

# Ambiente e Memoria nella AM di LP

Sommario: 11 marzo, 2015

- Naming: Uso degli identificatori nei LP
- Blocchi: in-line e Procedura
- Ambiente Locale, NonLocale, e Globale
- Scope: Statico e Dinamico
- Memoria: Statica, Stack RDA, Dinamica
- Struttura del Record Di Attivazione, RDA
- Struttura dell'Heap nella gestione Dinamica

# Identificatori e Naming

- **Identificatori**

- Nel Lambda-Calcolo sono usati per nominare i parametri delle funzioni (gli astratti)
- Nei Linguaggi di Programmazione sono usati per nominare i **valori denotabili**

- **Valori Denotabili** sono Strutture che possono essere introdotte nel programma e riferite attraverso nomi.

- V.D. hanno struttura e proprietà anche molto diverse tra loro, a seconda del linguaggio
- V.D. permessi formano una prima caratteristica del linguaggio e dei programmi che possiamo scrivere con esso:
  - valori costanti;
  - valori modificabili;
  - funzioni, procedure e loro parametri
  - tipi;
  - ...

# Ambienti

## Definition (Ambiente)

è l'insieme delle associazioni (identificatore, valore-denotabile) che esistono a run-time, in uno specifico punto del programma e in uno specifico momento dell'esecuzione

- **Dichiarazione** L'introduzione di queste associazioni avviene attraverso costrutti di dichiarazione

```
int f(){
    return 10;
}
int main(void){
    int *x, *y;
    x=(int *)malloc(sizeof(int));
    *x=5;
    y=x;
    *y=f();
    printf("x=%d\n",*x);
    return 0;
}
```

- Indichiamo: (a) dichiarazioni, (b) le associazioni, (c) l'ambiente in 3 punti a scelta, nel programma C sopra.

# Aliasing nell'ambiente

## Definition (Aliasing)

tra identificatori che condividono uno stesso valore denotabile.

- Da evitare quando il valore denotato è un *valore modificabile*

```
int f(){
    return 10;
}
int main(void){
    int *x, *y;
    x=(int *)malloc(sizeof(int));
    *x=5;
    y=x;
    *y=f();
    printf("*x=%d\n",*x);
    return 0;
}
```

- Indichiamo: (a) la presenza di aliasing, (b) aliasing su valore modificabile

## Definition (blocco)

è una regione testuale di programma identificata da un delimitatore di inizio e uno di fine che può contenere dichiarazioni che chiameremo dichiarazioni locali del blocco

- **Blocco in-line** racchiude in modo anonimo sezioni di testo.
  - Sono attraversati come un comando (composto).
  - Possono essere composti tra loro, in sequenza o per annidamento.
- **Blocco Procedura** sono dichiarazioni di procedura/funzione.
  - Hanno un nome e una lista di parametri associato al blocco che forma il corpo della procedura/funzione.
  - Sono attraversati mediante il meccanismo di invocazione.

# Blocchi e Ambienti

## Definition (tipi di ambiente)

Ad ogni blocco è associato un ambiente formato da 3 parti:

- **Locale** delle dichiarazioni locali
- **Non Locale** degli identificatori visibili ma dichiarati in altri blocchi
- **Globale** degli identificatori visibili in tutti i blocchi del programma

- **Ambiente Locale** dipende unicamente dal blocco
- **Ambiente Globale** è unico per il programma, coincide con il locale di un blocco (quello del blocco più esterno del)programma.
- **Ambiente Non Locale** Il vero problema.

# Ambiente Non Locale

- **Ambiente Non Locale** dipende da:
  - tipo di blocco (in-line, procedura)
  - annidamento di blocchi
  - scope statico o dinamico utilizzato dal linguaggio

```
A:{int a =1;
    B:{int b = 2;
        int c = 2;
        C:{int c =3;
            int d;
            d = a+b+c;
            write(d)
        }
        D:{int e;
            e = a+b+c;
            write(e)
        }
    }
}
```

- Solo blocchi in-line. Dipende dall'annidamento: Sono non locali di un blocco, tutti gli identificatori di blocchi che lo racchiudono e non ridefiniti in blocchi intermedi (incluso il blocco stesso)
- (a) chi sono a, b, c nel blocco D? (b) utilizzando tabelle per gli ambienti, mostrare gli ambienti locali

# Regole di Scope

- **Ambiente Non Locale** Quando abbiamo anche blocchi procedura, dobbiamo considerare lo scope dell'identificatore
- **Scope di un identificatore** è l'insieme delle regioni di un programma, in cui tale identificatore (e quindi, il suo legame al valore denotato) è visibile
- I linguaggi usano uno tra due meccanismi di scope:
  - **Scope Statico** l'identificatore è visibile in ogni blocco in-line annidato e ogni blocco procedura dichiarata nel blocco, in cui non sia ridefinito.
  - **Scope Dinamico** l'identificatore è visibile in ogni blocco in-line annidato e ogni blocco procedura di invocazioni nel blocco, in cui non sia ridefinito.

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

- Assunto che il linguaggio usi scope statico: Cosa stampa il programma?
- Assunto che il linguaggio usi scope dinamico: Cosa stampa il programma?

# Scope: Confronto

- **Scope Statico**

- Usato dalla quasi totalità dei L.P.
- Il significato degli identificatori non locali di procedure e funzioni non cambia quando invocate in punti diversi
- Più complesso da gestire, MA
- Esistono tecniche particolarmente efficienti che lo rendono il migliore (accesso in tempo costante ai non locali) se utilizzato da un compilatore

- **Scope Dinamico**

- Introdotto dal linguaggio Lisp e utilizzato in dialetti di Lisp
- Il significato degli identificatori non locali di procedure e funzioni dipende da dove sono invocate.
- Più semplice da gestire da interpreti del linguaggio, MA
- accesso alle non locali non è in tempo costante (e dipende dal livello di annidamento dei blocchi nel programma)

# Memoria

- **Memoria**

- Deve contenere
  - (a) il programma (Codice Oggetto o Abstract Syntax Tree)
  - (b) tutte le informazioni per gestire l'esecuzione (stato del programma, Activation Record)
  - (c) i dati su cui opera il programma
- Tre principali tipi di memoria:
  - Memoria Statica
  - Memoria Dinamica a Stack
  - Memoria Dinamica a Heap
- Le Macchine Astratte dei Linguaggi di oggi, richiedono tutti e tre questi tipi
- che possono altrimenti, essere emulati su Macchine Astratte con memoria del solo tipo Statica.

# Memoria Statica

- **Memoria Statica.**
  - L'allocazione delle strutture è fatta prima dell'esecuzione del programma.
  - L'allocazione della memoria non può essere modificata dell'esecuzione del programma.
- Per quale dei contenuti (a), (b), (c) (vedi slide precedente) è adatta?
  - (a). Adatta a memorizzare il programma (se il linguaggio non prevede riflessione<sup>1</sup> sui programmi)
  - Ma per (b) e (c) in generale, la risposta è non adatta
- Quali caratteristiche di un LP per poter utilizzare Memoria statica anche in (b) e in (c)?
  - (b) nessun meccanismo di ricorsione
  - (c) nessun meccanismo di allocazione dinamica di valori (malloc, liste,..)

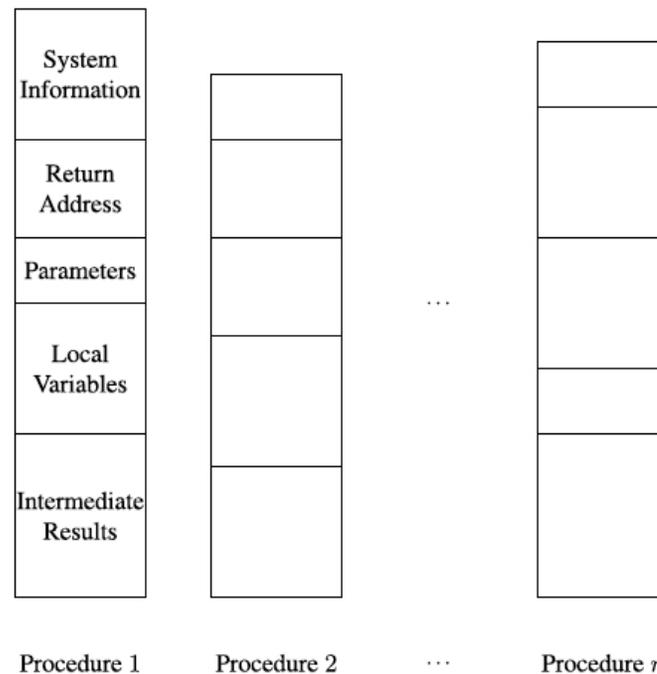
---

<sup>1</sup>l'esecuzione del programma può modificare il programma in esecuzione  11/24

# Macchina Astratta con sola Memoria Statica

- **Organizzazione della memoria quando**

- (b) nessun meccanismo di ricorsione
- (c) nessun meccanismo di allocazione dinamica di valori

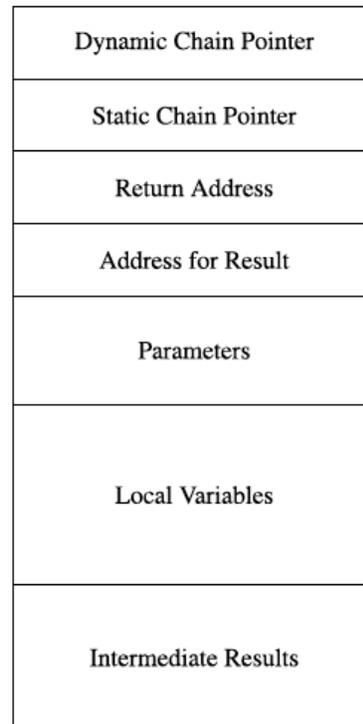


- La figura mostra solo (b) e per i soli blocchi procedura, completarla per:
- (b) per blocchi in-line
- (a) per mem. programma; (c) per mem. dati

# Memoria Dinamica a Stack

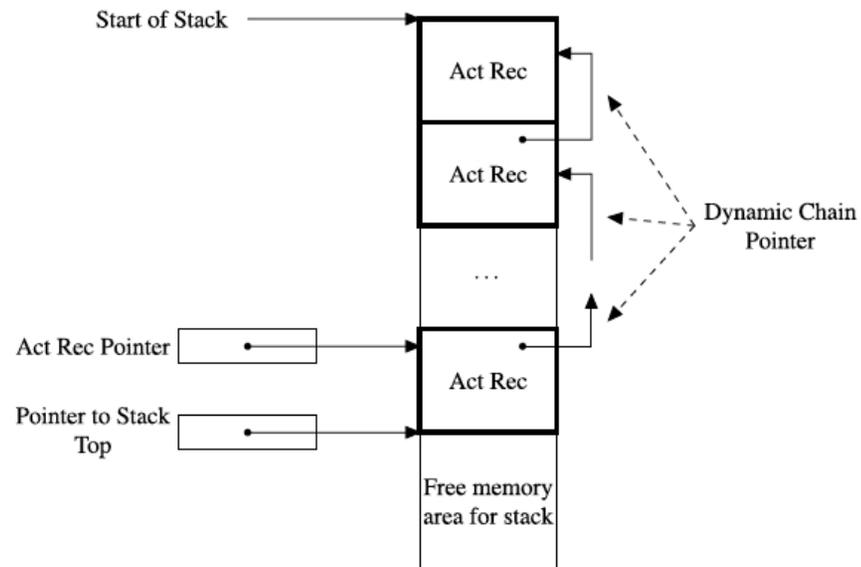
- **Memoria Dinamica a Stack.**
  - Utilizzata per (b) (gestire il controllo dell'esecuzione) in presenza di meccanismi per ricorsione
  - **Activation Record** Le informazioni sull'esecuzione di ogni blocco sono organizzate in AR che hanno forma prossima a quella mostrata nella slide precedente.
  - Perchè M. Dinamica?
    - Ogni invocazione ricorsiva richiede di allocare un nuovo AR per la procedura in cui mettere le nuove informazioni di controllo quali i valori dei nuovi parametri, il nuovo indirizzo di ritorno,...
  - Perchè organizzata in Stack?
    - 1) L'attraversamento di un blocco in inline, 2) l'invocazione di procedura ricorsiva, generano una sequenza di allocazioni di AR, in corrispondenza a:
      - 1) blocchi annidati nel blocco;
      - 2) invocazioni ricorsive attivate nell'invocazione;che devono essere deallocate e/o rilasciate in ordine inverso alla loro creazione

# AR per Memoria a Stack



- Ruolo e contenuto dei componenti

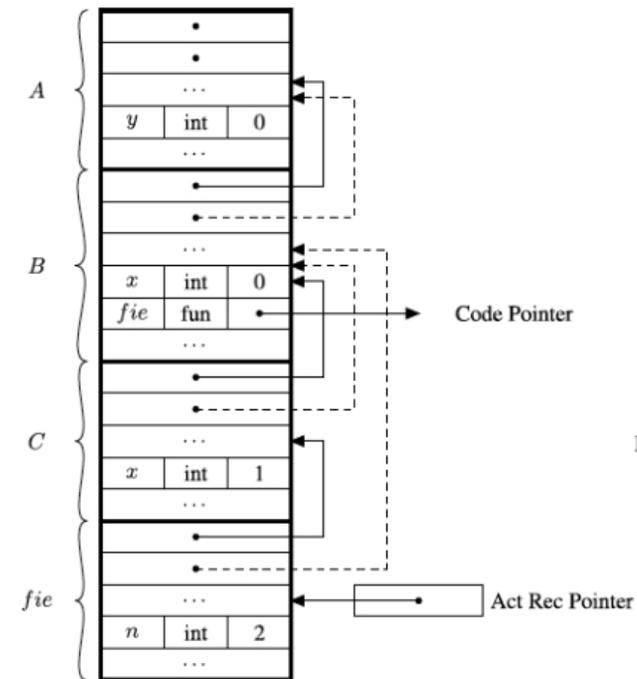
# Organizzazione di Memoria a Stack



- Ruolo e contenuto dei componenti

# Stato della Macchina e Stack di AR

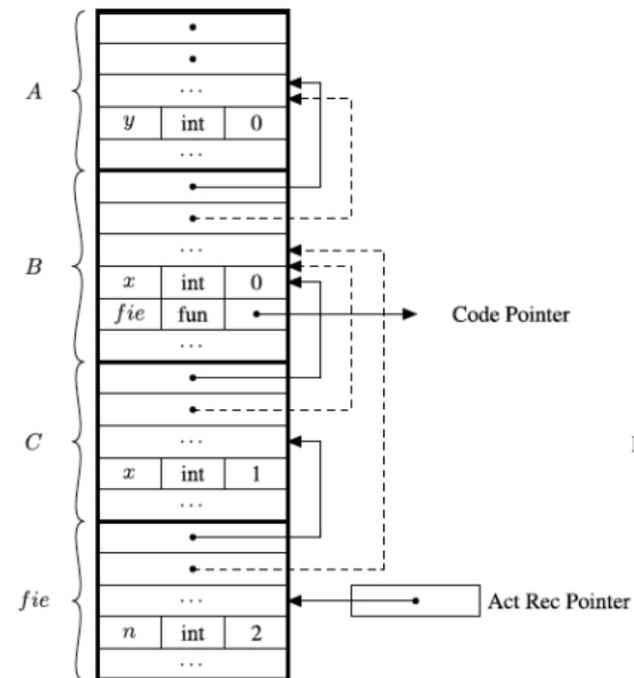
```
A:{int y=0;
  B:{int x = 0;
    void (int n){
      x = n+1;
      y = n+2;
    }
    C:{int x = 1;
      fie(2);
      write(x);
    }
  }
  write(y);
}
```



- Stack di AR = Parte centrale dello stato (insieme alla memoria per programma e dati)
- Contiene tutte le informazioni per il controllo dell'esecuzione, incluse quelle sullo scope degli identificatori
- Durante la computazione del programma:
  - (a) Si esegue dichiarazione d, comando c, o espressione e: Operiamo sull'AR, top dello stack, interpretiamo d/c/e, quindi (a)-(d),
  - (b) Si entra in un blocco inline: Push dell'AR associato al blocco, quindi (a)-(d)
  - (c) Si entra in blocco procedura invocata: Push dello AR creato dall'invocazione, quindi (a)-(d)
  - (d) Si esce da blocco: Pop, quindi (a)-(d)

# Stato della Macchina e Stack di AR: Dich./Com/Exp

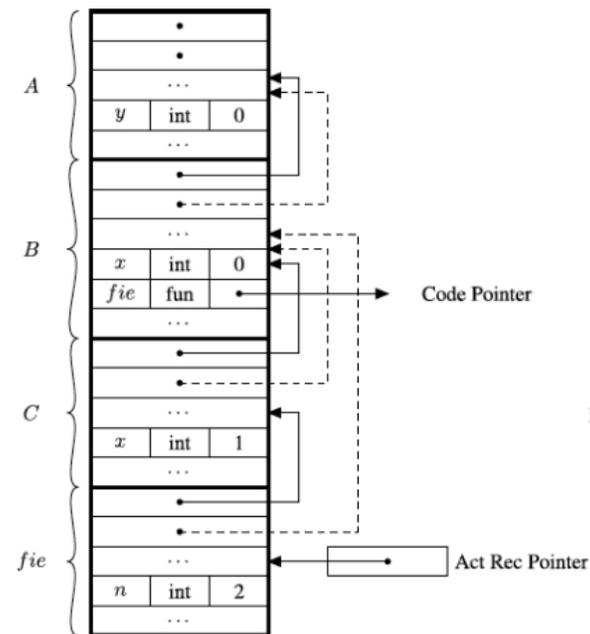
```
A: {int y=0;
   B: {int x = 0;
      void (int n){
        x = n+1;
        y = n+2;
      }
   C: {int x = 1;
      fie(2);
      write(x);
    }
  }
write(y);
}
```



- Stack di AR = Parte centrale dello stato (insieme alla memoria per programma e dati)
- Durante la computazione del programma:
  - (a) Si esegue dichiarazione *d*, comando *c*, o espressione *e*: Operiamo sull'AR, top dello stack, interpretiamo *d/c/e*, quindi (a)-(d),
    - Interpretazione ed esecuzione di:
      - dichiarazione: Aggiunge legami nell'ambiente locale e può modificare/allocare memoria
      - comando: modifica/alloca memoria (dati/prog.) oppure è un'invocazione
      - invocazione: creazione AR, trasmissione parametri, collegamenti tra AR invocante e creato ...
      - espressione: usa area val. intermedi, può modificare/allocare memoria (datt/prog) oppure è un'invocazione...
  - Trovare in figura tutte le operazioni dette per (a) e mostriamo la memoria

# Stack di AR: Scope degli identificatori

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



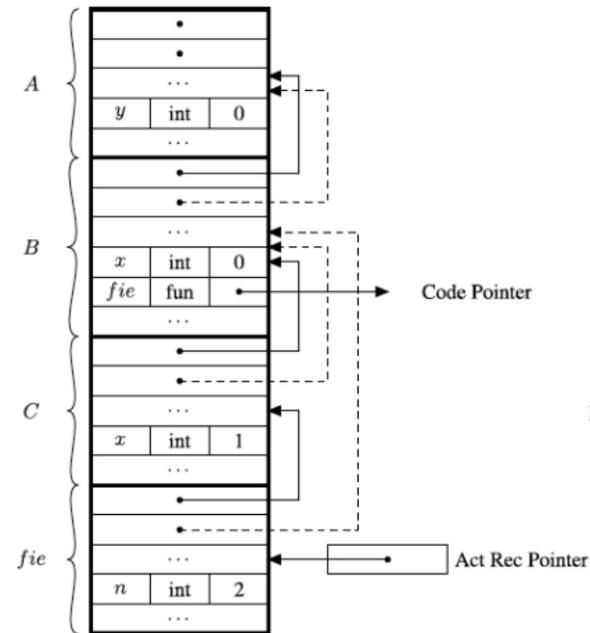
- Stack di AR = Parte centrale dello stato (insieme alla memoria per programma e dati)
- Durante la computazione del programma,
  - si incontrano identificatori e si deve accedere alle loro denotazioni.
  - Come si trovano a partire dall'AR, top dello stack?
    - Scope Statico:
      - (1) Ambiente Locale dell'AR. Se non è trovato, allora:
      - (2) Percorrere Catena Statica, selezionando lo AR immediatamente collegato e torna in (1)
    - Scope Dinamico:
      - (1) Ambiente Locale dell'AR. Se non è trovato, allora:
      - (2) Percorrere Catena Dinamica, selezionando lo AR immediatamente collegato e torna in (1)
  - Trovare in figura i legami degli identificatori non locali e
  - Dire se il programma ha stesso comportamento assunto scope Statico oppure Dinamico.

# Scope ad accesso diretto: Display e CRT. Display

```

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

```



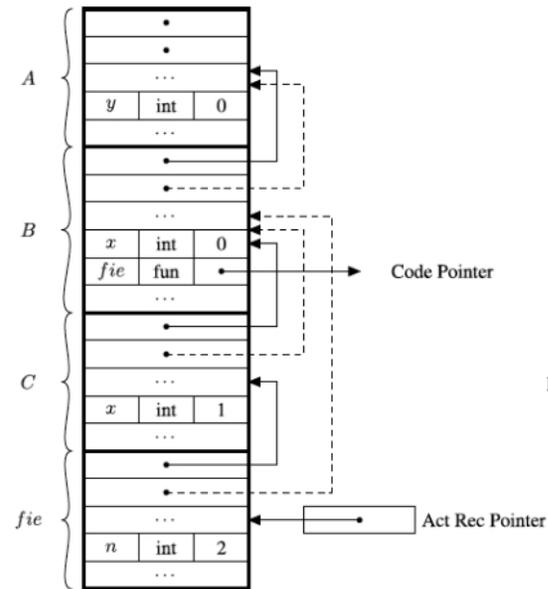
- Display
  - Scope Statico. Tecnica implementativa standard per compilatori/interpreti. Sintassi astratta.
  - Ogni identificatore occorrente in un blocco è rimpiazzato dalla coppia (cs,pe), indicante:
    - cs = livello di annid. in c. statica del blocco di definizione rispetto al blocco di occorrenza
    - pe = posizione legame identificatore nell'ambiente del blocco di definizione $y = n + 2$  diventa  $(2, 0) = (0, 0) + 2$ .
  - Ogni blocco ha un Display = vettore contenente gli indirizzi degli AR dei blocchi nella c. statica: Ad es., il b. di fie ha D.  $[AR_{fie}, AR_B, AR_A]$  con indici 0,1,2 da sin. a destra.
- CRT
  - Scope Dinamico. Tecnica implementativa utilizzata in alcuni interpreti (expensive).

# Scope ad accesso diretto: Display e CRT. CRT

```

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

```



## ● CRT

- Scope Dinamico. Tecnica implementativa utilizzata in alcuni interpreti (expensive).
- Una tabella unica per gli identificatori di tutti i blocchi
- Una riga organizzata a pila per ogni nome usato
- A ogni accesso di un blocco:
  - gli identificatori dichiarati sono inseriti nel top, delle giuste righe, come coppia  $(AR_x, pe_I)$ :
    - $AR_x$  è indirizzo dell'AR del blocco, e
    - $pe_I$  è posizione dell'identificatore nell'ambiente locale del blocco

Ad es., all'entrata, a sinistra, e durante la valutazione, a destra, del blocco che forma il corpo di fie,

riga x: $(AR_B, 0), (AR_C, 0)$	riga x: $(AR_B, 0), (AR_C, 0)$
riga y: $(AR_A, 0)$	riga y: $(AR_A, 0)$
riga n: $\epsilon$	riga n: $(AR_{fie}, 0)$
riga fie: ....	riga fie: ....



# Memoria Dinamica: Allocazione/Dealloc. Dinamica di Dati

- Allocazione Dinamica.
  - Quando e Perché
    - Dati la cui struttura varia durante la loro manipolazione
    - Per questi dati la modifica dei valori dei componenti è una modifica di struttura: Liste, Alberi, Grafi, Pile, Oggetti
  - Dove
    - In alcuni linguaggi è solo per Programmi che manipolano strutture dati dinamiche
      - richiede un controllo esplicito, ad esempio: m/r/c-alloc in C, new in Pascal
    - In altri linguaggi è la gestione standard per i dati strutturati:
      - allocazione implicita e gestita automaticamente dal linguaggio (inglobata nei costruttori di dati)
      - Liste per i Linguaggi Funzionali
      - Oggetti per Linguaggi OO come Java
- Deallocazione Dinamica (e riuso della memoria per la stessa esecuzione)
  - Quando. Tali dati ,o parti di essi, non sono più accessibili
  - Come.
    - Esplicita. Pericolo di rimuovere struttura ancora in uso (free in C)
    - Implicita. Gestita automaticamente dal linguaggio: Garbage Collection



# Memoria Dinamica: Heap/2

- **Heap** è la struttura standard per la memoria dinamica.
  - **Consiste in:**
  - **Dimensione del blocco.** Il problema: Allocare  $K$  words su blocchi di dimensione diversa
    - dimensione **fissa**. Deve essere maggiore di  $K$  (per ogni necessario  $K$ ).
    - dimensione **variabile**. Riunire più blocchi contigui per formare un blocco adeguato
- È la soluzione dei Linguaggi attuali che hanno strutture dinamiche di tipi (e occupazione di memoria) diversi. MA
- Frammentazione Interna: Il blocco allocato ha size maggiore
- Frammentazione Esterna: Blocchi per formare il blocco richiesto ci sono ma non sono contigui.
  - diversamente dai S.O.: quando un p. è in esecuzione, i dati, una volta caricati, non possono essere spostati
  - allocazione fallisce