Memoria Dinamica

Informatica A AA 2024 / 2025

Giacomo Boracchi

20 Novembre 2024

giacomo.boracchi@polimi.it

Slide credits Prof. Alessandro Campi

Durata delle variabili (tempo di vita)

- Va dalla creazione (allocazione della memoria) alla distruzione (rilascio della memoria allocata)
- Due classi di variabili
  - **Automatiche**:
    - Sono quelle dichiarate nelle funzioni (inclusi i parametri) e nei blocchi
    - Sono **create** quando il flusso di esecuzione "entra" nel loro **ambito di visibilità**
    - Sono **distrutte** all'uscita da tale ambito
    - Sono allocate di volta in volta in celle differenti
    - Non conservano i valori prodotti da precedenti esecuzioni della funzione o del blocco

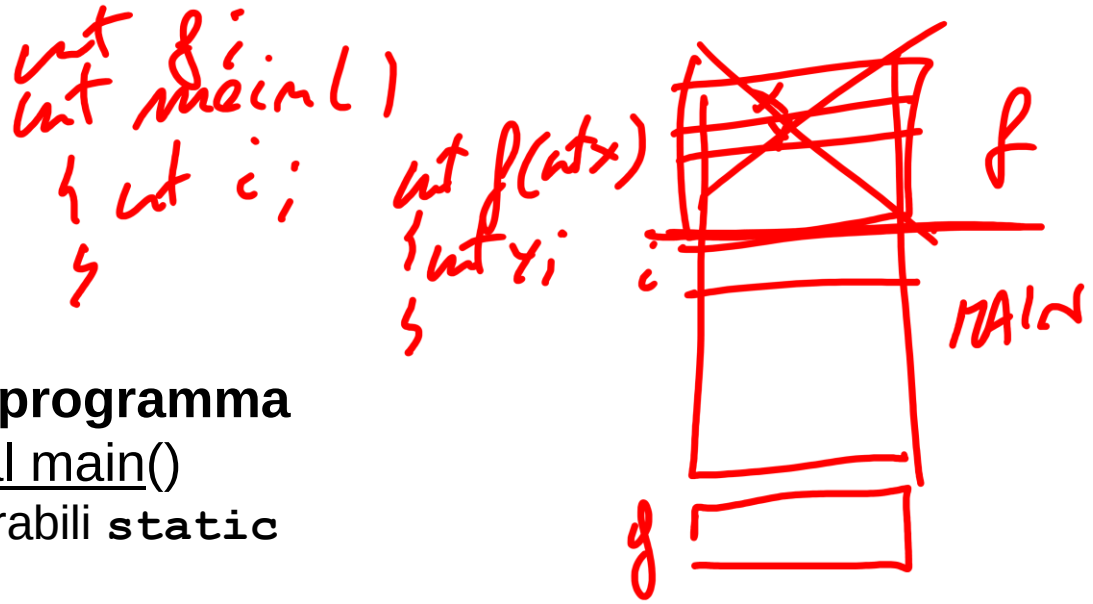
Durata delle variabili (tempo di vita)

- Va dalla creazione (allocazione della memoria) alla distruzione (rilascio della memoria allocata)
- Due classi di variabili
  - **Statiche**:
    - Allocate una volta per tutte
    - Distrutte solo al termine dell'esecuzione del **programma**
    - Sono tutte le variabili globali e quelle locali al main()
      - Anche le variabili di funzione o blocco dichiarabili **`static`**

      (hanno però visibilità locale non come global)
    - Possono fungere da canale di comunicazione tra funzioni (ma è sconsigliato!)

Durata delle variabili (tempo di vita)

- Va dalla creazione (allocazione della memoria) alla distruzione (rilascio della memoria allocata)
- Due classi di variabili
  - **Statiche**:
    - Allocate una volta per tutte
    - Distrutte solo al termine dell'esecuzione del **programma**
    - Sono tutte le variabili globali e quelle locali al main()
      - Anche le variabili di funzione o blocco dichiarabili **`static`** (hanno però visibilità locale non come global)
    - Possono fungere da canale di comunicazione tra funzioni (ma è sconsigliato!)



Un intermezzo: con i puntatori…

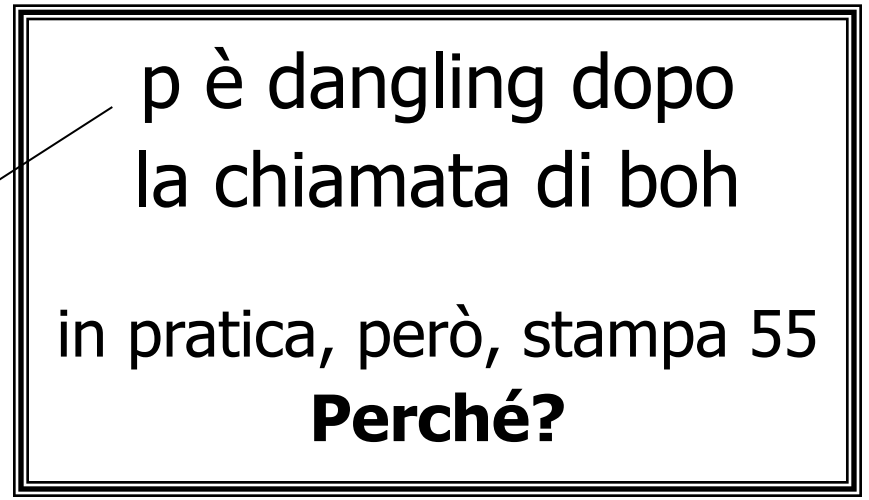
- **…è possibile programmare molto male**
  - in modo "criptico"
  - generando effetti difficili da "tracciare"
  - in modo che il funzionamento del programma dipenda da come uno specifico sistema gestisce la memoria
    - Lo stesso programma, se scritto "male", può funzionare in modo diverso su macchine diverse
- Si possono fare danni considerevoli
  - Non sempre la macchina reale si comporta come il modello suggerirebbe
- Vediamo due "esempi" di cosa "si riesce" a fare..

Puntatori Globali a Variabili Automatiche

```
#include <stdio.h>

int * p;

void boh() {
   int x = 55;
   p = &x;
}

int main() {
   int x = 1;
   boh();
   printf("risultato= %d", *p);
   return 0;
}
```

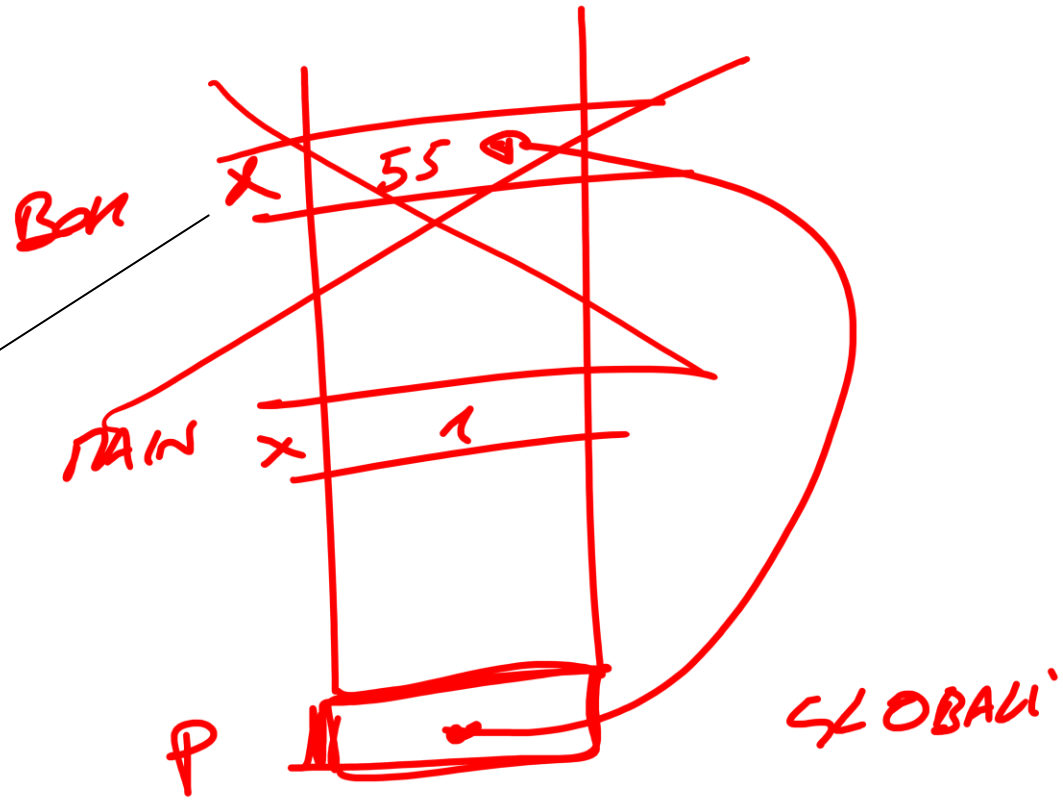
Puntatori Globali a Variabili Automatiche

```
#include <stdio.h>

int * p;

void boh() {
   int x = 55;
   p = &x;
}

int main() {
   int x = 1;
   boh();
   printf("risultato= %d", *p);
   return 0;
}
```



Puntatori Globali a Variabili Automatiche

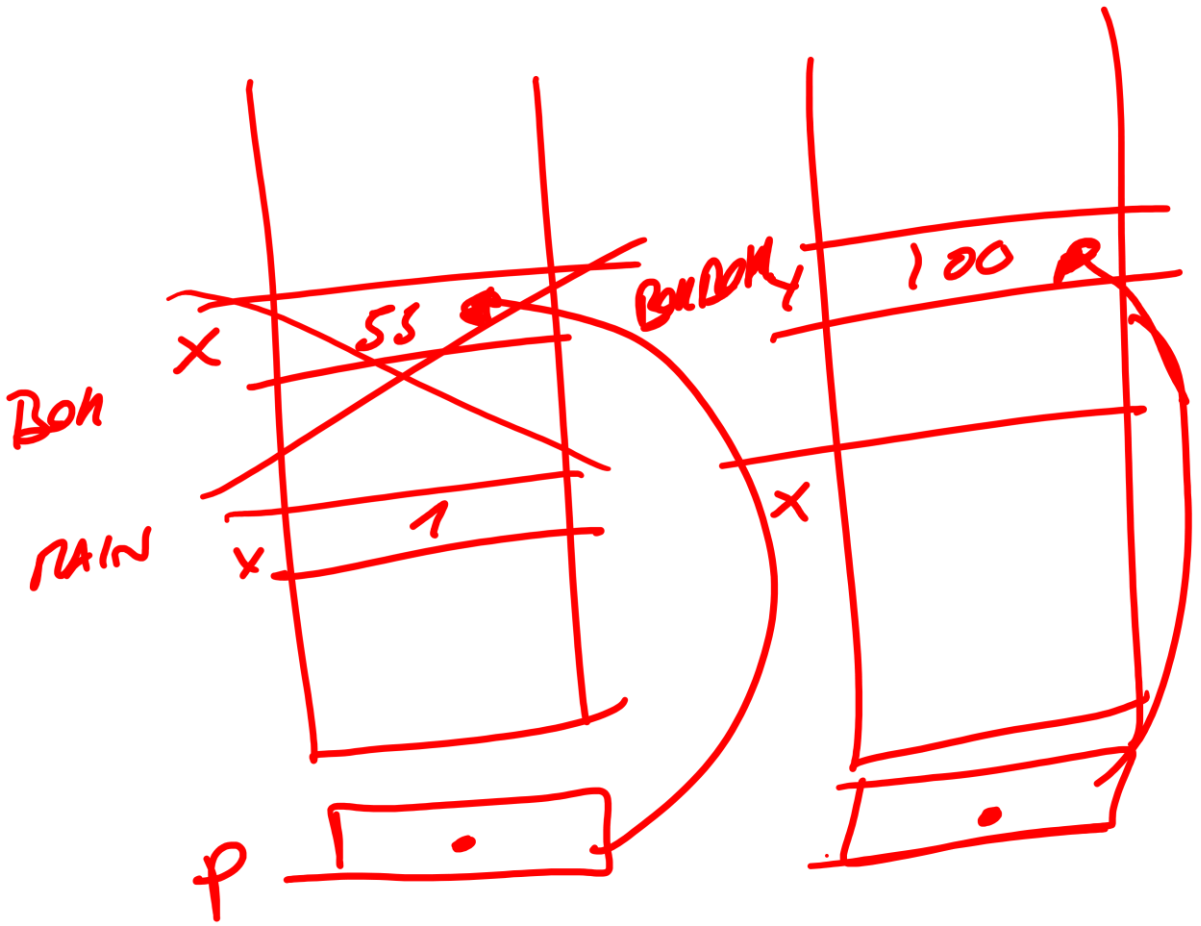
```
#include <stdio.h>

int * p;

void boh() {
   int x = 55;
   p = &x;
}

int main() {
   int x = 1;
   boh();
   printf("risultato= %d", *p);
   return 0;
}
```



55

Che cosa fa?

```
#include <stdio.h>

int * p;

void boh() {      int x = 55;
                  p = &x; }

void bohboh() { int y = 100; }

int main() {
  int x = 1;
  p = &x;
  boh();
  bohboh();
  printf("risultato= %d\n", *p);
  return 0;
}
```



```
#include <stdio.h>

int * p;

void boh() {      int x = 55;
                  p = &x; }

void bohboh() { int y = 100; }

int main() {
  int x = 1;
  p = &x;
  boh();
  bohboh();
  printf("risultato= %d\n", *p);
  return 0;
}
```

Che cosa fa?

p è assegnato all'indirizzo della x "statica" del main, che ha valore 1.

Poi la chiamata di boh() lo rias-segna alla x "automatica" (p è globale!)

Poi boh termina, e il suo record di attivazione è sovrascritto da quello di bohboh, che è strutturalmente uguale

Quindi la y "cade" dove prima c'era la x, e la printf **stampa** il valore **100**

SONO ESEMPI DI QUELLO CHE NON SI DEVE FARE CON I PUNTATORI

(indipendentemente dalla memoria dinamica)

Avvertimento

Puntatori e variabili dinamiche portano a programmare a basso livello e **"pericolosamente" se non si tiene conto dei meccanismi di gestione della memoria**

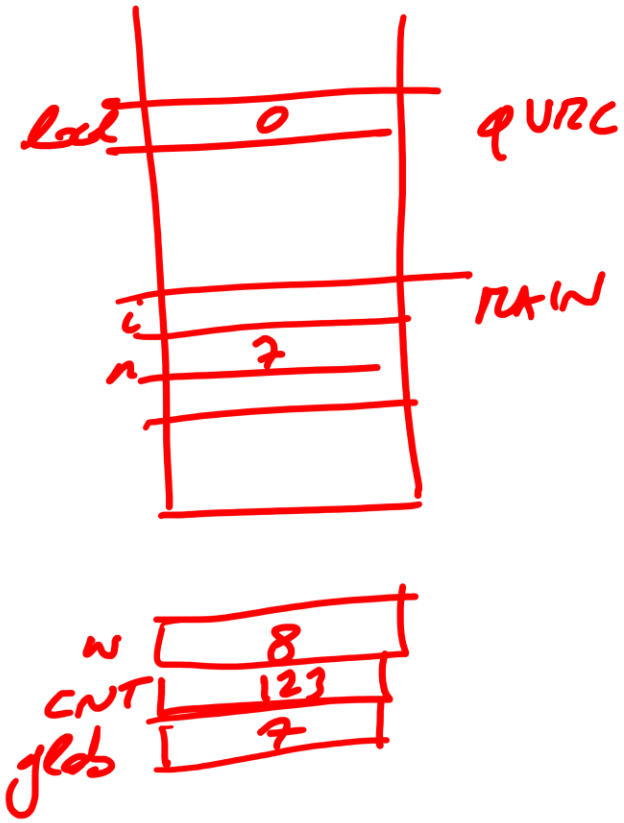
→ Sono da usare con consapevolezza:

- per passare parametri per indirizzo
- per costruire strutture dati complesse
  - Liste, alberi, grafi, ... (che studiamo subito)
- In altri casi di uso della mem. dinamica

Esempio di variabile statica e locale

```
void quanteVolteMiChiamano() {
    static int cont=0;
    cont++;
    printf("%d“,cont);
}
```

```c
void quanteVolteMiChiamano(void);
int glob = 7;
int main()
{
    int i, n = 4;
    static int w = 8; // locale al main, non visibile nella funzione

    printf("\nindirizzo di int i: %p", &i);
    printf("\nindirizzo di static int w: %p", &w);

    for(i = 0; i < n; i++)
            quanteVolteMiChiamano();
    printf("\n global %d", glob);
    return 0;
}

void quanteVolteMiChiamano(void)
{
    static int cont = 123;
    // 123 è il valore dell'inizializzazione nella prima esecuzione di questa riga
    // - se non compare viene inizializzata a zero
    // - se la variabile esiste, le successive esecuzioni di questa riga vengono ignorate
    int local = 0;
    // w++; // errore, non è visibile è statica ma del main
    local++; //
    cont++; // cont è statica e permane nelle varie chiamate
    glob++; // glob è globale ed è raggiungibile da questa funzione
    printf("\ncont (static) = %5d, local (automatic) = %d, glob (global) = %d ", cont, local, glob);
}
```



```
indirizzo di int i: 0060FF08
indirizzo di static int w: 00402004
global 7 (00402000)
cont (static) =   124, local (automatic) = 1, glob (global) = 8

cont (static) =   125, local (automatic) = 1, glob (global) = 9

cont (static) =   126, local (automatic) = 1, glob (global) = 10

cont (static) =   127, local (automatic) = 1, glob (global) = 11

 global 11
Process returned 0 (0x0)   execution time : 0.090 s
Press any key to continue.
```

`cont` non è parte visibile nel main, lo scope è limitato alla funzione

```c
void quanteVolteMiChiamano(void);
int glob = 7;
int main()
{
    int i, n = 7;
    static int w = 8; // locale al main, non visibile nella funzione

    printf("\nindirizzo di int i: %p", &i);
    printf("\nindirizzo di static int w: %p", &w);

    for(i = 0; i < n; i++)
            quanteVolteMiChiamano();

    printf("\n glob = %d cont = %d", glob, cont);
    return 0;
}

void quanteVolteMiChiamano(void)
{
    static int cont = 123;
    // 123 è il valore dell'inizializzazione nella prima esecuzione di questa riga
    // - se non compare viene inizializzata a zero
    // - se la variabile esiste, le successive esecuzioni di questa riga vengono ignorate
    int local = 0;
    // w++; // errore, non è visibile è statica ma del main
    local++; //
    cont++; // cont è statica e permane nelle varie chiamate
    glob++; // glob è globale ed è raggiungibile da questa funzione
    printf("\ncont (static) = %5d, local (automatic) = %d, glob (global) = %d ", cont, local, glob);
}
```

'cont' undeclared (first use in this function);

```
unsigned int randomNumber(int s){
static unsigned int seed;
seed = (F * seed + I) % M;
//printf("\n&seed (static) = %p", &seed); // sta nello
stac
return seed;
}
```

Il seed per la generazione di numeri casuali è una variabile statica. Ecco un semplice esempio.

Allocazione della memoria

- In C i dati hanno una **dimensione** nota a **tempo di compilazione** che posso calcolare con la funzione sizeof(…)
  - La quantità di memoria necessaria per eseguire una funzione è nota al compilatore
    - dimensione di un record di attivazione
  - Non si conosce, però, il numero di "istanze" del record di attivazione da allocare (esempio: ricorsione)
- Come si gestiscono i dati la cui dimensione è nota solo a **tempo di esecuzione**?
  - Es: invertiamo una sequenza di interi letti da stdin

Soluzione con array

Invertiamo una sequenza di interi letti da stdin

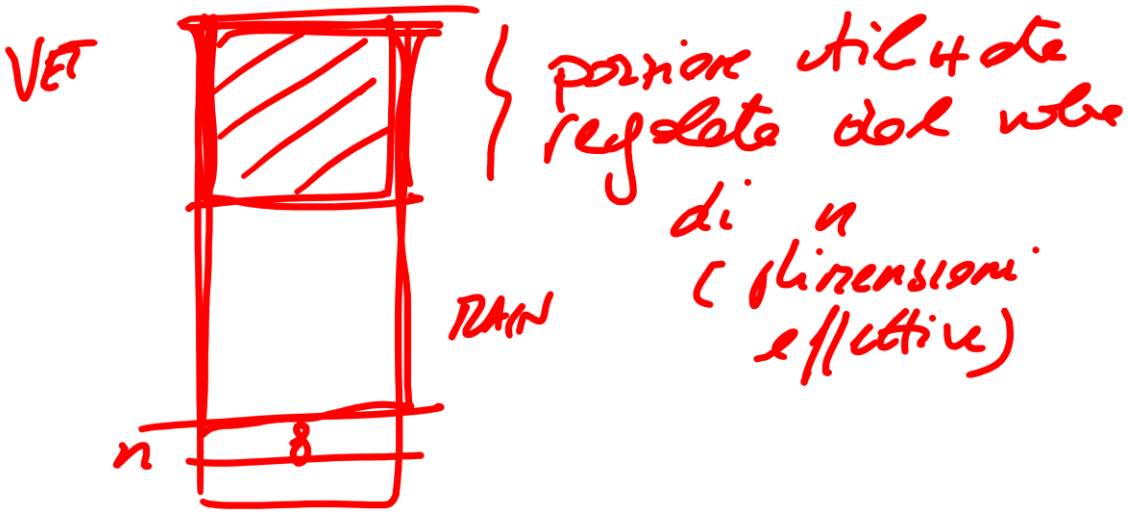
- Si **prealloca** un'area dati **sovradimensionata** rispetto all'effettivo utilizzo
- Si tiene traccia di quanta parte di essa è effettivamente occupata
  - Nell'esempio della sequenza, l'indice che progredisce "conta" i valori validi inseriti

Soluzione con array

Invertiamo una sequenza di interi letti da stdin

- Si **prealloca** un'area dati **sovradimensionata** rispetto all'effettivo utilizzo
- Si tiene traccia di quanta parte di essa è effettivamente occupata
  - Nell'esempio della sequenza, l'indice che progredisce "conta" i valori validi inseriti

```
int main()
{ int VET[100]
  int n;
```



Soluzione con array

Invertiamo una sequenza di interi letti da stdin

- Si **prealloca** un'area dati **sovradimensionata** rispetto all'effettivo utilizzo
- Si tiene traccia di quanta parte di essa è effettivamente occupata
  - Nell'esempio della sequenza, l'indice che progredisce "conta" i valori validi inseriti

→ **Problema: grande spreco di memoria**

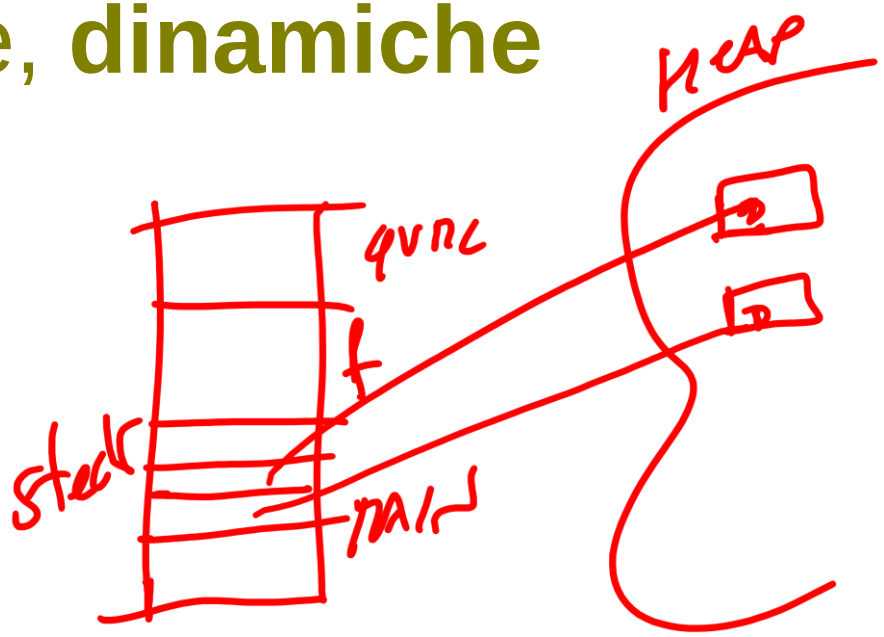
→ **Soluzione: MEMORIA DINAMICA**

Memoria dinamica: motivazioni

- Dimensionamento "fisso" iniziale (ad esempio di array) – problemi tipici:
  - **Spreco** di memoria se a runtime i dati sono pochi
  - **Violazione di memoria** se i dati sono più del previsto
    - Un **accesso oltre il limite** dell'array ha effetti imprevedibili
  - **Spreco di tempo per *ricompattare/spostare*** i dati
    - **Cancellazione di un elemento intermedio** in un array ordinato
      - occorre far scorrere "indietro" tutti gli elementi successivi
    - **Inserimento di un elemento intermedio** in un array ordinato
      - occorre far scorrere "in avanti" i dati per creare spazio

Variabili statiche, automatiche, **dinamiche**

- *Statiche*
  - *allocate prima dell'esecuzione del programma*
  - *restano allocate per tutta l'esecuzione*
- *Automatiche*
  - *allocate e deallocate automaticamente*
  - *gestione della memoria a stack (LIFO)*
- **Dinamiche**
  - **Allocate e deallocate esplicitamente a run-time (leggi, durante l'esecuzione del programma)**
  - **Non hanno un nome/riferimento simbolico!**
  - **Accessibili solo tramite puntatori**
  - **Referenziabili "da ogni ambiente"**
    - A patto di **disporre di un puntatore** che punti ad esse



Gestione della memoria

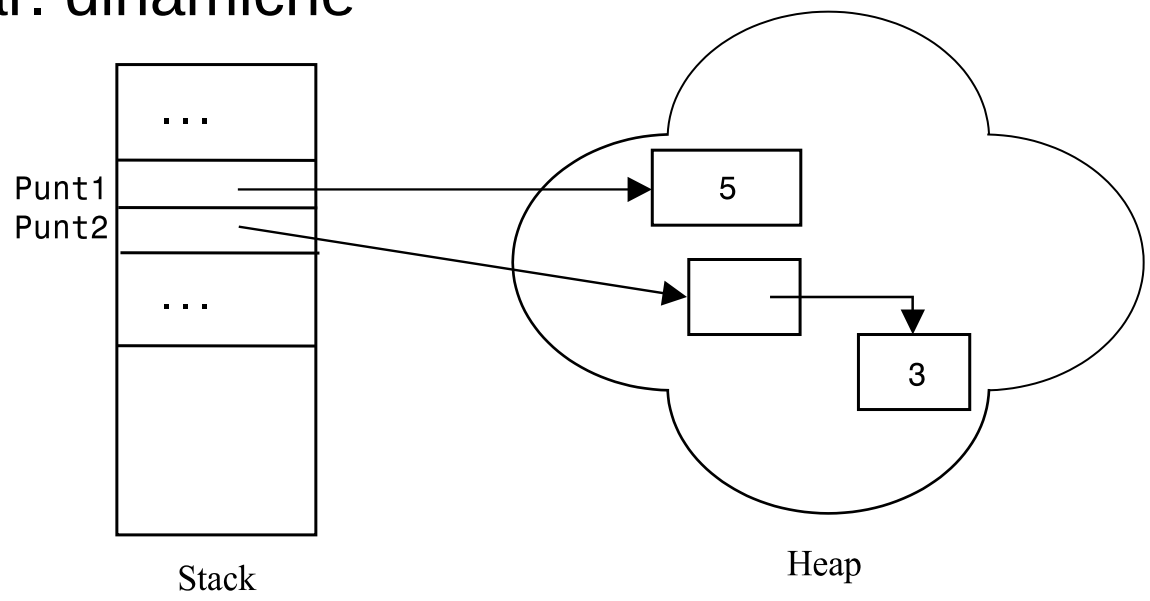
La memoria riservata ai dati del programma è partizionata in due "zone"

- pila (***stack***) per var. statiche e automatiche
- mucchio (***heap***) per var. dinamiche

Queste variabili stanno nello stack (sono dichiarate normalmente).Non abbiamo ancora allocato le variabili dell'heap

Esempio

```
int * Punt1;
int ** Punt2;
```



Punt2 è un puntatore ad un puntatore nello heap!

Questo è completamente trasparente.

Gestione della memoria

La memoria riservata ai dati del programma è partizionata in due "zone"

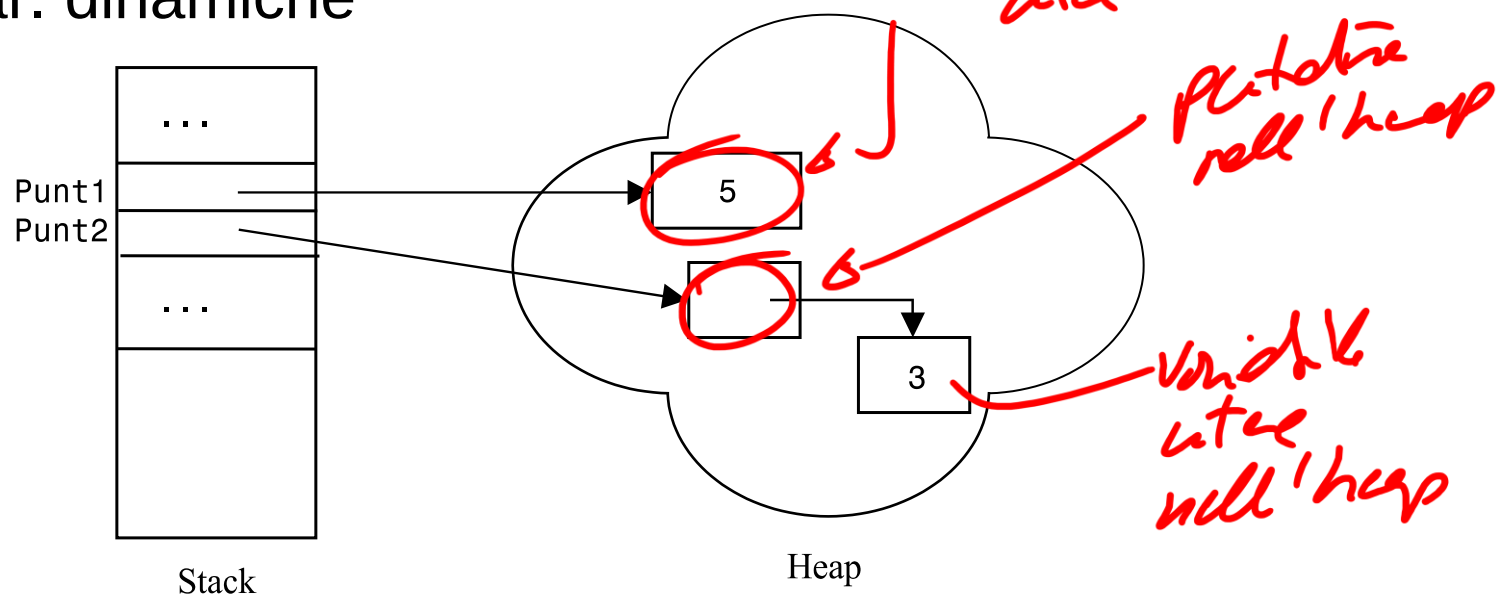
- pila (***stack***) per var. statiche e automatiche
- mucchio (***heap***) per var. dinamiche

Queste variabili stanno nello stack (sono dichiarate normalmente).Non abbiamo ancora allocato le variabili dell'heap

Esempio

int * Punt1;

int ** Punt2;



Punt2 è un puntatore ad un puntatore nello heap!

Questo è completamente trasparente.

Allocazione e **Rilascio** di memoria

- **Apposite funzioni** definite nella standard library <stdlib.h> si occupano della **gestione della memoria dinamica**:
  - malloc(...) ***m****emory* ***alloc****ation* - per l'allocazione
  - free(...) per il rilascio

- Queste funzioni possono essere **invocate a runtime** in qualsiasi momento per agire sullo heap

Puntatori e `<stdlib.h>`

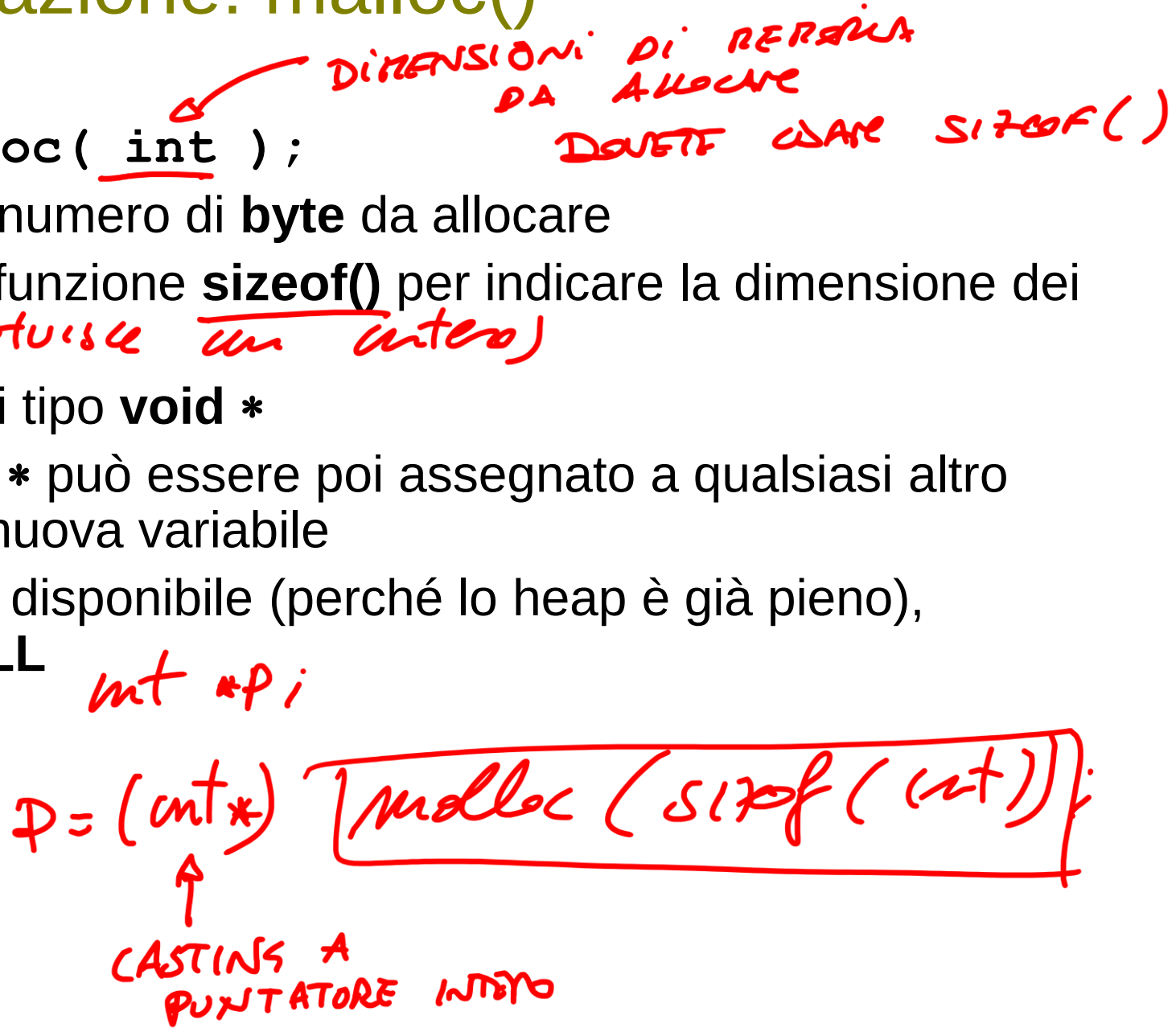
La libreria **stdlib.h** contiene:

- I prototipi delle funzioni di allocazione dinamica della memoria
  - malloc(...)
  - free(...)
- La dichiarazione della costante NULL
  - puntatore nullo
  - non punta ad alcuna area significativa di memoria
    - ANSI impone che rappresenti il valore 0

Allocazione: malloc()

La funzione malloc(...)

- Prototipo: `void * malloc( int );`
- Riceve come parametro il numero di **byte** da allocare
  - Normalmente si usa la funzione **sizeof()** per indicare la dimensione dei dati da allocare
- Restituisce un puntatore di tipo **void ***
  - il puntatore di tipo **void *** può essere poi assegnato a qualsiasi altro puntatore per usare la nuova variabile
  - se non c'è più memoria disponibile (perché lo heap è già pieno), malloc() restituisce **NULL**



Tipi e dimensione di memoria occupata

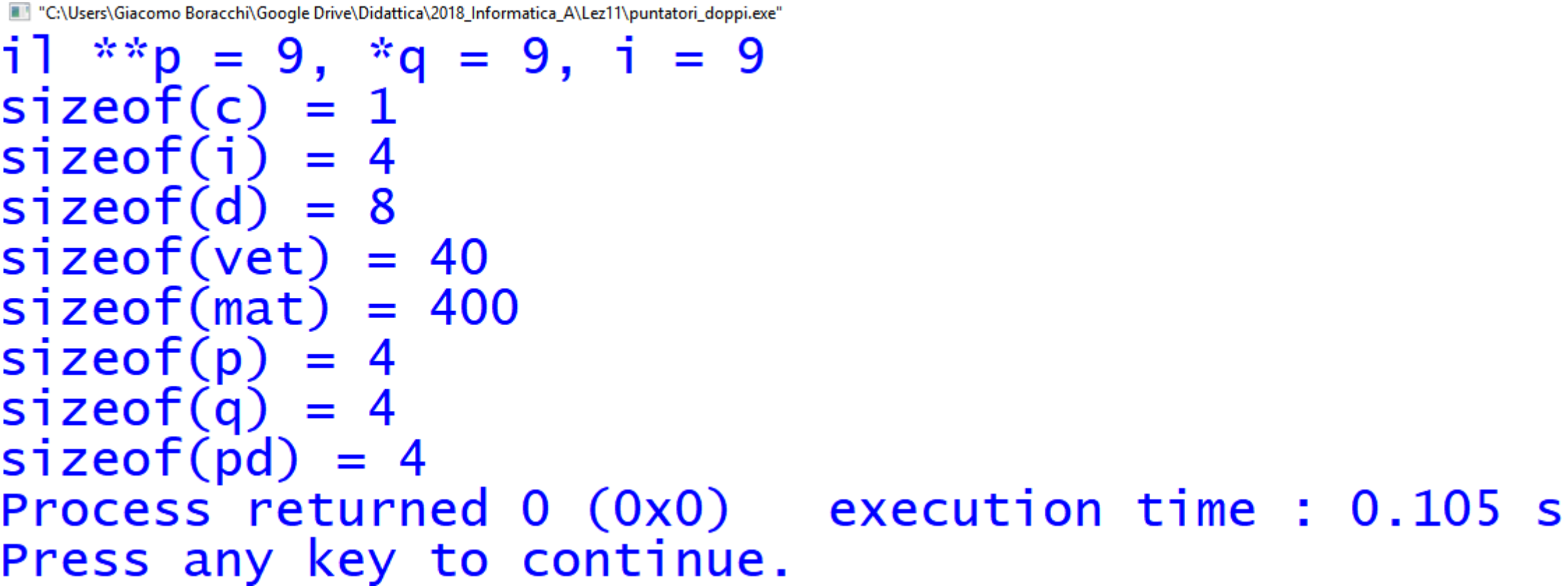
- Le variabili occupano in memoria un numero di parole che dipende dal tipo
- Sono allocate in parole di memoria consecutive
- L'operatore **`sizeof()`** dà il numero di byte occupati da un tipo (o da una variabile):

```
double A[5], *p;

sizeof(A[2]) →   8
sizeof(A)    →  40
sizeof(p)    →   4
```

```c
#include<stdio.h>
int main(){
    int **p, *q, i = 9;
    int vet[10], mat[10][10];
    char c = '0';
    double d = 0.0, *pd;

    p = &q;
    q = &i;
    printf("**p=%d, *q=%d, i=%d", **p, *q, i);

    printf("\nsizeof(c) = %d", sizeof(c));
    printf("\nsizeof(i) = %d", sizeof(i));
    printf("\nsizeof(d) = %d", sizeof(d));
    printf("\nsizeof(vet) = %d", sizeof(vet));
    printf("\nsizeof(mat) = %d", sizeof(mat));
    printf("\nsizeof(p) = %d", sizeof(p));
    printf("\nsizeof(q) = %d", sizeof(q));
    printf("\nsizeof(pd) = %d", sizeof(pd));
    return 0;
}
```

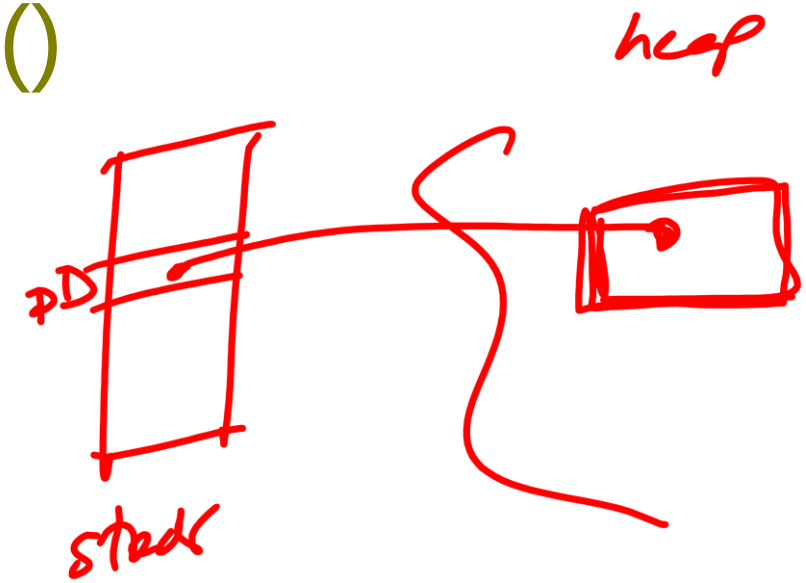
L'operatore `sizeof`

- Le dimensioni in memoria dipendono dal tipo della variabile
- Le dimensioni dei puntatori (e doppi puntatori) sono quelle della singola cella, indipendentemente dal loro tipo
- Le dimensioni dei vettori contengono tutte le celle allocate per l'array



Allocazione: malloc()

```
typedef …any definition… TipoDato;
typedef TipoDato * PTD;
PTD ref;
…
ref = (PTD) malloc( sizeof(TipoDato) );
```



- Alloca nello heap una variabile dinamica (grande quanto un **TipoDato**) e restituisce l'indirizzo della prima cella occupata da tale variabile
  - **LA VARIABILE RESTITUITA DALLA MALLOC DI PER SÉ È ANONIMA,** si può accedere solo tramite il puntatore a cui viene assegnata inizialmente
  - Ovviamente **`ref`** perde il valore precedente, e punta alla nuova variabile, che è accessibile per dereferenziazione (**`*ref`**)

Attenzione

- Lo spazio allocato da malloc() è per la nuova variabile, di tipo TipoDato
- Non è per il puntatore **ref**, che già esisteva!
  - ref è una variabile **STATICA**

CAST ESPLICITO

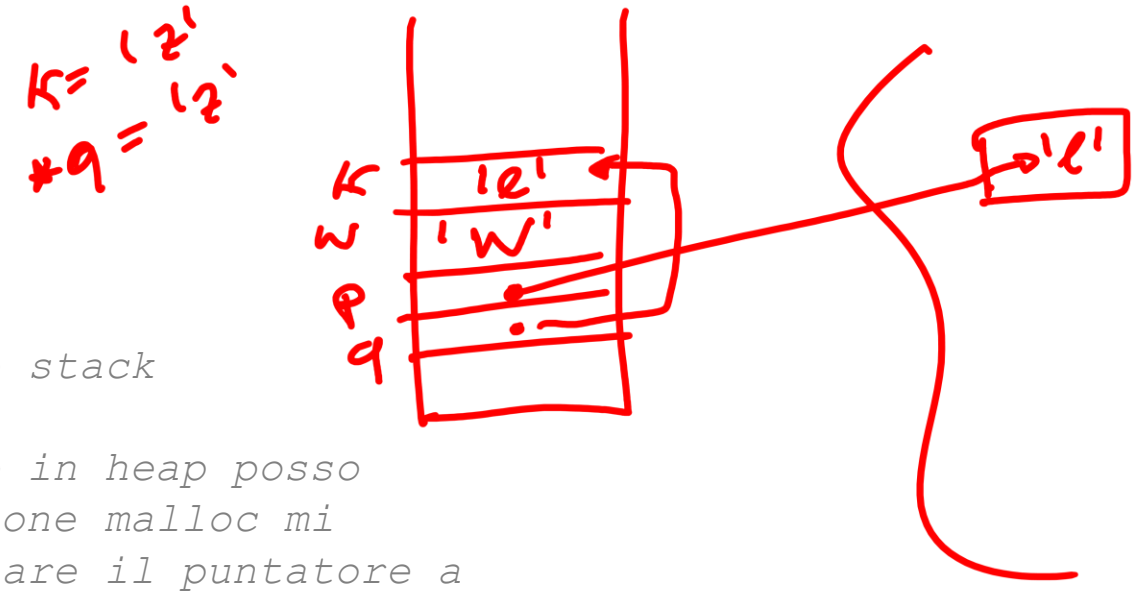
```
ref = (PTD)malloc( sizeof(TipoDato) );
```

  - Il cast esplicito specifica al compilatore che il programmatore è consapevole che il puntatore è convertito ***da*** void * ***a*** PTD (cioè a puntatore a `TipoDato` come tipo di dato)
  - A volte sarà omesso (come nel libro di testo)
    - Si tenga comunque presente che alcune piattaforme non segnalano nulla, altre segnalano un warning, altre ancora ne considerano l'omissione un vero e proprio errore
- Il casting è possibile anche tra variabili built in, non solo con i puntatori, ad esempio

`n = (float) x;` equivale a `n = 1.0 * x;`

```c
#include <stdio.h>

int main()
{
    char *p, *q, k, w = 'W';

    k = 'l';
    q = &k; // q punta ad una variabile nello stack

    /* alloco una variabile dinamica, finisce in heap posso
accederci solo tramite un puntatore, la funzione malloc mi
restituisce il puntatore a void. occorre forzare il puntatore a
void (casting) a diventare un puntatore a char */

    p = (char *) malloc (sizeof(char));
    *p = 'l';

    // si noti come siano diversi gli indirizzi di p, q e w
    printf("HEAP: *p = %c (%p)", *p, p); // questi cambiano con
diversi run
    printf("\nSTACK: *q = %c (%p), w = %c, (%p)", *q, q, w, &w);

    return 0;

}
```

```c
#include <stdio.h>

int main()
{
    char *p, *q, k, w = 'W';

    k = 'l';
    q = &k; // q punta ad una variabile nello stack

    /* alloco una variabile dinamica, finisce in heap posso
accederci solo tramite un puntatore, la funzione malloc mi
restituisce il puntatore a void. occorre forzare il puntatore a
void (casting) a diventare un puntatore a char */

    p = (char *) malloc (sizeof(char));
    *p = 'l';

    // si noti come siano diversi gli indirizzi di p, q e w
    printf("HEAP: *p = %c (%p)", *p, p); // questi cambiano con
diversi run
    printf("\nSTACK: *q = %c (%p), w = %c, (%p)", *q, q, w, &w);

    return 0;

}
```



```
"C:\Users\Giacomo Boracchi\Google Drive\Didattica\2018_Informatica_A\Lez14\mainMalloc.exe"
HEAP: *p = 1 (00AE0DE0)
STACK: *q = 1 (0060FF07), w = W, (0060FF06)
Process returned 1 (0x1)   execution time : 0.098 s
Press any key to continue.
```

```
"C:\Users\Giacomo Boracchi\Google Drive\Didattica\2018_Informatica_A\Lez14\mainMalloc.exe"
HEAP: *p = 1 (00B30DE0)
STACK: *q = 1 (0060FF07), w = W, (0060FF06)
Process returned 1 (0x1)   execution time : 0.059 s
Press any key to continue.
```

```
"C:\Users\Giacomo Boracchi\Google Drive\Didattica\2018_Informatica_A\Lez14\mainMalloc.exe"
HEAP: *p = 1 (00710DE0)
STACK: *q = 1 (0060FF07), w = W, (0060FF06)
Process returned 1 (0x1)   execution time : 0.067 s
Press any key to continue.
```

Risultati di **tre esecuzioni** differenti, senza aver ricompilato:

- Le variabili nello stack k (a cui punta q) e w sono vicine
- La variabile a cui punta p (allocata dinamicamente) è ad un indirizzo molto diverso.
- L'indirizzo delle variabili nello heap cambia quando ri-eseguo il programma, l'indirizzo nello stack non cambia

Deallocazione: free()

- **La funzione free()**
  - Prototipo: `void free( void * );`
  - Libera la memoria allocata tramite la **malloc**, che dopo l'esecuzione è pronta ad essere riusata
  - Riceve un puntatore **void** `*` come argomento
  - `free( ref );`

- N.B.: non serve specificare la dimensione in byte, che è derivabile automaticamente

malloc() e free()

- Esempio: allocare una var. dinamica di tipo char, assegnarle 'a', stamparla e infine deallocarla

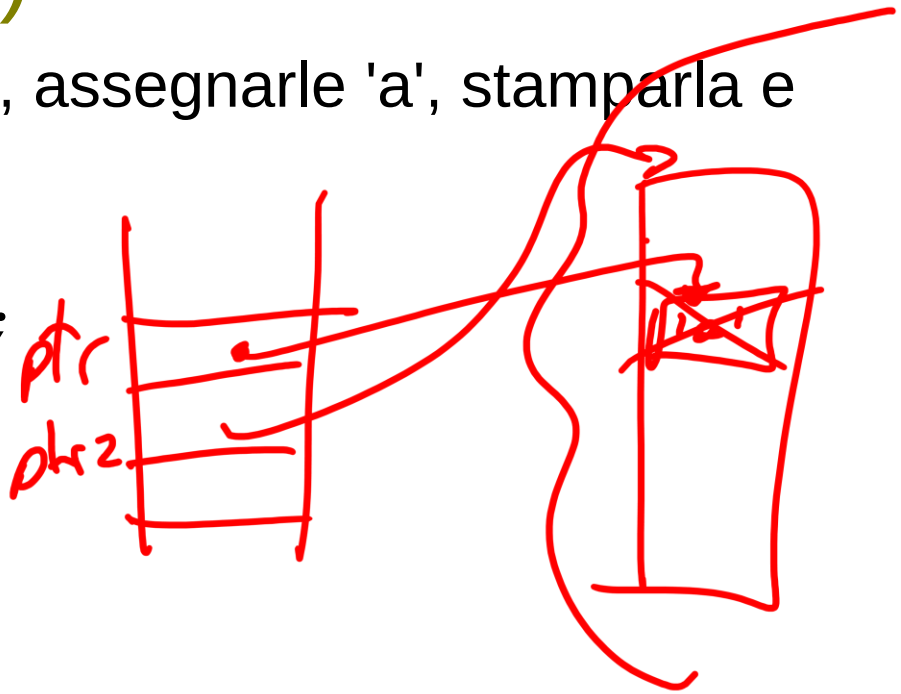
```
char * ptr;
ptr = (char *) malloc( sizeof(char) );
*ptr = 'a';
printf("Carattere: %c\n", *ptr);
free( ptr );
```

- Attenzione:
  - **`ptr` NON è eliminato**, e può essere riusato per una nuova **`malloc`**
    - **`ptr`** è UNA VARIABILE STATICA, quindi NON DEALLOCABILE

malloc() e free()

- Esempio: allocare una var. dinamica di tipo char, assegnarle 'a', stamparla e infine deallocarla

```
char * ptr;
ptr = (char *) malloc( sizeof(char) );
*ptr = 'a';
printf("Carattere: %c\n", *ptr);
free( ptr );
```

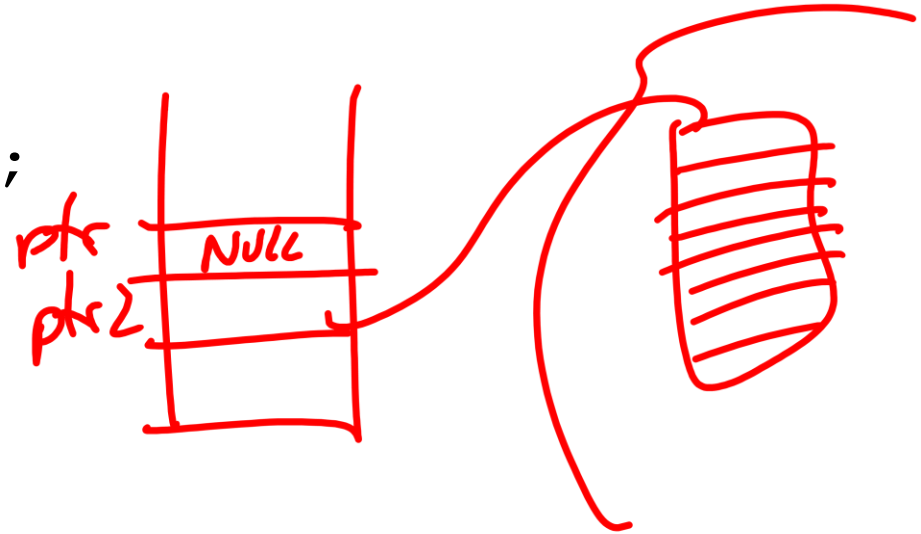


- Attenzione:
  - **`ptr` NON è eliminato**, e può essere riusato per una nuova **`malloc`**
    - `ptr` è UNA VARIABILE STATICA, quindi NON DEALLOCABILE

malloc() e free()

- Esempio: allocare una var. dinamica di tipo char, assegnarle 'a', stamparla e infine deallocarla

```
char * ptr;
ptr = (char *) malloc( sizeof(char) );
*ptr = 'a';
printf("Carattere: %c\n", *ptr);
free( ptr );
ptr=NULL;
```



*ptr = 7

- Attenzione:
  - **`ptr` NON è eliminato**, e può essere riusato per una nuova **`malloc`**
    - `ptr` è UNA VARIABILE STATICA, quindi NON DEALLOCABILE
  - **Il valore di `ptr` non cambia** (contiene lo stesso indirizzo) solo che la porzione di heap è stata «ripulita» e pronta per essere assegnata ad altri. E' sbagliato continuare ad accederci
  - Quindi è meglio assegnare **`p`** a **`NULL`** e poterlo ri-utilizzare poi

Confrontare ...

```
char c = 'a';  /* varibile char STATICA */
printf("Carattere: %c\n", c);
```

con

```
char c;       /* varibile char STATICA */
char * ptr; /* puntatore */
ptr = &c; *ptr = 'a';
printf("Carattere: %c\n", *ptr);
```

e

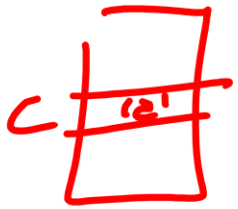
```
char * ptr; /* puntatore */
ptr = (char*) malloc( sizeof(char) ); /* var. char DINAMICA */
*ptr = 'a';
printf("Carattere: %c\n", *ptr);
free( ptr );
ptr=NULL;
```

e

```
char c;       /* varibile char STATICA */
void * ptr; /* puntatore "buono per tutti gli usi" */
ptr = &c; *ptr = 'a';
printf("Carattere: %c\n", *ptr);
```

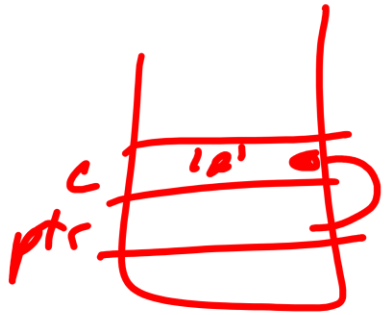
Confrontare ...

```
char c = 'a';  /* varibile char STATICA */
printf("Carattere: %c\n", c);
```



con

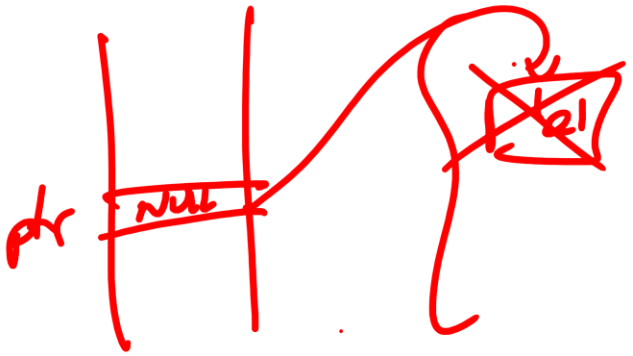
```
char c;       /* varibile char STATICA */
char * ptr; /* puntatore */
ptr = &c; *ptr = 'a';
printf("Carattere: %c\n", *ptr);
```



e

```
char * ptr; /* puntatore */
ptr = (char*) malloc( sizeof(char) ); /* var. char DINAMICA */
*ptr = 'a';
printf("Carattere: %c\n", *ptr);
free( ptr );
ptr=NULL;
```

(se non faccio la free() ottengo "garbage"

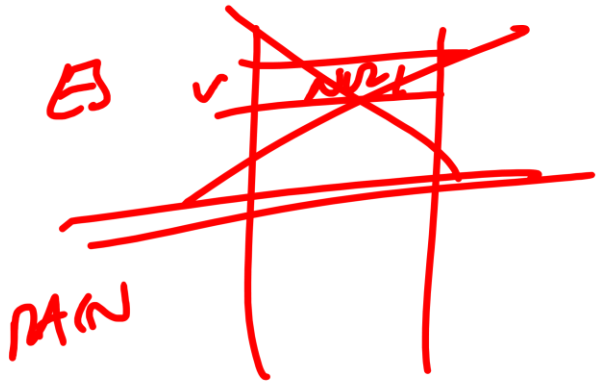


Allocazione Dinamica Array

- Avevamo imparato studiando questo problema che l'uso degli array può semplificare assai la scrittura dei programmi
- Restava però "irrisolto" il problema del pre-dimensionamento opportuno dell'array
- Ora possiamo pensare di allocare un array **dinamico** "piccolo", e sostituirlo con uno più grande solo se necessario
- ATTENZIONE:
  - La malloc() alloca blocchi contigui di memoria ad ogni invocazione, ma invocazioni diverse restituiscono blocchi totalmente scorrelati
  - Quando un vettore si riempie, quindi, occorre ricopiare nel nuovo vettore la sequenza memorizzata fino a quel punto

```c
#include<stdlib.h>
#define LEN 2
#define INCR 2
#define STOP_VAL -1

int* estendi(int*, int, int);

int main()
{
    int *v, len, i, n;
    int lung_max; // lunghezza massima del vettore
    // inizializzo v ad un vettore di lunghezza LEN
    v = (int *) malloc(LEN * sizeof(int));
    len = 0;
    n = 0;
    lung_max = LEN;
    while(n != STOP_VAL)
    {
        printf("inserire v[%d]: ", len);
        scanf("%d", &n);
        v[len] = n;
        len++;
        if(len == lung_max)
        {
            // invoco la funzione per copiare v
            // in un vettore grande len + INCR in heap
            v = estendi(v, len, INCR);
            lung_max += INCR; // aggiorno la lung. max
        }
    }
    printf("\n");
    for(i = 0; i < len; i++)
        printf(" %d ", v[i]);

    return 0;
}
```

```c
int* estendi(int *v, int len, int incr)
{
    int *q, i;
    // dichiaro un array in heap di lunghezza len + incr
    q = (int *) malloc(sizeof(int)*(len + incr));
    for(i = 0; i < len; i++)
            q[i] = v[i]; // oss non modificare il puntatore, devi restituirlo poi

    free(v); // importante, se non libero lo heap si intasa a furia di invocare!
    return q;
}
```



non serve fare v=NULL perché v è una variabile locale e il R.A. viene distrutto.

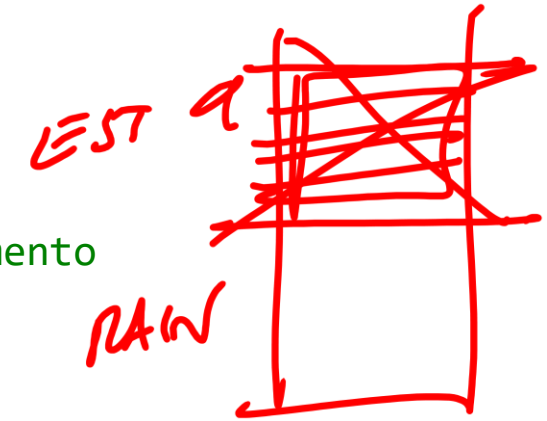
E se avessi usato un vettore?

Le righe evidenziate inconsistenze: si prova a modificare il vettore! Il vettore, inteso come l'indirizzo della prima cella, è una costante. Inoltre, con `int v[LEN]` alloco nello stack, non serve fare la malloc

```c
#include<stdlib.h>
#define LEN 2
#define INCR 2
#define STOP_VAL -1

int* estendi(int*, int, int);

int main()
{
    int v[LEN], len, i, n;
    int lung_max; // lunghezza massima del vettore
    // inizializzo v ad un vettore di lunghezza LEN
    v = (int *) malloc(LEN * sizeof(int));
    len = 0;
    n = 0;
    lung_max = LEN;
    while(n != STOP_VAL)
    {
        printf("inserire v[%d]: ", len);
        scanf("%d", &n);
        v[len] = n;
        len++;
        if(len == lung_max)
        {
            // invoco la funzione per copiare v
            // in un vettore grande len + INCR in heap
            v = estendi(v, len, INCR);
            lung_max += INCR; // aggiorno la lung. max
        }
    }
    printf("\n");
    for(i = 0; i < len; i++)
        printf(" %d ", v[i]);

    return 0;
}
```

E se non avessi usato la memoria dinamica?

```c
int* estendi(int *v, int len, int incr)
{
    int q[2*LEN], i; // OSS: potrebbe non bastare come incremento
    for(i = 0; i < len; i++)
            q[i] = v[i];
    return q;
}
```



Oltre a non poter definire ad ogni invocazione di «estendi» una diversa dimensione di q, c'è un problema ancora più fondamentale: **Il vettore q è dichiarato nel record di attivazione di «estendi» che al termine dell'invocazione viene distrutto.**
La memoria dinamica invece è accessibile sia da stack che da heap

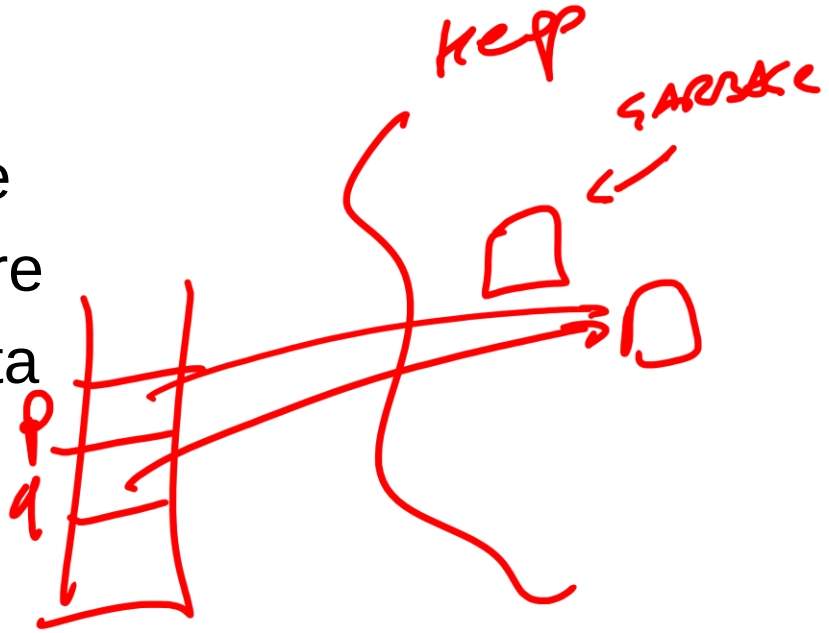
```c
        len++;
        if(len == lung_max)
        {
            // invoco la funzione per copiare v
            // in un vettore grande len + INCR in heap
            v = estendi(v, len, INCR);
            lung_max += INCR; // aggiorno la lung. max
        }
    }
    printf("\n");
    for(i = 0; i < len; i++)
        printf(" %d ", v[i]);

    return 0;
}
```

Allocazione Dinamica Array

- Ora sappiamo usare (quasi) solo la memoria realmente necessaria per la memorizzazione
- Lo schema di incremento è piuttosto rigido
  - Si può migliorare per cercare di limitare il numero di ricopiature, man mano che la sequenza si allunga
    - Incrementare / raddoppiare l'incremento a ogni incremento
    - Incrementare ogni volta di una percentuale (fissa o variabile)
- *Possiamo ancora migliorare questa soluzione*: impareremo **come allocare le variabili una alla volta, senza ricopiare mai la sequenza**

Produzione di "spazzatura"

- La memoria allocata dinamicamente può diventare **inaccessibile** se nessun puntatore punta più ad essa
  - Risulta **sprecata**, e non è recuperabile
    - Per invocare free() è necessario un puntatore
  - È "spazzatura" (***garbage***) che non si può smaltire
- Occorre fare la free della memoria dinamica allocata

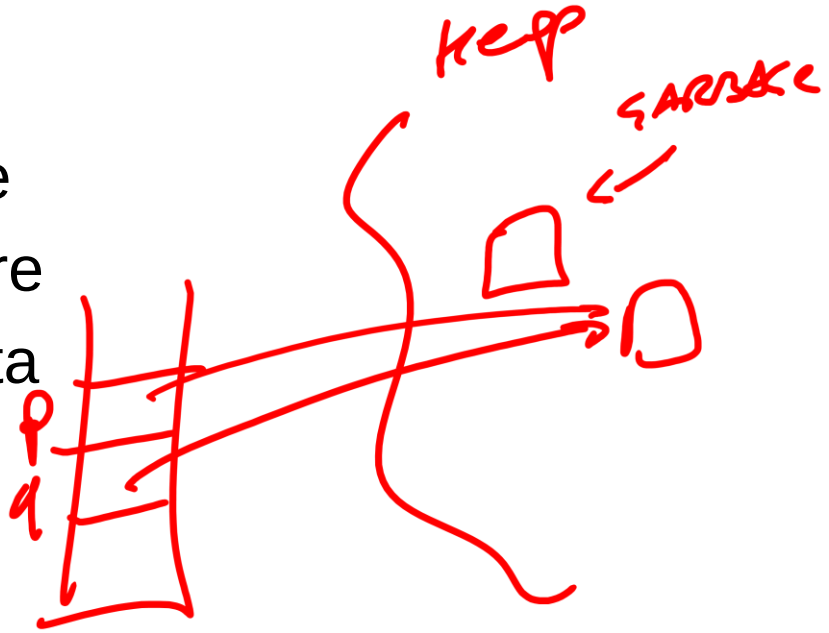


- Esempio banale

```
TipoDato *P, *Q;
P = (TipoDato *) malloc(sizeof(TipoDato));
Q = (TipoDato *) malloc(sizeof(TipoDato));
P = Q; /* la variabile che era puntata da P è garbage */
```

Produzione di "spazzatura"

- La memoria allocata dinamicamente può diventare **inaccessibile** se nessun puntatore punta più ad essa
  - Risulta **sprecata**, e non è recuperabile
    - Per invocare free() è necessario un puntatore
  - È "spazzatura" (***garbage***) che non si può smaltire
- Occorre fare la free della memoria dinamica allocata

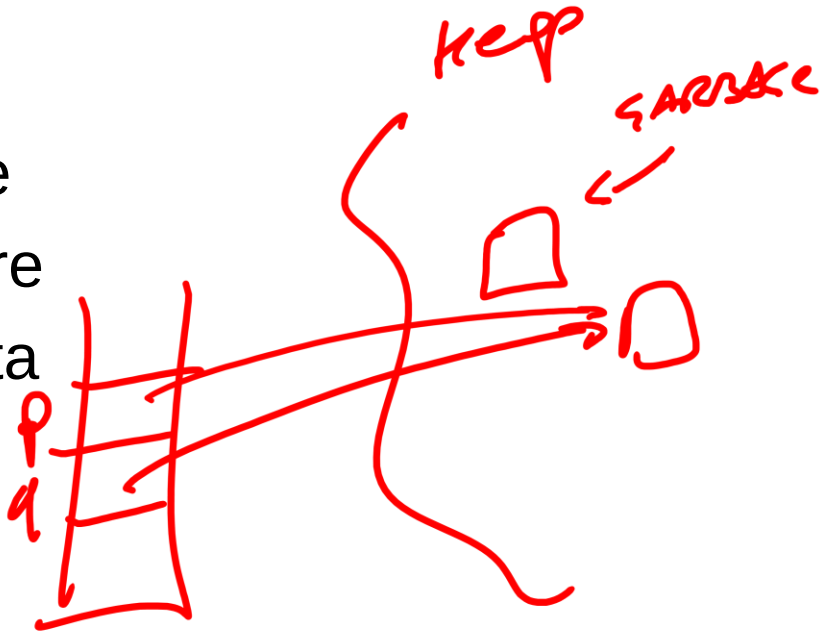


- Esempio banale

```
TipoDato *P, *Q;
P = (TipoDato *) malloc(sizeof(TipoDato));
Q = (TipoDato *) malloc(sizeof(TipoDato));
free(P) /* così non creo garbage */
P = Q; /* la variabile che era puntata da P è garbage */
```

Produzione di "spazzatura"

- La memoria allocata dinamicamente può diventare **inaccessibile** se nessun puntatore punta più ad essa
  - Risulta **sprecata**, e non è recuperabile
    - Per invocare free() è necessario un puntatore
  - È "spazzatura" (***garbage***) che non si può smaltire
- Occorre fare la free della memoria dinamica allocata



- Esempio banale

```
TipoDato *P, *Q;
P = (TipoDato *) malloc(sizeof(TipoDato));
Q = (TipoDato *) malloc(sizeof(TipoDato));
P = Q; /* la variabile che era puntata da P è garbage */
```

free(P); [NON CREO SPAZZATURA

Altro problema…i puntatori "ciondolanti"

- Detti abitualmente *dangling references*
- Sono puntatori a zone di memoria deallocate
  (puntano a variabili dinamiche "non più esistenti")

```
P = Q;
free(Q);
/* ora accedere a *P causa un errore */
```

- Si risolverebbe facendo **`P = NULL`** ma è difficile ricordarsene, visto che si fa **`free(Q)`** e non figurano **`free`** su **`P;`**

- Portano a veri e propri errori
  - Alcuni linguaggi (come Java) non hanno una operazione **`free()`**, ma un ***garbage collector***
    - Un componente della macchina astratta che trova e riutilizza la memoria inaccessibile (non più referenziata)