Astrazione e Gestione della Memoria

Meccanismi di Astrazione

Al programmatore sono forniti tre meccanismi di base per costruire nuovi tipi di dati con relative operazioni:

- dichiarazione di tipi
- sottoprogrammi
- oggetti/ereditarietà

Tipo di Dato Astratto:

- un insieme di **data object**, costruiti tramite una o più definizioni di tipo
- un insieme di **operazioni** astratte su quei dati
- **incapsulamento** del tutto in maniera tale che chi usa il nuovo tipo non possa manipolare i dati di quel tipo se non attraverso le operazioni appositamente definite

information hiding: è il principio secondo cui nella costruzione di nuove astrazioni da parte del programmatore le informazioni non pertinenti devono essere il più possibile nascoste a chi poi usa tali nuove componenti del programma.

Il concetto di incapsulamento rende più forte l'information hiding in quanto a chi usa la nuova componente astratta:

- non è chiesto di conoscere l'informazione nascosta (dettagli implementativi)
- non è permesso di usare/manipolare direttamente l'informazione nascosta (solo indirettamente tramite le operazioni)

Esempi

la funzione per il calcolo della radice quadrata fornita dai linguaggi è un'operazione astratta in quanto all'utente sono nascosti sia i dettagli sulla rappresentazione a bit del numero sia l'algoritmo usato per il calcolo

il tipo `integer` del Pascal non solo nasconde i dettagli sulla rappresentazione a bit del numero ma anche la incapsula in quanto è difficile manipolarne i singoli bit

il tipo `int` del C al contrario non incapsula la rappresentazione numerica:

```
int x = 2;
printf("x = %d\n", x);
((char *)&x)[1] = '\1';
printf("x = %d\n", x);
```

Sottoprogrammi

Meccanismo di astrazione comune a tutti i linguaggi

specifica di un sottoprogramma:

- nome
- segnatura
- funzione sui domini semantici (valori possibili) $\rightarrow$ azione semantica

Esempio: specifica indipendente dal linguaggio

nome $\rightarrow$ `sum`

segnatura $\rightarrow$ `sum:` `int` $\times$ `int` $\rightarrow$ `int`

azione semantica $\rightarrow$
specifica formale:

```
sum(0,y) = y
sum(x+1,y) = (sum(x,y)) + 1
```

`sum(x,y)` = indefinita se `x<0` o `y<0`

specifica in C

nome + segnatura → `int sum(int,int);` (prototipo)

implementazione `sum` →

```
int sum(int x, int y)
{ if (x<0 || y<0)
     return -1;
  else
     if (x==0)
        return y;
     else
        return (sum(x,y) + 1);
}

  ...
     while(x!=0) {
        x = x-1;
        y = y+1;
     }
     return y;
}
```

Problema: l'implementazione di `sum` in C deve essere "conforme" alla specifica formale

La **verifica** della **correttezza** è **indecidibile**

Sono stati proposti vari metodi per dare risposte "parziali" al problema della correttezza: logici, algebrici...

Ricordate le difficoltà nell'avere (sotto)programmi che sono funzioni matematiche:

- operazioni indefinite per certi input
- argomenti impliciti
- effetti collaterali
- history sensitivity
- (eccezioni)

Sottoprogrammi

funzioni: viene restituito esplicitamente un dato

procedure: funzionano per effetti collaterali

Nella specifica di un sottoprogramma vengono dati nomi ai parametri formali

```
void fp(int *q, int x)
```

$\Rightarrow$ `fp: (int *)` $\times$ `int` $\rightarrow$ `void`

Il meccanismo del passaggio dei parametri verrà discusso più avanti

implementazione di un **sottoprogramma** in un linguaggio L

sottoprogramma = nuova operazione macchina astratta di L

Per non fare confusione linguistica con il concetto di "implementazione" del meccanismo dei sottoprogrammi usiamo il termine **definizione di sottoprogramma**

Definizione di Sottoprogramma

La definizione sfrutta strutture dati e operazioni di L

`intestazione`: segnatura + nomi parametri formali

`dichiarazioni locali`: dati + sottoprogr. (non in C!)

`comandi/espressioni`: azioni del sottoprogramma

```
int f(int x)              /* intestazione */
{ int y = 3;              /* dichiarazioni locali */
  y=y+1; return y;}       /* comandi */
```

N.B. i parametri formali sono oggetti dichiarati localmente

ML:

```
fun f(x:int):int          (* intestazione *)
    =
      let val y = 3 in    (* dichiarazione loc *)
        y+x               (* espressione *)
      end;
```

ML:

```
fun f(x:int):int =
    let fun g y = y+1 in  (* dich loc di funz*)
      (g x) + x
    end;
```

Pascal:

```
function f(x:integer):integer;
  function g (y:integer):  integer;
    begin
      g:=y+1
    end;
  begin
    f:= x + g(x)
  end;
```

Implementazione del Meccanismo dei Sottoprogrammi

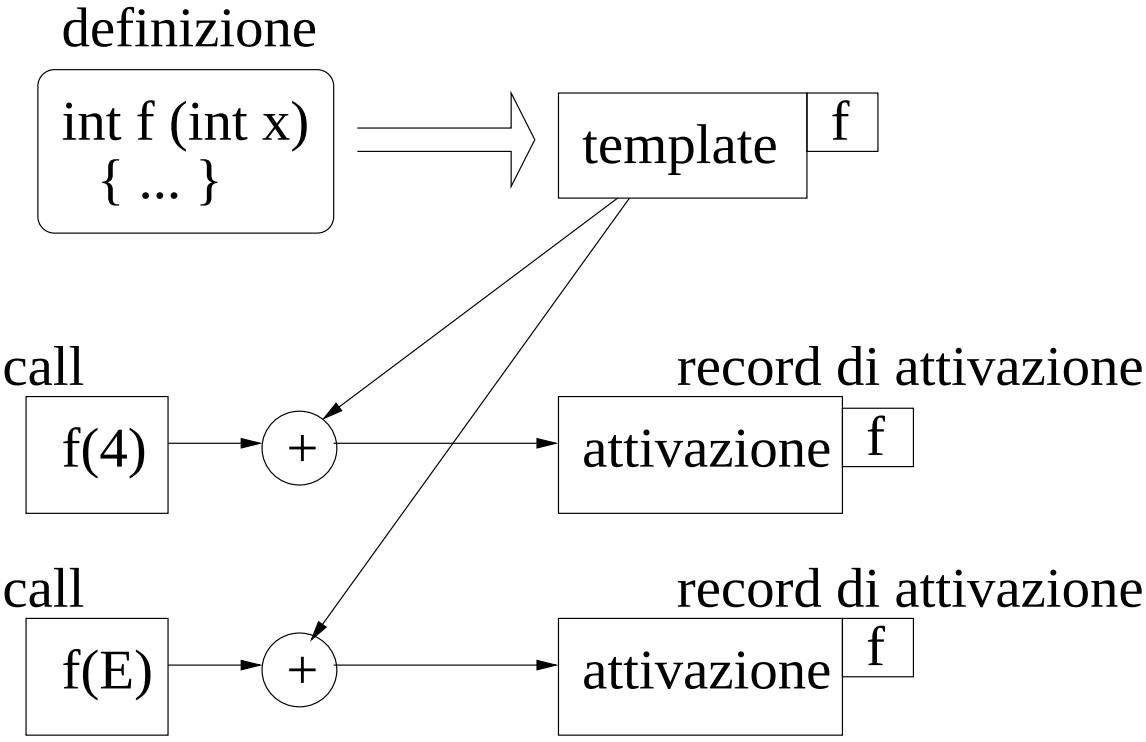
`sottoprogramma`

`definizione (+ dichiarazione)` $\Rightarrow$ `template`

(compile time)

`attivazione (chiamata/invocazione)`

(run time)



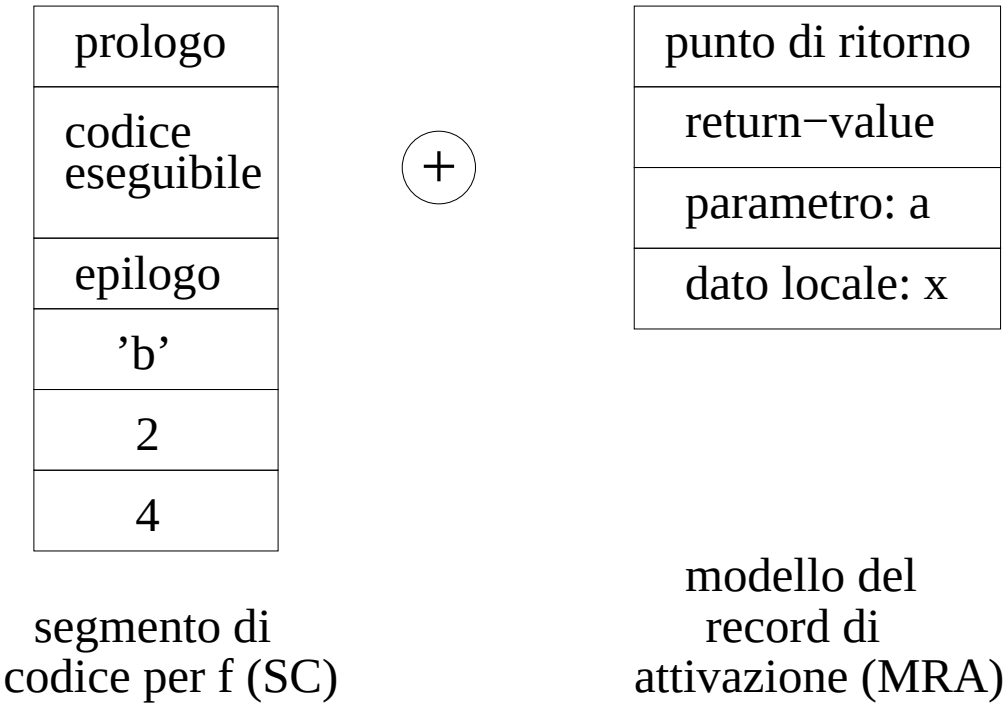
Esempio (C-like)

```
int f (char a)
    {   int x;
        x=4;
        if (a=='b') return 2;
            else return x; }
```

1. memoria per i parametri (`a`)
2. memoria per il return value (`int`)
3. memoria per le variabili locali (`x`)
4. memoria per costanti e letterali (`'b',2,4`)
5. memoria per il codice eseguibile

(4)+(5) invarianti a run-time

TEMPLATE



`SC` $\Rightarrow$ allocato stabilmente in memoria durante il tempo di vita del programma

`MRA` $\Rightarrow$ è gestito come un record (info necessaria: la sua lunghezza; esiste solo virtualmente)

A run time ogni chiamata provoca un'`attivazione`:

- `SC`: è fisso, comune per tutte le chiamate
- `Record di Attivazione (RA)`: uno per ogni chiamata $\Rightarrow$ per l'allocazione è sufficiente conoscere `size(MRA)`

verso il concetto di Memoria Dinamica

Invocazione Sottoprogramma:

$\Rightarrow$ allocazione memoria per RA

$\Rightarrow$ deallocazione memoria per RA

- `SC` vive nella memoria statica
- ogni `RA` vive nella memoria dinamica

invocazione
$\Rightarrow$ gestore memoria $\rightarrow$ alloc. memoria per RA

return
$\Rightarrow$ gestore memoria $\rightarrow$ deallocazione memoria

Definizione di Nuovi Tipi

Pascal

```
type cmpx = record
              r:  real;
              i:  real
            end;
var m:  cmpx;
```

`cmpx` è usato a tutti gli effetti come un tipo primitivo

C

```
struct cmpx {
             float r;
             float i; }
struct cmpx m;
```

il nuovo tipo non è `cmpx` ma `struct cmpx`

Il C offre "zucchero sintattico"

```
typedef struct cmpx complex;
```

provoca solo una macroespansione

```
struct a {
          int x;
          struct a *p; }
```

SI

```
typedef struct {
                int y;
                int z; } A
```

SI

```
typedef struct {
                int x;
                A *p; } A
```

NO

Pascal

```
type l = ↑cell;
     cell = record
              x:  integer;
              p:  l
            end;
```

In C i nuovi tipi sono creabili solo con `struct`, però `typedef` è usato come se fosse un costruttore di tipo (quando possibile) per rendere più leggibile il codice

Implementazione: la definizione di nuovi tipi è usata solo durante la traduzione. Il traduttore inserisce le info relative al tipo in una tabella che consulta quando incontra un qualche oggetto di quel tipo sia per produrre il codice eseguibile appropriato sia per type checking.

Equivalenza di Tipi

Quand'è che

$$T_1 \text{ equivalente } T_2?$$

Esempio Pascal

```
type v = array[1..5] of real;
     w = array[1..5] of real;

var x:v; y:w;

procedure p(z:v)
     ...
     end;
...
x:=y;                errore di tipo?
p(y);                errore di tipo?
...
```

soluzione 1: controllo dei tipi forte

T_1 equivalente $T_2 \Leftrightarrow T_1 = T_2$

$\rightarrow$ equivalenza di nomi

soluzione 2: equivalenza strutturale

T_1 equivalente $T_2 \Leftrightarrow$

T_1 e T_1 hanno la stessa struttura interna

$\rightarrow$ vari livelli di equivalenza

Nomi dei Tipi

L'equivalenza di tipi diventa ancor più interessante se pensiamo anche ai nomi dei tipi, importanti per:

- leggibilità

```
fun f(x:bool * real):  bool * real = ...

type t = bool * real;
fun f(x:  t):  t = ...
```

- migliorare la correttezza del programma accorpando i dati in accordo al loro significato nel programma

```
type temperatura = real;
type angolo = real;
fun allarmeTemperatura(x:temperatura):...
fun allarmeAngolo(x:angolo):  ...
var t:  temperatura; a:  angolo;
allarmeTemperatura(t) (* OK *)
a:=t (* Errore di Tipo *)
allarmeTemperatura(a) (* Errore di Tipo *)
```

Per questo potrebbe essere utile a volte che dei tipi strutturalmente equivalenti fossero trattati come non-equivalenti $\rightarrow$ equivalenza di nomi

Certo è che l'equivalenza di nomi è molto restrittiva

I LdP per lo più adottano soluzioni miste

Equivalenza dei Tipi in C

Il C usa equivalenza strutturale per array e funzioni, e equivalenza di nomi per `struct`, `union` e `enum`.

```
char a[100];
void f(char b[]);
f(a); /* ok */

struct polar{float x; float y;}
struct rect{float x; float y;}
struct polar a;
struct polar b;
a = b; /* errore di tipo */
```

L'uso di `typedef` è solo zucchero sintattico.

Questa politica rende semplice l'equivalenza di tipi ricorsivi, che possono essere definiti solo usando la `struct`:

```
struct T1{float x; struct T1 *y;} a;
struct T2{float x; struct T1 *y;} b;
a = b; /* errore di tipo */
```

Equivalenza dei Tipi in ML

l'ML adotta l'equivalenza strutturale, eccetto che per ciascuna dichiarazione `datatype`:

```
datatype polar = POLAR of real * real;
datatype rect = RECT of real * real;
val a = POLAR(1.0, 2.0);
val b = RECT(1.0, 2.0);
if (a=b) ...  (* errore di tipo *)
```

Nota che l'obbligatorietà dell'uso dei costruttori (`POLAR`, `RECT`) rende possibile l'identificazione univoca dei tipi dei letterali.

`datatype` non richiede necessariamente la dichiarazione di `tuple`:

```
datatype fahrenheit = F of real;
datatype centigrade = C of real;
val a = F 150.0;
val b = C 150.0;
if (a=b) ...  (* errore di tipo *)
fun convert(F x) = C(1.8 *x + 32.0); (* ok *)
```

Anche qui l'uso di `type` è solo zucchero sintattico.

Uguaglianza di Oggetti

`a:` T_1 `b:` T_1

`a=b`?

solo per pochi tipi elementari è prevista l'uguaglianza

Data `struct {int a[10]; } aa, bb;` il test

```
(aa == bb)
```

nasconde un test di uguaglianza tra array
$\Rightarrow$ compito del programmatore

In ML non è possibile fare il test di uguaglianza tra reali

Tipi Parametrici

Ammessi in Ada, ML.

stack di interi:

`<int-stack, empty, push, top, pop>`

`empty:` $\texttt{void} \rightarrow \texttt{int-stack}$
`push:` $\texttt{int-stack} \times \texttt{int} \rightarrow \texttt{int-stack}$
`top:` $\texttt{int-stack} \rightarrow \texttt{int}$
`pop:` $\texttt{int-stack} \rightarrow \texttt{int-stack}$

e se volessimo il tipo "stack di reali"?

- definire un nuovo tipo di dato
- ricorrere ai tipi di dato parametrici
  <$\{TP_i\}_{i \in I}$, $\{T_j\}_{j \in J}$, $\{OP_k\}_{k \in K}$ >
  $\{TP_i\}_{i \in I} \rightarrow$ parametri di tipo
  $\{T_j\}_{j \in J} \rightarrow$ tipi, evententualmente dipendenti dai parametri di tipo

stack parametrico:

```
<T, stack(T), empty, push, top, pop>
```

$$\texttt{empty: void} \rightarrow \texttt{stack(T)}$$
$$\texttt{push: stack(T)} \times \texttt{T} \rightarrow \texttt{stack(T)}$$
$$\texttt{top: stack(T)} \rightarrow \texttt{T}$$
$$\texttt{pop: stack(T)} \rightarrow \texttt{stack(T)}$$

Esempio ML:

```
datatype 'a stack = empty | ++ of 'a * 'a stack;
```

(++ è il costruttore)

Implementazione Tipi Parametrici

Le definizioni di tipo parametrico sono usate come template esattamente come le altre definizioni di tipo, con la differenza che quando il compilatore incontra la dichiarazione di una variable con un parametro associato al tipo, esso per prima cosa riempie il valore del parametro con quello incontrato in modo da ottenere una definizione senza parametro.

E se ai sottoprogrammi vengono passati parametri di tipo parametrico (quindi di dimensione non fissata)?

(pp. 372-373 Pratt& Zelkowitz, III ed.)

Gestione della Memoria

Principali elementi che richiedono allocazione di memoria:

- segmenti di codice utente
- supporto run-time
- dati e costanti utente (statici)
- strutture dati utente dinamiche
- valori temporanei nella valutazione di espressioni
- spazio per la trasmissione dei parametri
- buffer di I/O

Problemi

- allocazione
- recupero
  - esplicito (dispose, free...)
  - implicito (garbage collection in L funzionali)
- compattamento

Meccanismi utente di allocazione

- dichiarazione
- allocazione esplicita tramite puntatori
- operazioni primitive che richiedono memoria (es. `cons` del LISP)

Gestione memoria a carico dell'utente?

SI $\rightarrow$ conoscenza puntuale delle necessità di memoria per i dati

NO $\rightarrow$

- possibili perdite di informazione
- interferenze col sistema

mix delle due possibilià?

Fasi di gestione della memoria

`allocazione` $\Rightarrow$ `recupero` $\Rightarrow$ `compattazione`

Metodi di gestione

- statica
- a stack (pila)
- a heap
  - recupero spazzatura
  - perdita informazioni

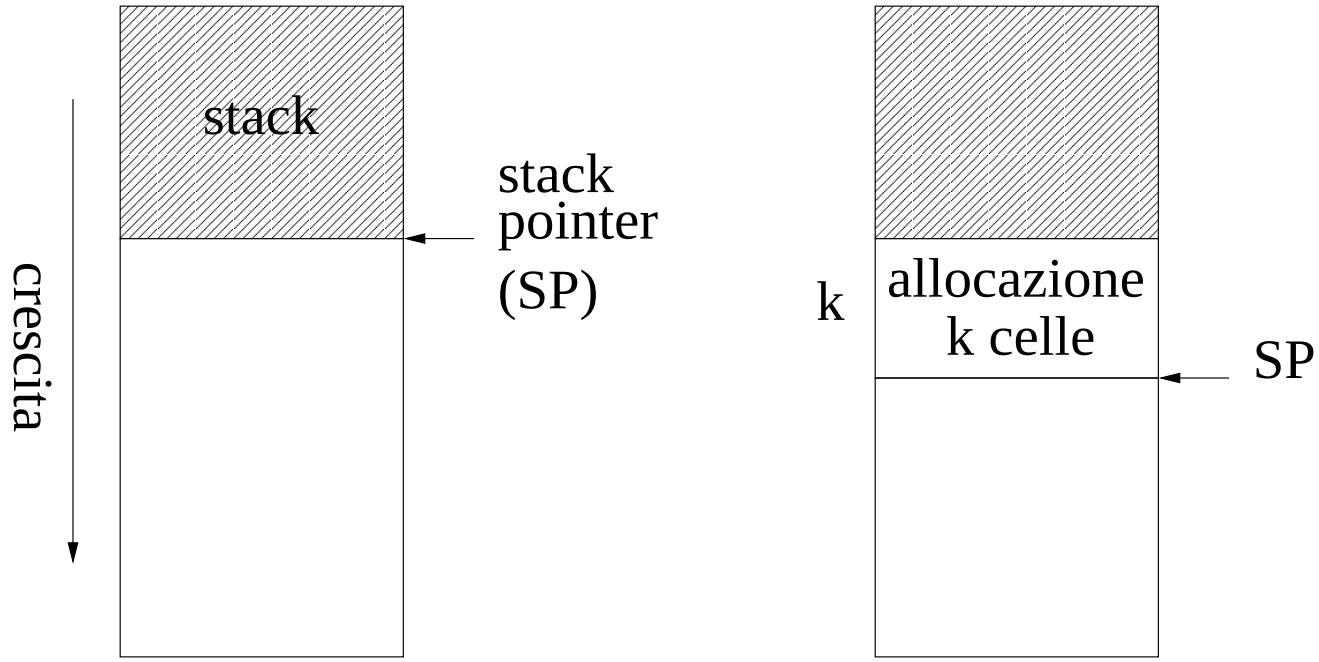
Gestione Statica

- allocazione a complile time
- non occorre gestione memoria a run-time
- nessun problema di recupero memoria
- efficienza a run-time
- impossibilità di avere ricorsione (Fortran e Cobol vecchie versioni; limite superato nel Fortran_{90})
- impossibilità di gestire strutture dati di dimensioni che dipendono dallo stato dell'elaborazione o da qualche valore di input

Tutti i linguaggi moderni hanno una qualche forma di gestione dinamica

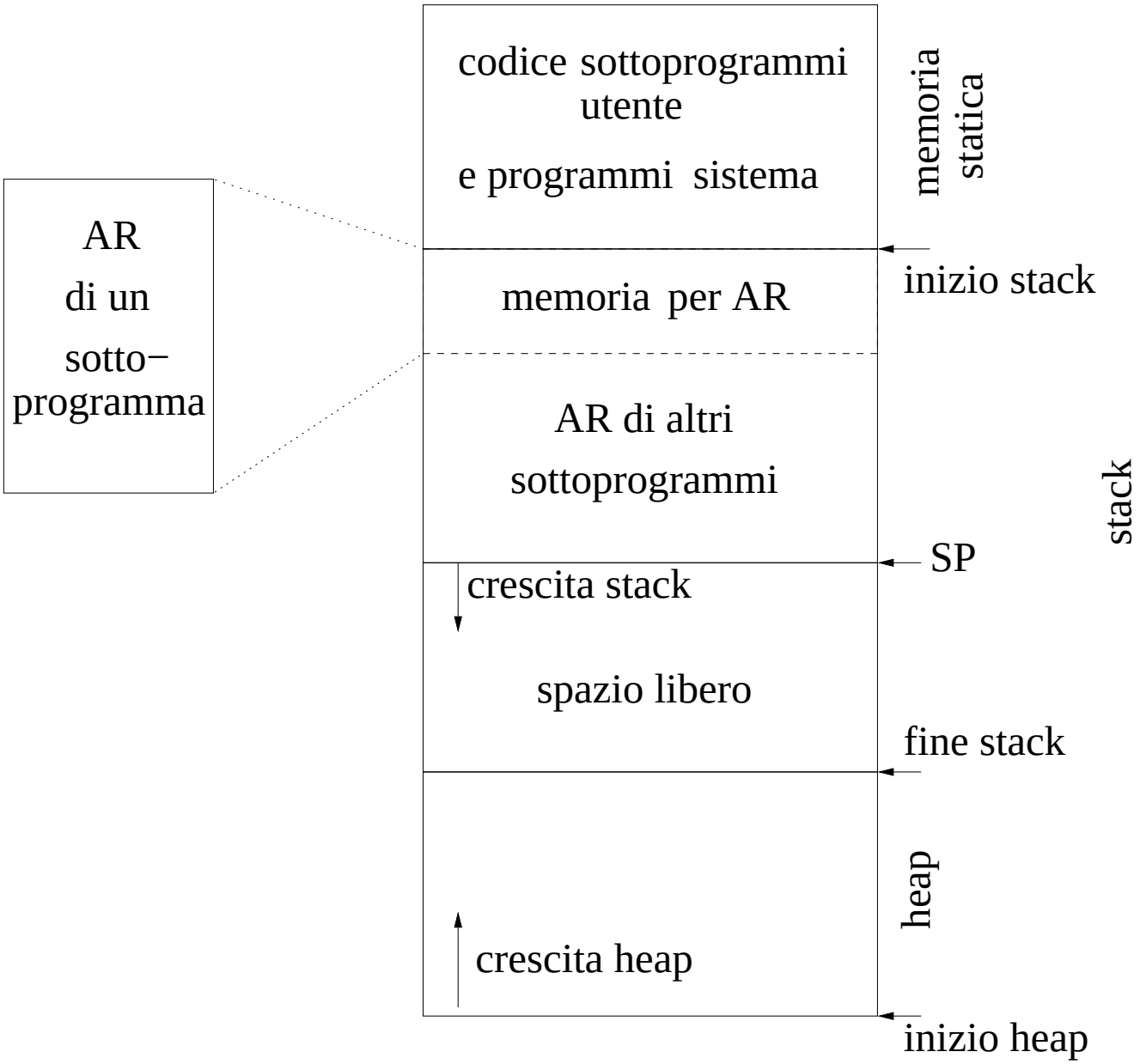
Gestione a Stack (Pila)

È la forma più semplice di gestione run-time della memoria.



- allocazione e deallocazione con semplice spostamento di SP
- tecnica che si addice alla gestione first-in first-out delle chiamate dei sottoprogrammi: applicata alla allocazione/deallocazione dei record di attivazione

Organizzazione della Memoria



Heap: per operazioni dinamiche sui dati quali malloc/free (C), new/dispose (Pascal) o operazioni su liste (Lisp/ML)

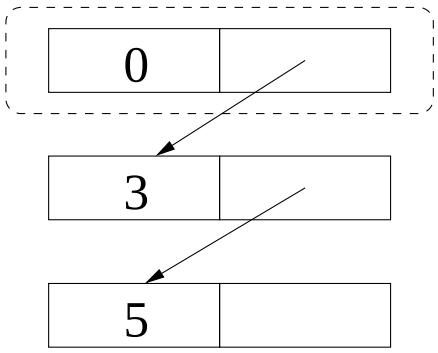
Gestione a Heap

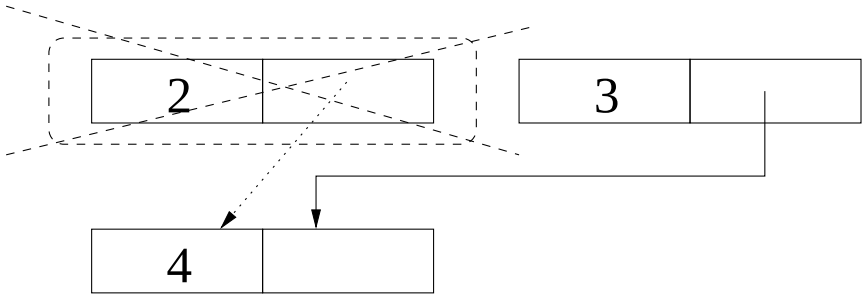
Heap: zona di memoria per i dati dinamici
$\Rightarrow$ gestione a "lista libera"

Esempi di operazioni che interessano l'heap

ML:

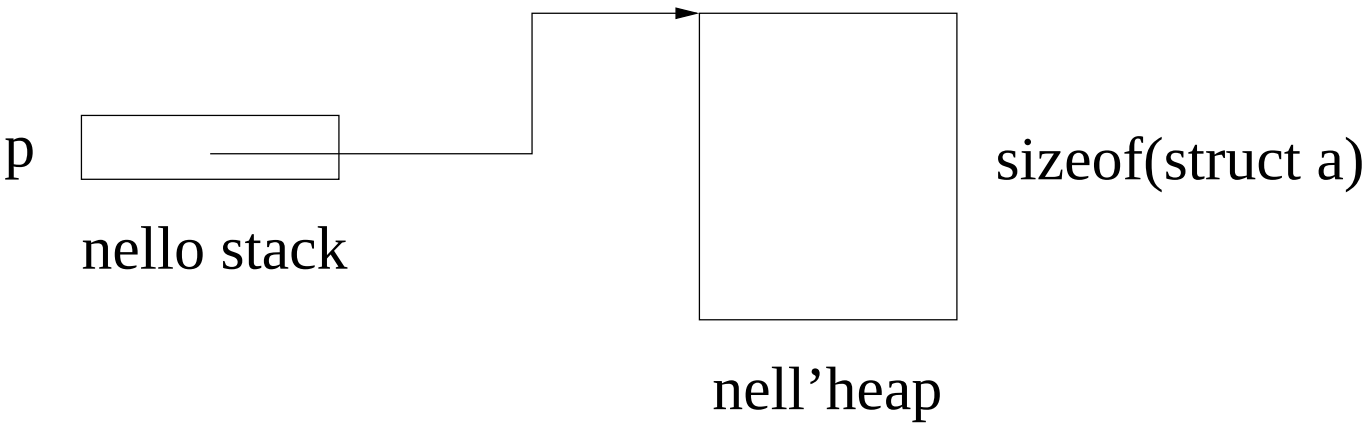
```
fun f []     = []
|   f(x::r) = if x=0 then r
                     else (x+1)::r;
```





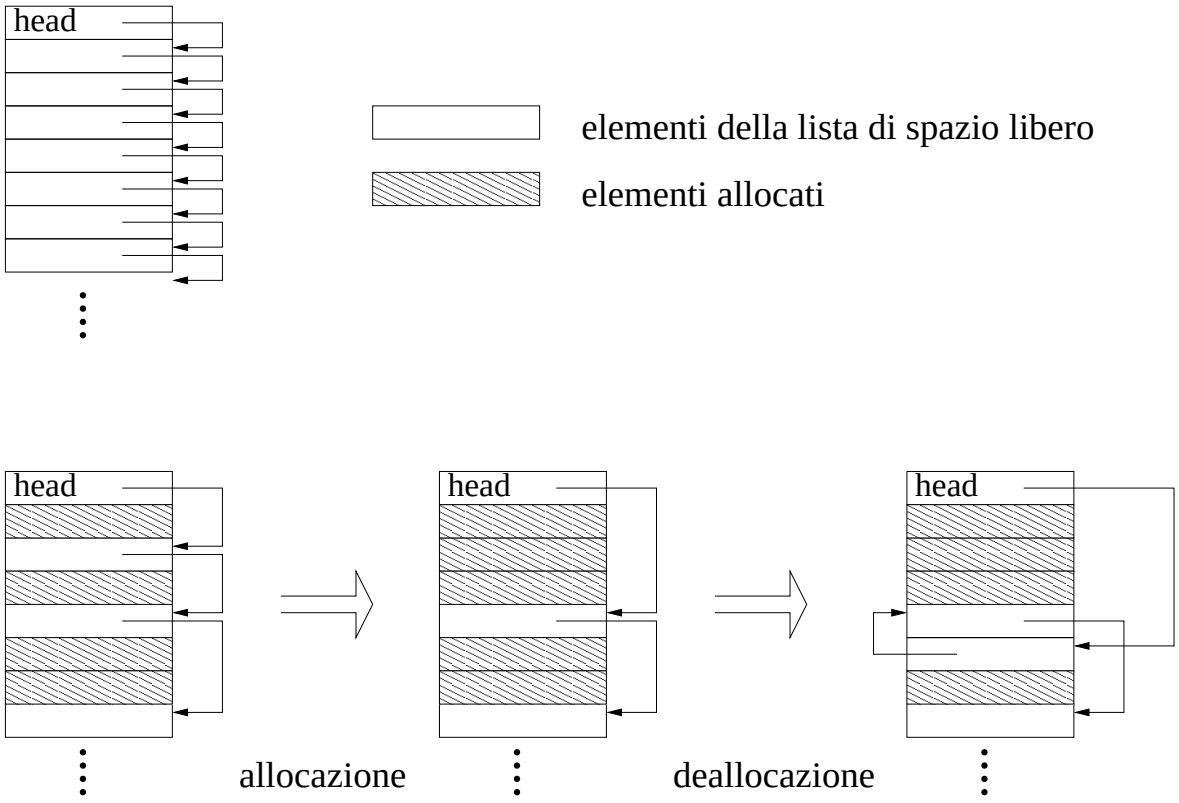
C:

```
struct a { int x;
           int y; };

struct a *p;

p = (struct a*) malloc(sizeof(struct a));
```



Caso 1. Heap con Elementi a Dimensione Fissa

dimensione dei blocchi tipica: 1 o 2 parole

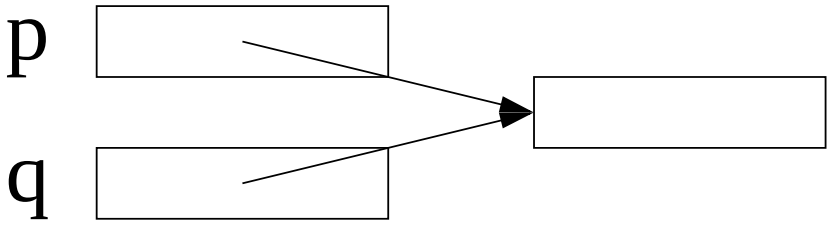


Recupero della Memoria

Problemi

Ad un dato nello heap è associato un cammino di accesso

```
*p = &I;
*q = &I;
```



Garbage: il dato esiste (allocato) ma tutti i cammini di accesso sono distrutti

Dangling References: cammino di accesso continua ad esistere dopo il tempo di vita del dato associato

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

- `p = NULL;`
  $\Rightarrow$ garbage: la mem. dinamica rischia di esaurirsi

- `q = p;`
  `free(p);`
  $\Rightarrow$ dangling-reference: gravi errori in esecuzione

Soluzioni

contatore di riferimenti:
recupero esplicito memoria
$+$
meccanismo conteggio cammini di accesso

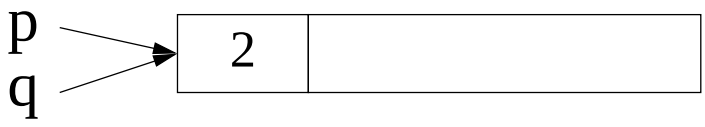
garbage collection:
ammesso garbage ma non dangling-reference
$+$
garbage collector (quando heap esaurito)

Contatore di Riferimenti

K: numero di puntatori all'elemento $\Rightarrow$ recupero memoria solo se K=0

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

q = p

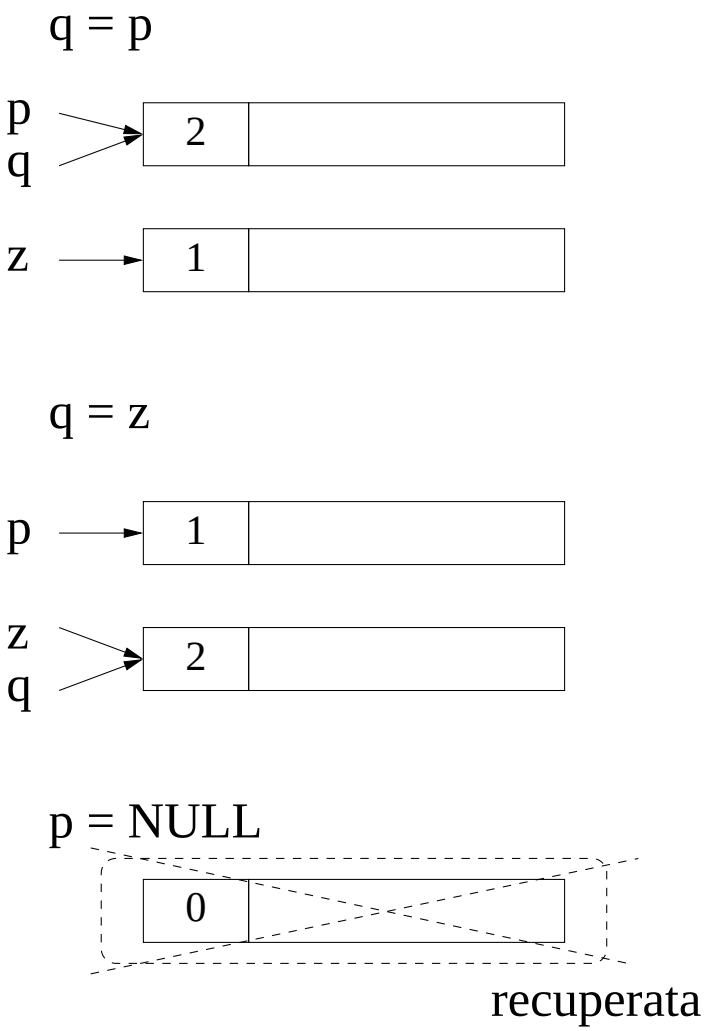


free(p)

Difetto: operazioni costose

`q = p;` $\Rightarrow$ situazione può essere complicata

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



- `free(p)`
  $\Rightarrow$ decremento unitario contatore struttura puntata da `p` con eventuale recupero memoria se contatore = 0

- `p=q`
  (a) decremento unitario contatore struttura puntata da `p` con eventuale recupero memoria se contatore = 0
  (b) incremento unitario contatore struttura puntata da `q`

L'accesso ad una struttura con un puntatore `p` è lecita solo se il contatore della struttura è $\neq 0$

Garbage Collection

- permettere la creazione di garbage al fine di evitare dangling references

- recuperare il garbage solo quando la memoria è esaurita

  1. interruzione computazione programma
  2. controllo a "garbage collector"
  3. ripristino computazione programma

Il garbage collector può essere computazionalmente dispendioso

Il garbage collector opera in due fasi **mark** e **sweep**

Ogni elemento nell'heap ha un bit `M` di marcatura:

- 0/OFF
- 1/ON (marcatura iniziale)

Un elemento è **attivo** se fa parte di una struttura accessibile

1. **mark**: ogni elemento attivo viene marcato OFF
2. **sweep**: tutti gli elementi ON sono restituiti all'heap
3. (tutti gli elementi sono infine rimarcati ON per futuri recuperi)

- **sweep**: semplice scansione lineare dell'heap
- **mark**: difficile!

Cosa significa che un elemento `E` è attivo?

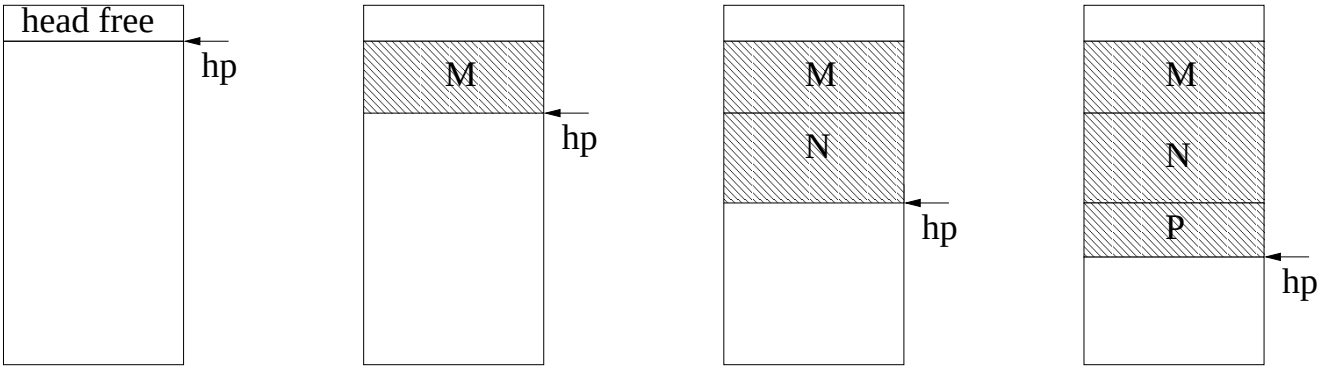
`E` è attivo se

- è puntato da un elemento esterno all'heap
- è puntato da un elemento attivo dell'heap

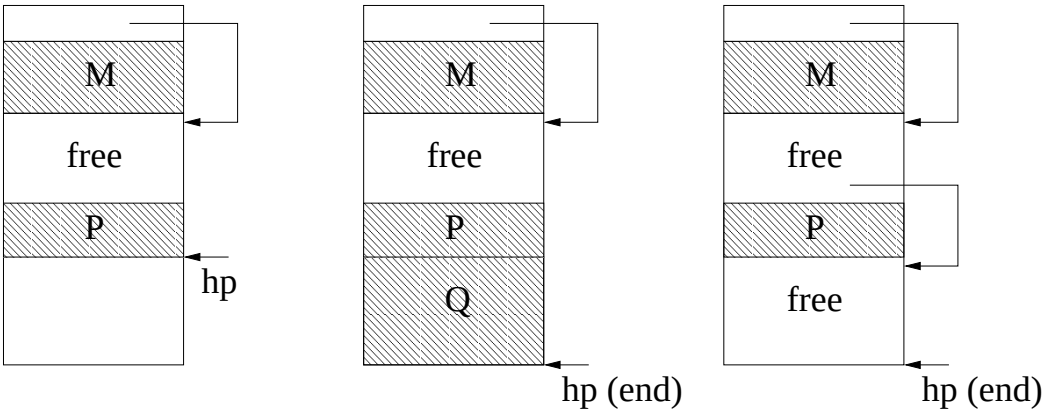
Affinché tale politica di recupero sia possibile devono essere soddisfatte le seguenti condizioni

- ogni elemento attivo deve essere raggiungibile tramite una catena di puntatori che inizia fuori dall'heap
- deve essere possibile identificare tutti i puntatori esterni allo heap che puntano ad elementi dell'heap
- deve essere possibile identificare in un qualsiasi elemento attivo dello heap quei campi che contengono puntatori ad altri elementi dell'heap

Caso 2. Heap con Elementi a Dimensione Variabile



hp: puntatore heap



Come si prosegue?

1. si usa la lista creata dinamicamente

   (a) se serve un blocco lungo m si cerca un blocco B lungo $n \geq m$

   (b) si spezza blocco B (se $n > m$)

   Due politiche per scegliere blocco B:

   - first fit
   - best fit

   $\rightarrow$ problema della frammentazione

2. tecnica di compattazione

   tutto lo spazio libero viene compattato in un unico blocco e spostato in fondo all'heap, con conseguente arretramentro del puntatore all'heap