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

Sistemi di tipi

- Il *sistema di tipi* di un linguaggio:
  - Tipi predefiniti
  - Meccanismi per definire nuovi tipi
  - Meccanismi relativi al controllo:
    - equivalenza
    - compatibilità
    - inferenza
  - Specifica se i controlli sono statici o dinamici

- Un sistema di tipo è sicuro relativamente ai tipi (type safe):
  - in esecuzione non possono avvenire errori non segnalati derivanti da un errore di tipo

Tipi scalari

- Booleani
  - val: true, false
  - op: or, and, not, condizionali
  - repr: un byte
  - note: C non ha un tipo bool
- Caratteri
  - val: a,A,b,B, …, è,é,ë, ; , ‘, …
  - op: uguaglianza; code/decode; dipendenti dal linguaggio
  - repr: un byte (ASCII) o due byte (UNICODE)

Tipi scalari

- Interi
  - val: 0,1,-1,2,-2,…,maxint
  - op: +, -, *, mod, div, …
  - repr: alcuni byte (2 o 4); complemento a due
  - note: interi e interi lunghi (anche 8 byte); limitati problemi nella portabilità quando la lunghezza non è specificata nella definizione del linguaggio

- Reali
  - val: valori razionali in un certo intervallo
  - op: +, -, *, /, …
  - repr: alcuni byte (4); virgola mobile
  - note: reali e reali lunghi (8 byte); gravi problemi di portabilità quando la lunghezza non è specificata nella definizione del linguaggio

Tipi scalari

- Complessi
  - val: ....
  - op: …
  - repr: due reali
  - note: Scheme, Ada

- Fixed point
  - val: valori razionali in un certo intervallo
  - op: +, -, *, /, …
  - repr: alcuni byte (2 o 4); complemento a due, virgola fissa
  - note: Ada.

Fixed-point rappresentano in modo compatto un ampio intervallo con pochi decimali di precisione

Tipi Scalari

- Il tipo `void`
  - ha un solo valore
  - nessuna operazione
  - serve per definire il tipo di operazioni che modificano lo stato senza restituire alcun valore

```
void f (…) {…}
```

  - il tipo di f deve avere un valore (e non nessuno) altrimenti non potremmo definire una tale f !
  - il valore restiuito da f di tipo `void` è sempre il solito (e dunque non interessa)

Una prima classificazione

- Tipi ordinali (o discreti):
  - booleani, interi, caratteri
  - ogni elemento possiede un succ e un prec (eccetto primo/ ultimo)
  - altri tipi ordinali:
    - enumerazioni
    - intervalli (subrange)
  - vi si può “iterare sopra”
  - indici di array
- Tipi scalari
  - tutti quelli che abbiam visto sin qui:
    - ordinali
    - reali, complessi
  - hanno una “diretta” rappresentazione nell’implementazione
  - non sono costituiti dall’aggregazione di altri valori

Enumerazioni

- introdotti in Pascal
  ```
  type Giorni = (Lun,Mar,Mer,Giov,Ven,Sab,Dom);
  ```
- programmi di più semplice comprensione
- valori ordinati: `Mar < Ven`
- iterazione sui valori: `for i:= Lun to Sab do …`
- succ, pred
- rappresentati come short integer (un byte)

- In C:
  ```
  enum giorni = {Lun,Mar,Mer,Giov,Ven,Sab,Dom};
  ```
  ma è equivalente a
  ```
  typedef int giorni;
  const giorni Lun=0, Mar=1, Mer=2….,Dom=6;
  ```
- contra: Pascal distingue `Mar` e `1`

Intervalli (subrange)

- introdotti in Pascal
- i valori sono un intervallo dei valori di un tipo ordinale (il *tipo base* dell'intervallo)
- Esempî:

```
type MenoDiDieci = 0..9;
type GiorniLav = Lun..Ven;
```

- rappresentati come il tipo base
- perché usare un tipo intervallo invece del suo tipo base:
  - documentazione "controllabile"
  - generazione di codice efficiente

Tipi composti, o strutturati, o non scalari

- Record
  - collezione di campi (fields), ciascuno di un (diverso) tipo
  - un campo è selezionato col suo nome
- Record varianti
  - record dove solo alcuni campi (mutuamente esclusivi) sono attivi ad un dato istante
- Array
  - funzione da un tipo indice (scalare) ad un altro tipo
  - array di caratteri sono chiamati stringhe; operazioni speciali
- Insieme
  - sottinsieme di un tipo base
- Puntatore
  - riferimento (*reference*) ad un oggetto di un altro tipo

Records

- manipolare in modo unitario dati di tipo eterogeneo
- C, C++, CommonLisp, Algol68: `struct` (strutture)
- Java: non ha tipi record, sussunti dalle classi
- Esempio, in C:

```
struct studente {
    char nome[20];
    int matricola; };
```

- Selezione di campo:

```
studente s;
s.matricola=343536;
```

- record possono essere annidati
- memorizzabili, esprimibili e denotabili
  - Pascal non ha modo di esprimere “un valore record costante”
  - C lo può fare, ma solo nell’inizializzazione (initializer)
  - uguaglianza generalmente non definita (contra: Ada)

Records: memory layout

- memorizzazione sequenziale dei campi
- allineamento alla parola (16/32 bit)
  - spreco di memoria

- packed records
  - disallineamento
  - accesso più costoso

Record varianti

- In un record variante alcuni campi sono alternativi tra loro: solo uno di essi è attivo in un dato istante

```
type stud = record
       nome : packed array [1..6] of char;
       matricola: integer;
       case fuoricorso : Boolean of
            true: (ultimoanno: 2000..maxint);
            false:(anno:(primo,secondo,terzo);
                   inpari:Boolean;)
       end;
var s: stud;
…
s.fuoricorso := true;
s.ultimoanno:= 2001;
…
```

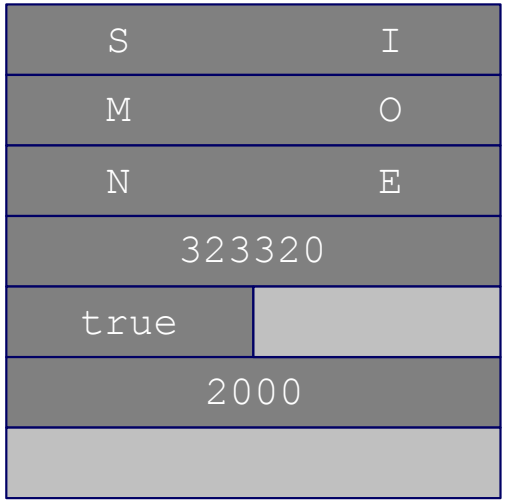
I due campi (le due varianti) `ultimoanno` e `anno` possono condividere la stessa locazione di memoria

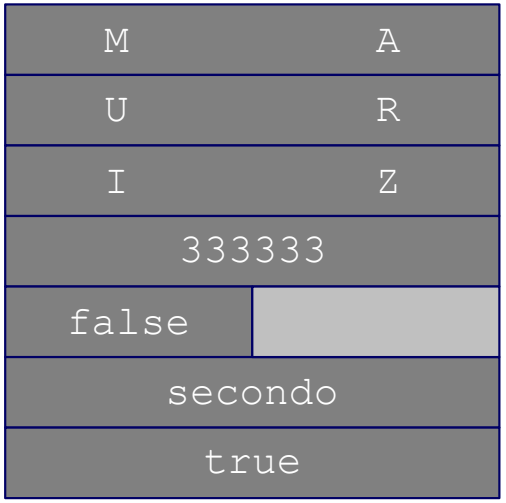
Il tipo del tag `fuoricorso` può essere un qualsiasi tipo ordinale

Record varianti: memory layout

```
type stud = record
 nome : packed array [1..6] of char;
 matricola: integer;
 case fuoricorso : Boolean of
      true: (ultimoanno: 2000..maxint);
      false:(anno:(primo,secondo,terzo);
             inpari:Boolean;)
 end;
```

Due variabili di tipo `stud`:





Record varianti

- Possibili in molti linguaggi
  - C: union + struct

```
struct student {char nome[6];
    int matricola;
    bool fuoricorso;
    union {
            int ultimoanno;
            struct {   int anno;
               bool inpari;} campivarianti;
               }
    };
```

- Pascal (Modula, Ada) unisce unioni e record con eleganza

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

Array: operazioni

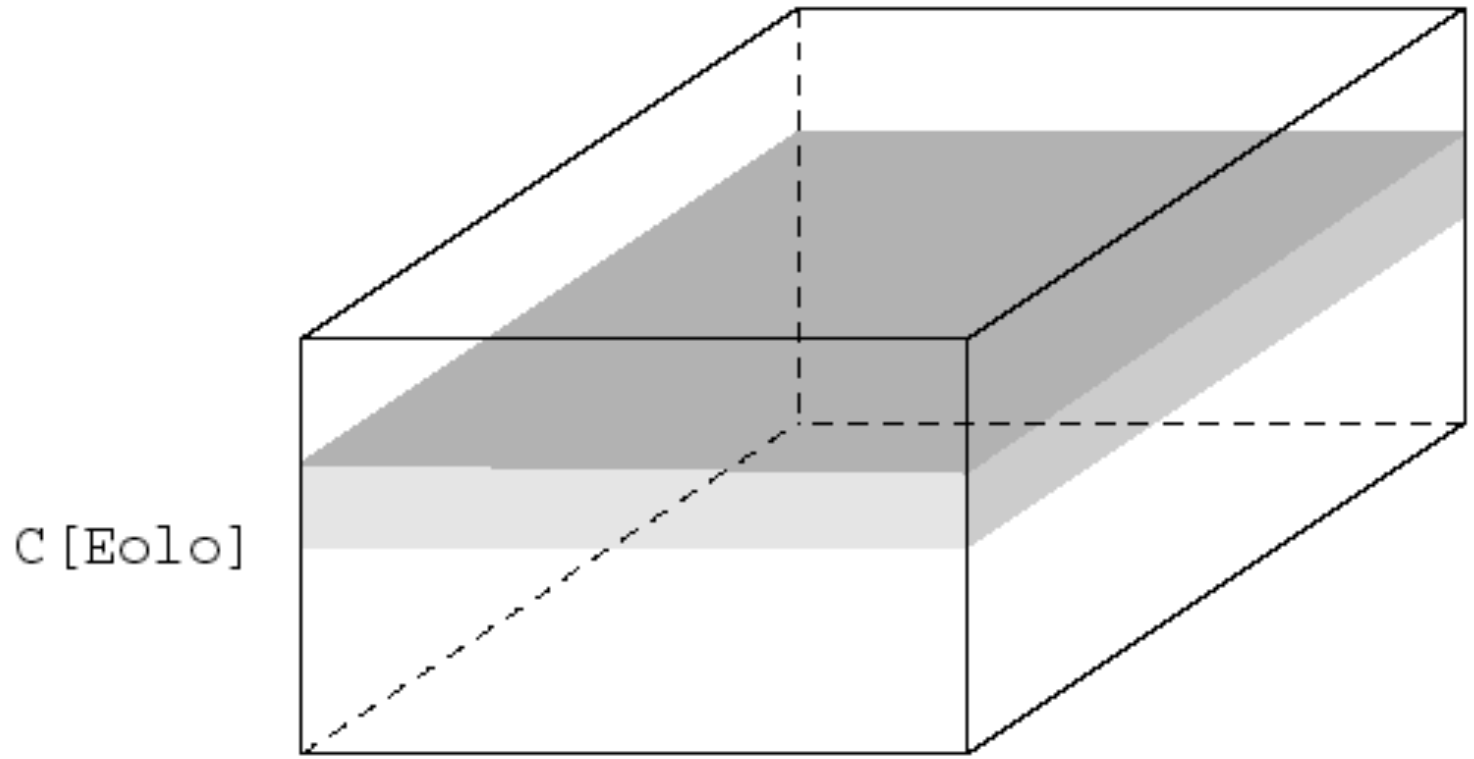
- Principale operazione permessa:
  - selezione di un elemento: vet[3] mat[10,'c']
  - attenzione: modifica non è un'operazione sull'array, ma sulla locazione modificabile che memorizza un (elemento di) array

- Alcuni linguaggi permettono *slicing*:
  - selezione di parti contigue di un array
  - Esempio: in Ada, con
    `mat : array(1..10) of array (1..10) of real;`
    `mat(3)` indica la terza riga della matrice quadrata `mat`

- In presenza di slicing, presenti spesso anche operazoni globali su array:
  - Fortran90: A+B somma gli elementi di A e B (dello stesso tipo)
  - APL: linguaggio (~1962) per manipolazioni di array
    - potenti operazioni;linguaggio sintetico e austero
    - lessico "matematico": lettere greche
    - impossibile da leggere (e da scrivere sulle telescriventi dell'epoca)

Slice su array: esempio

```
int W[21..30];
type Nano = {Brontolo, Cucciolo, Dotto, Eolo, Gongolo, Mammolo,
             Pisolo};
int V[1..10,1..10];
char C[Dotto..Mammolo,0..10,1..10];
```



Memorizzazione degli array

- Elementi memorizzati in locazioni contigue:
  - ordine di riga: V[1,1];V[1,2];…;V[1,10];V[2,1];…
    - 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]; …;V[10,1];V[1,2];…
- 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:
    α 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+ α
  - se l'indice inizia sempre da zero (l1=l2=l3=0) nessuna precomputazione

Array: shape

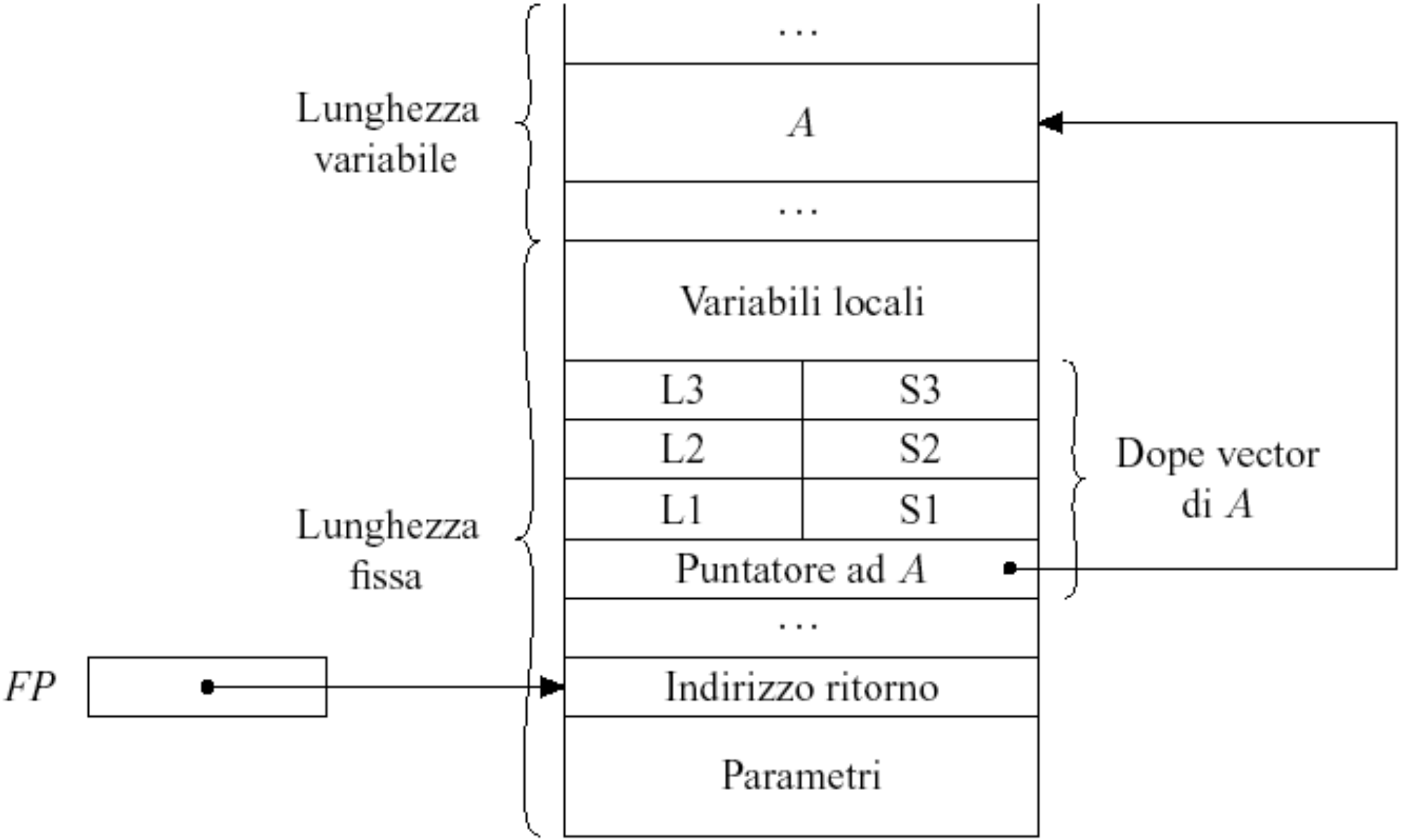
- Forma (o shape): numero delle dimensioni e intervallo dell'indice. Quando è fissata ?
  - Forma statica.
  - Forma fissata al momento dell'elaborazione della dichiarazione.
  - Forma dinamica.

Array: dope vector

- Info sulla forma dell'array è mantenuta dal compilatore
- Se la forma varia a run time, info mantenuta in un descrittore dell'array detto dope vector dell'array che contiene:
  - puntatore all'inizio dell'array (nella parte variabile)
  - numero dimensioni
  - limite inferiore (se a run-time vogliamo controllare anche l'out-of-bound anche limite superiore)
  - occupazione per ogni dimensione (valore Si)
- Il dope vector è memorizzato nella parte fissa del RdA
- L'accesso è effettuato calcolando tutto l'indirizzo a run-time, usando il dope vector

Esempio: RdA con dope vector

A : array[l1..u1, l2..u2, l3..u3] of elem_type



Puntatori

- In linguaggi con variabili modificabili permettono di riferirisi (e modificare) un l-valore senza dereferenziarlo

- Valori : riferimenti (l-valori); costante `null` (o `nil`)
- Operazioni:
  - creazione
    - funzioni di libreria che alloca e restituisce un puntatore (eg, `malloc`)
  - dereferenziazione
    - accesso al dato “puntato”: `*p`
  - test di uguaglianza
    - in specie test di uguaglianza con `null`

Puntatori e array in C

- Array e puntatori sono intercambiabili in C (!!)

```
int n;
int *a;       // puntatore a interi
int b[10];    // array di 10 interi
…
a = b;         // a punta all'elemento iniziale di b
n = a[3];      // n ha il valore del terzo elemento di b
n = *(a+3);    // idem
n = b[3];      // idem
n = *(b+3);    // idem
```

- Ma ricorda: `a[3]=a[3]+1;` modificherà anche `b[3]` (è la stessa cosa!).
- Array multidimensionali:
  - `b[i][j]` è lo stesso di
    - `(*(b+i))[j]`
    - `*(b[i]+j)`
    - `*(*(b+i)+j)`

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 occore 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

Polimorfismo

- Uno stesso valore ha più tipi
- Distinguiamo tre forme di polimorfismo (Strachey):
  - polimorfismo ad hoc, o *overloading*
  - polimorfismo universale:
    - polimorfismo parametrico (esplicito e implicito)
    - polimorfismo di sottotipo

Polimorfismo ad hoc: overloading

- Uno stesso simbolo denota significati diversi:
  - `3 + 5`
  - `4.5 + 5.3`
- Il compilatore traduce + in modi diversi
- Sempre risolto a tempo di compilazione
  - dopo l'inferenza dei tipi

Polimorfismo parametrico

- Un valore ha polimorfismo universale parametrico quando ha un'infinità di tipi diversi, che si ottengono per istanziazione da un unico schema di tipo generale.

- Una funzione polimorfa è costituita da un unico codice che si applica uniformemente a tutte le istanze del suo tipo generale

```
Ide(x) = x;
sort(v) = ….;
```

Ide ha tipo <T> -> <T>

Ide ha tipo <T>[ ] -> void ossia ∀ T. <T>[ ] -> void

T è una variabile di tipo

Polimorfismo parametrico esplicito

- In C++: function template
  - una funzione swap che scambia due interi

```
void swap (int& x, int& y){
int tmp = x; x=y; y=tmp;}
```

  - una template swap che scambia due dati qualunque

```
template <typename T>        //T è una sorta di
  parametro
void swap (T& x, T& y){
T tmp = x; x=y; y=tmp;}
```

  - istanziazione automatica

```
int i,j;        swap(i,j); //T diventa int a link-time
float r,s;      swap(r,s); //T diventa float a link time
String v,w;     swap(v,w); //T diventa String a link
  time
```

Polimorfismo parametrico in ML (implicito)

- La funzione swap in ML:

```
- swap(x,y) = let val tmp = !x in
                          x = !y; y = tmp end;
val swap = fn : 'a ref * 'a ref -> unit
```

- ``ML inserisce TEMPLATE automaticamente ogni volta che può”

- Istanziazione a tempo di compilazione

```
- swap(x,y);     //x e y sono variabili intere (int ref)
val it = (): unit
- swap(v,w);     //r e s sono variabili string (string ref)
val it = (): unit
```

Polimorfismo di sottotipo

- Analogo a quello esplicito, ma non tutti i tipi possono essere usati per istanziare il tipo generale

- Supponiamo data una relazione di sottotipo: T < S significa T sottotipo di S

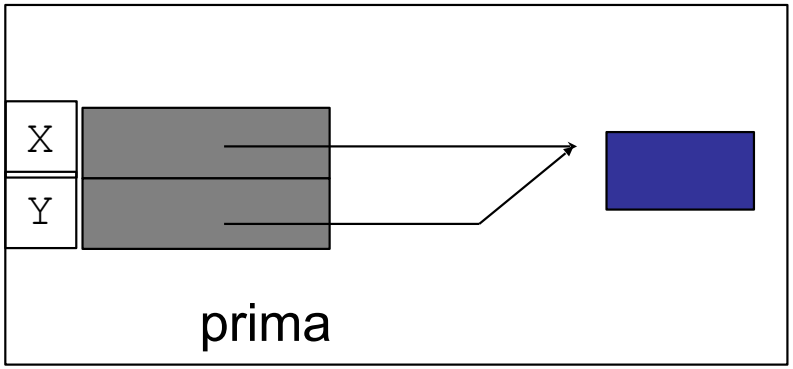
- Def. Un valore esibisce polimorfismo di sottotipo (o limitato) quando ha un'infinità di tipi diversi, che si ottengono per istanziazione da uno schema di tipo generale, sostituendo ad un opportuno parametro i sottotipi di un tipo assegnato.

- Una funzione polimorfa è costituita da un unico codice che si applica uniformemente a tutte le istanze ``legali'' del suo tipo generale, ossia

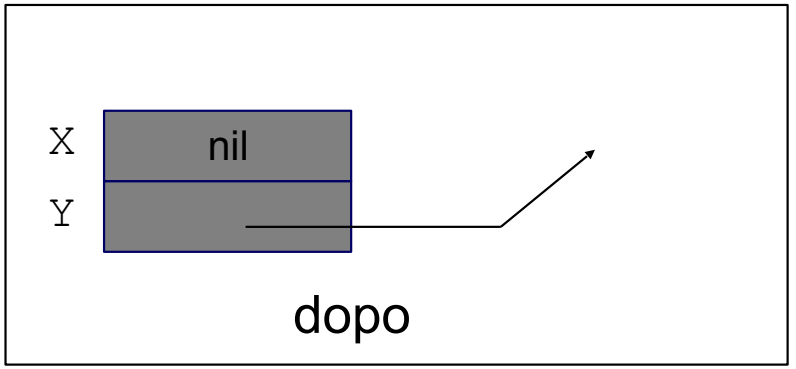
$$\forall\ T < D.\ \langle T \rangle[\ ] \rightarrow \text{void}$$

Tecniche per controllo dei dangling references

- Una dangling reference (puntatore pendente) è un puntatore che punta ad un oggetto non più valido

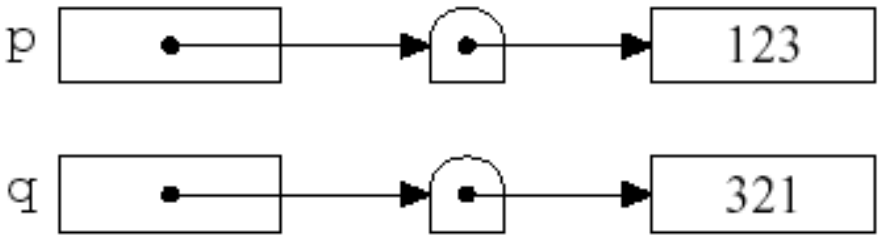



```
free(X);
```

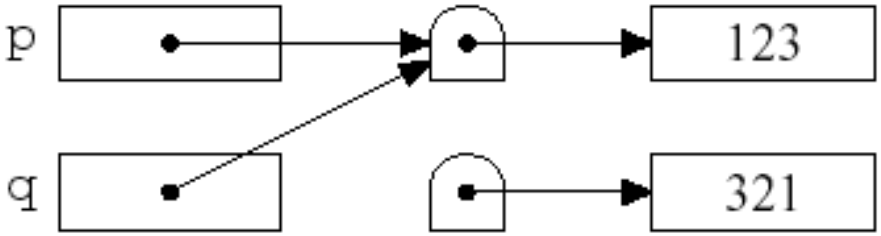


Tombstones

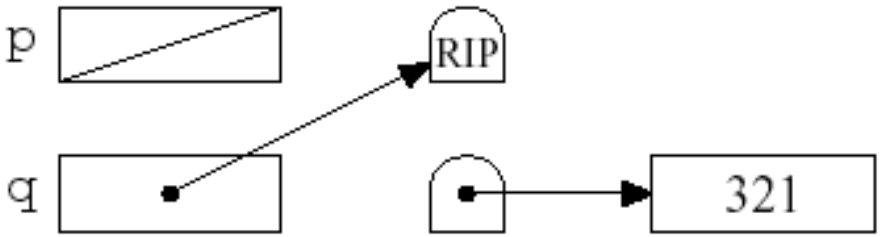
```
p=malloc();
q=malloc();
*p=123;
*q=321;
```




```
q=p;
```

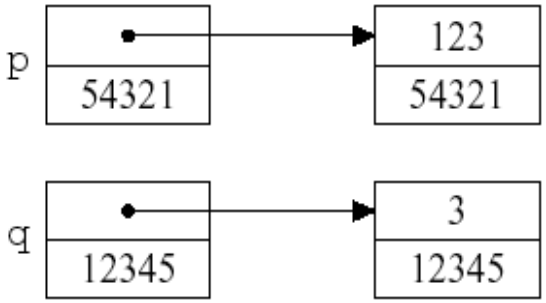



```
free(p);
```

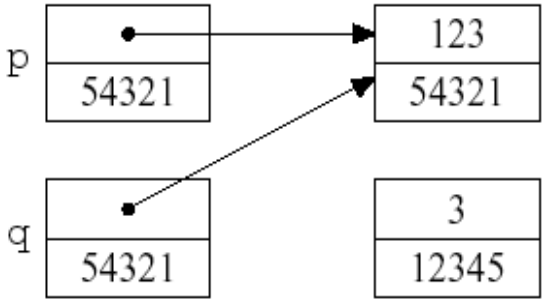


Locks and keys

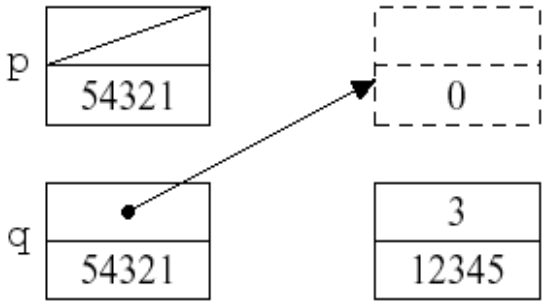
```
p=malloc();
q=malloc();
*p=123;
*q=3;
```




```
q=p;
```



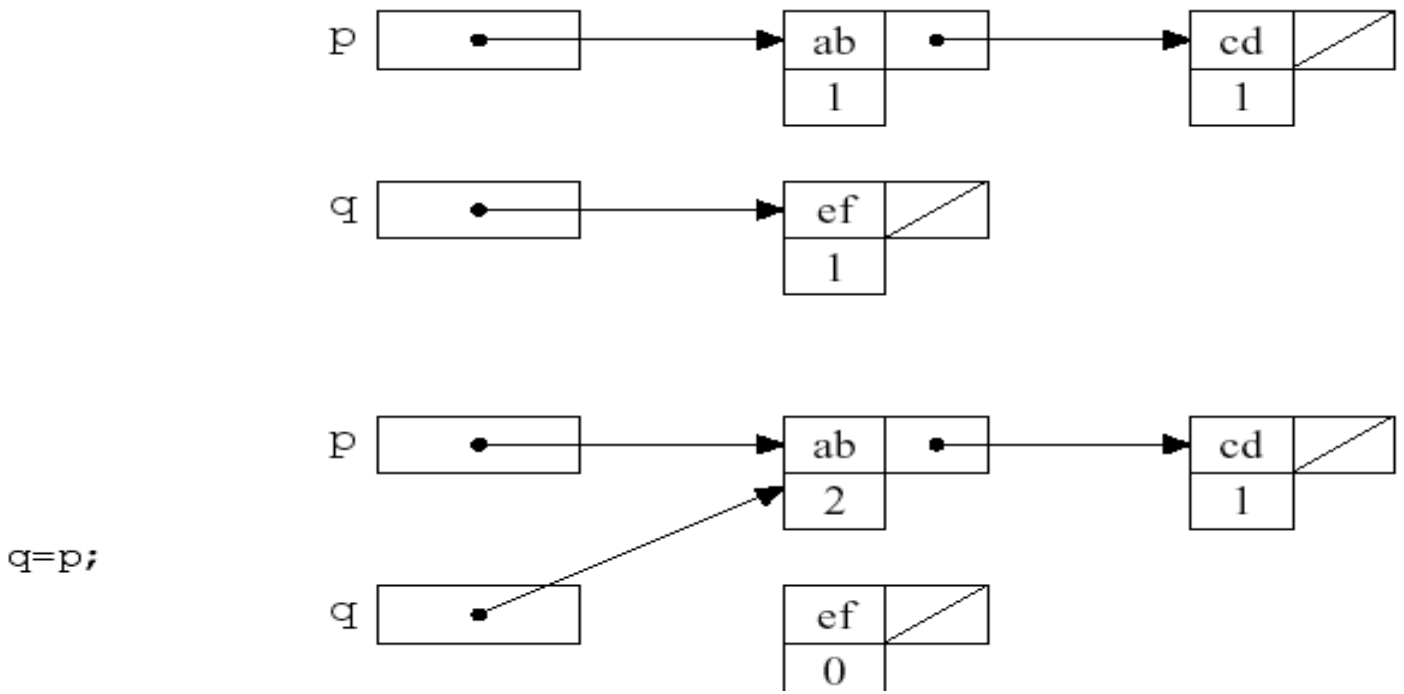

```
free(p);
```



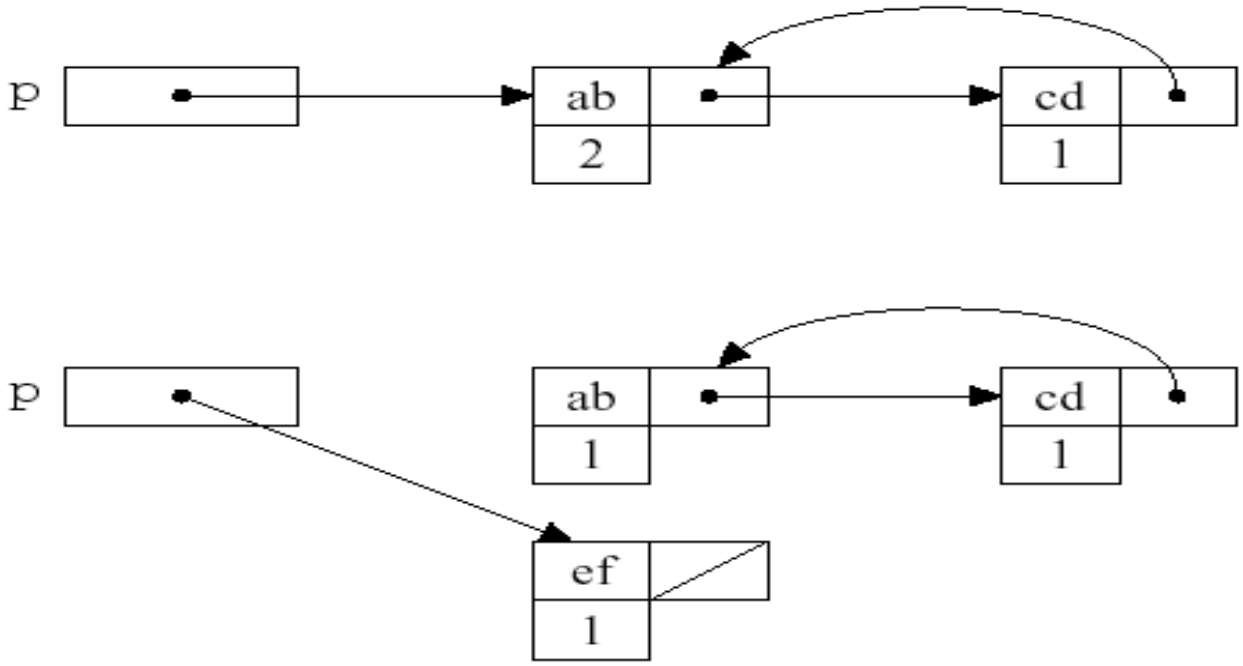
Garbage collection

- 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: Reference count



Problema: Strutture circolari



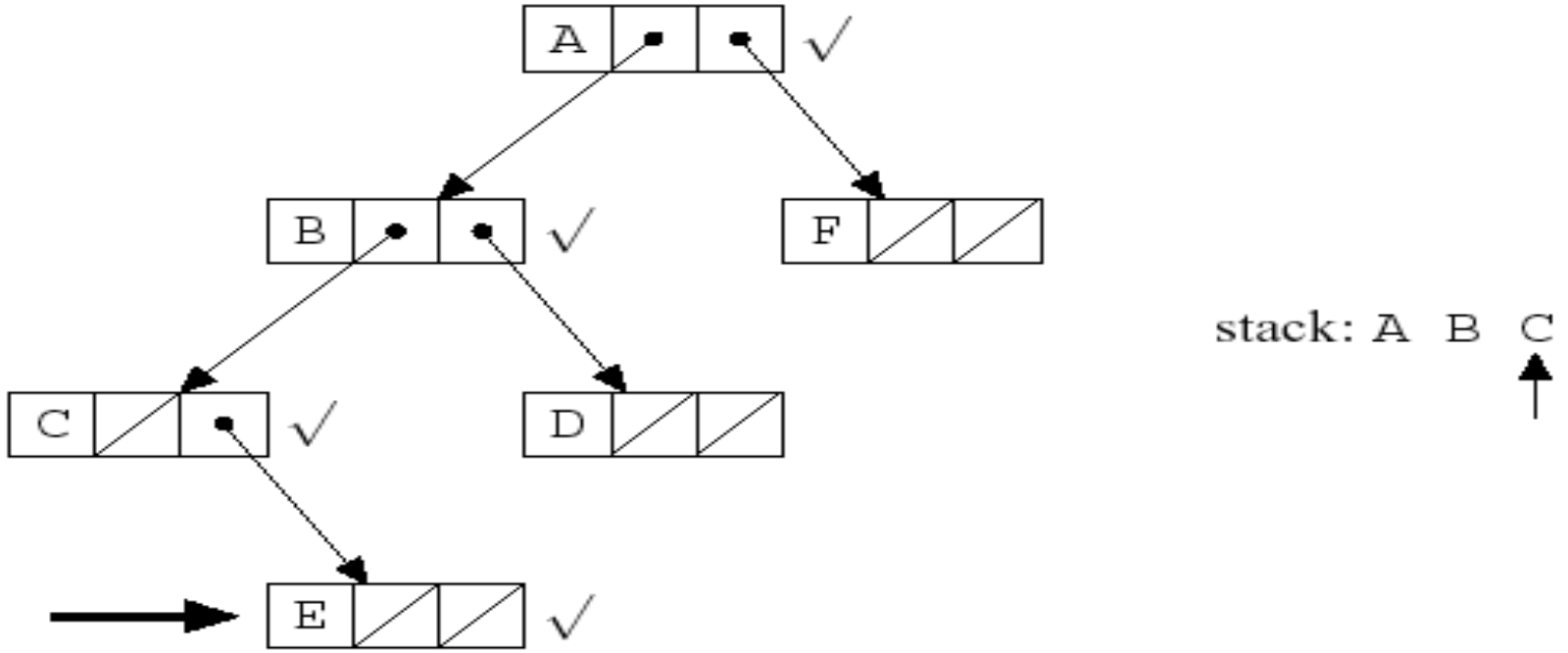

```
p=new();
...
```

Garbage collection: mark and sweep

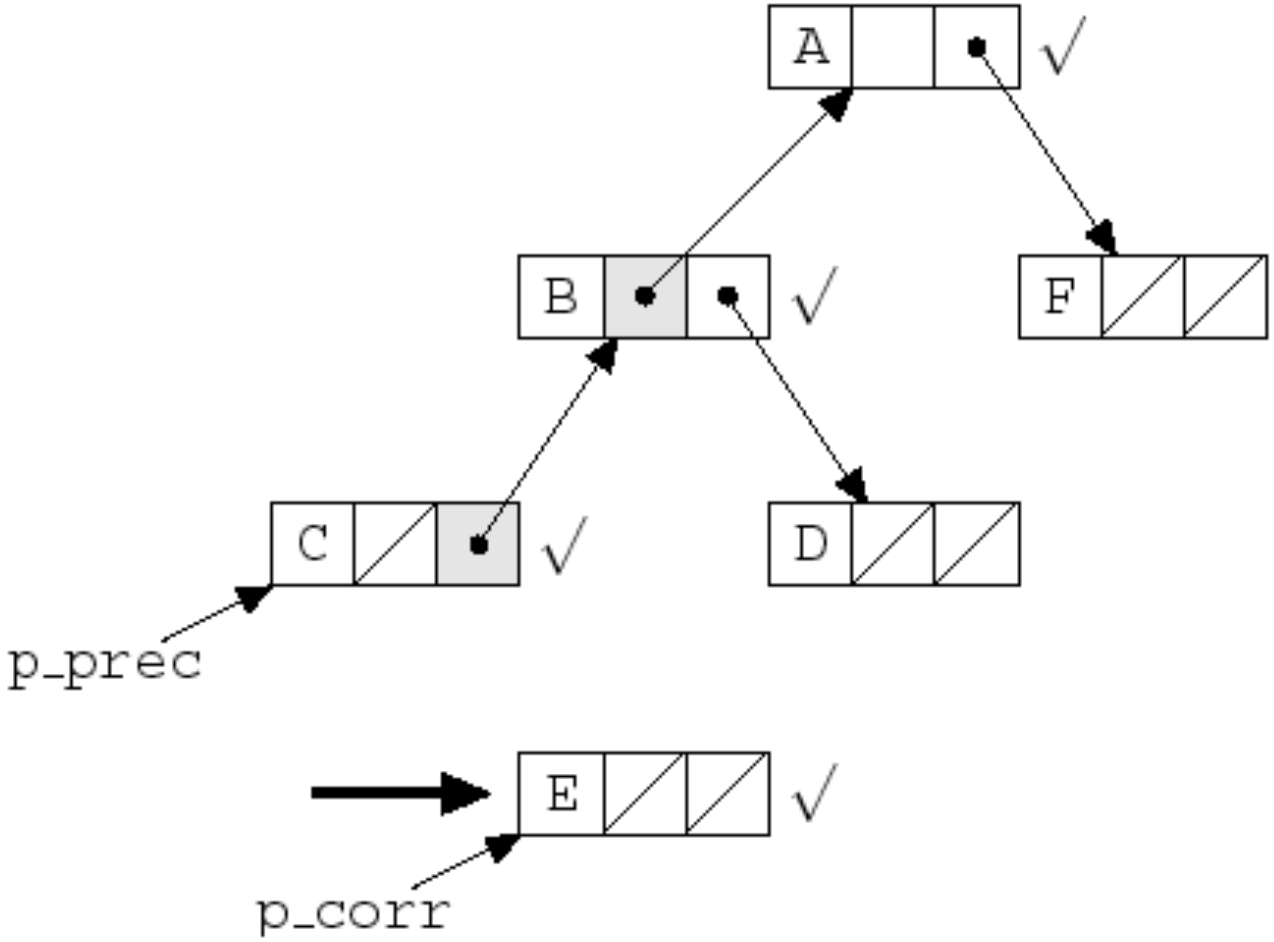
1. Marca tutti gli oggetti sullo heap come unused
2. Partendo dai puntatori fuori dello heap, visita tutte le strutture concatenate, marcando ogni oggetto come
3. Recupera dallo heap tutti gli oggetti rimasti unused

- 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)

Pila per la visita in profondità



Pointer reversal



Algoritmo di Cheney