** è *un sottotipo* di **Counter**

# Ridefinizione di metodo (overriding)

```
class NewCounter extending Counter{  
    private int num_reset = 0;  
    public void reset() {  
        x = 0;  
        num_reset = num_reset + 1;  
    }  
    public int quanti_reset() {  
        return num_reset;  
    }  
}
```

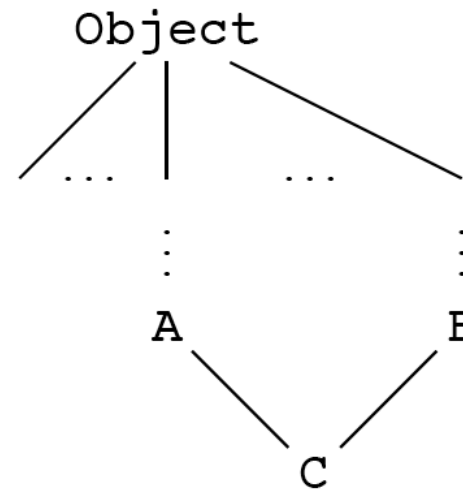
- **NewCounter** contemporaneamente:
  - estende l'interfaccia di **Counter** con nuovi campi
  - ridefinisce il metodo **reset**
- Un messaggio **reset** inviato ad un'istanza di **NewCounter** causa l'invocazione del *nuovo* codice

# La relazione di sottotipo

```
abstract class A{
    public int f();
}

abstract class B{
    public int g();
}

class C extending A,B{
    private x = 0;
    public int f(){
        return x;
    }
    public int g(){
        return x+1;
    }
}
```



- Un tipo può avere *più di un* sovratipo immediato
- La situazione si presenta in Java solo quando i sovratipi sono *classi totalmente astratte* (cioè senza implementazioni: nel gergo Java tali “classi” si chiamano *implementazioni*)
- La gerarchia dei sottotipi non è un albero

# Ereditarietà

```
class Counter{
    private int x;
    public void reset() {
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}
```

```
class NewCounter extending Counter{
    private int num_reset = 0;
    public void reset(){
        x = 0;
        num_reset = num_reset + 1;
    }
    public int quanti_reset(){
        return num_reset;
    }
}
```

- **NewCounter** *eredita* da **Counter** i metodi (e i campi) non ridefiniti
- L'ereditarietà permette il riutilizzo del codice in un contesto estendibile:
  - ogni modifica all'implementazione di un metodo in una classe è automaticamente disponibile a tutte le sottoclassi.

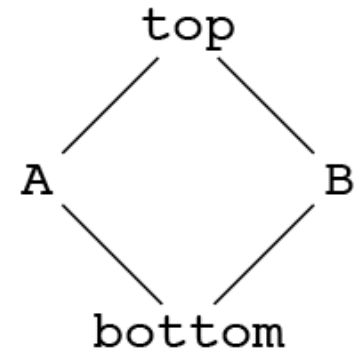
# Ereditarietà e sottotipo

- Sottotipo
  - ha a che vedere con la possibilità di usare un oggetto in un altro contesto
  - è una relazione tra le “interfacce” di due classi.
- Ereditarietà
  - ha a che vedere con la possibilità di riusare il codice che manipola un oggetto
  - è una relazione tra le “implementazioni” di due classi.
- Due meccanismi del tutto indipendenti
- (spesso) linguisticamente collegati nei linguaggi o.o
  - sia C++ che Java hanno costrutti che introducono *contemporaneamente* entrambe le relazioni tra due classi
  - p.e.: **extends** in Java introduce un sottotipo e *può* introdurre ereditarietà se ci sono metodi della superclasse che non sono ridefiniti nella sottoclasse

# Ereditarietà singola e multipla

- Singola (Java)
  - ogni classe eredita *al più* da una sola superclasse immediata
- Multipla (C++)
  - una classe *può* ereditare da più di una superclasse immediata
  - Problemi:
    - name clash:  
un'istanza di **Bottom** eredita **f** da **B** o da **Top**?
    - implementazione

```
class Top{
    int w;
    int f(){
        return w;
    }
}
class A extending Top{
    int x;
    int g(){
        return w+x;
    }
}
class B extending Top{
    int y;
    int f(){
        return w+y;
    }
    int k(){
        return y;
    }
}
class Bottom extending A,B{
    int z;
    int h(){
        return z;
    }
}
```



# Selezione dinamica dei metodi

- Un metodo *m* viene invocato su un oggetto *ogg*:
  - vi possono essere più versioni di *m* (per overriding e sottotipi)
  - come avviene la scelta di quella davvero invocata?
- Selezione *dinamica*
  - a tempo d'esecuzione, in funzione *del tipo dell'oggetto* che riceve il messaggio
- Attenzione: tipo dell'oggetto, non tipo del riferimento (o nome) di quell'oggetto

```
class Counter{
    private int x;
    public void reset(){
        x = 0;
    }
    public int get(){
        return x;
    }
    public void inc(){
        x = x+1;
    }
}

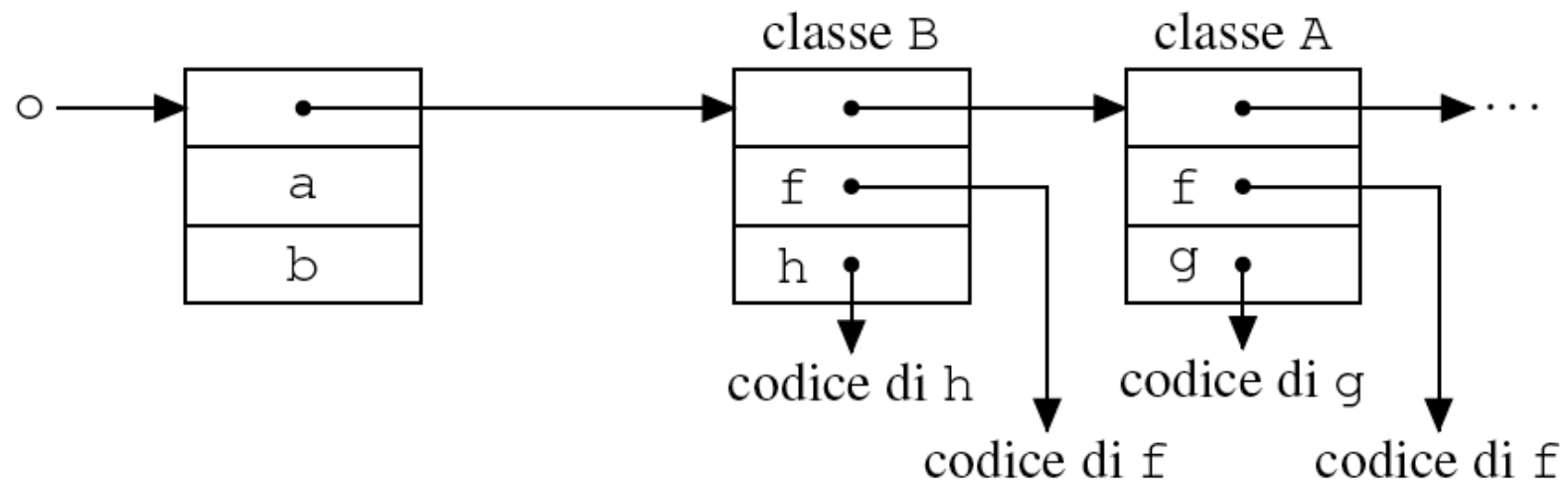
class NewCounter extending Counter{
    private int num_reset = 0;
    public void reset(){
        x = 0;
        num_reset = num_reset + 1;
    }
    public int quanti_reset(){
        return num_reset;
    }
}
```

```
NewCounter n = new Counter();
Counter c;
c = n;
c.reset();
```



# Semplice implementazione dell'ereditarietà

```
class A{  
    int a;  
    void f() {...}  
    void g() {...}  
}  
class B extending A{  
    int b;  
    void f() {...} // ridefinito  
    void h() {...}  
}  
B o = new B();
```



# Ereditarietà singola con tipi statici

```
class A{
    int a;
    char c;
    void g(){...}
    void f(){...}
}
class B extending A{
    int a;
    int b;
    void h(){...}
    void f(){...}
    // ridefinito
}

B pb = new B();
A pa = new A();
A aa = pb;
aa.f();
```

