Variabili dinamiche

Obiettivi:

- Presentare le variabili dinamiche, allocate e deallocate nell'area HEAP, e le funzioni **`malloc`** e **`free`**

Tipi di variabili in C

- In C è possibile classificare le variabili in base al loro tempo di vita:
  - variabili **globali** **DATA SEGMENT**
  - variabili **automatiche** **STACK**
  - variabili **dinamiche** **HEAP**
- Variabili dinamiche, quando è necessario avere “più” controllo sull’allocazione di memoria

  →Allocazione della memoria “by need”

Variabili globali e automatiche

- Le ***variabili globali*** nel DATA SEGMENT, hanno un tempo di vita pari all'intera esecuzione (idem le variabili ***static***, ma con visibilità legata al punto in cui sono definite)

- Le ***variabili automatiche*** (variabili locali e parametri formali di una attivazione di funzione) allocate e deallocate automaticamente nello STACK
  - Il programmatore non ha la possibilità di influire sul tempo di vita di variabili automatiche che è quello dell'attivazione della funzione

Allocazione dinamica

- Quando?
  - Tutte le volte in cui i dati possono crescere in modo non prevedibile staticamente a tempo di sviluppo
    - Un array con dimensione fissata a compile-time non è sufficiente

→ **È necessario avere "più" controllo sull'allocazione di memoria**

→ **Allocazione della memoria "by need"**

→ **Strutture dati dinamiche (liste e alberi)**

Variabili dinamiche

- Allocate (**`malloc`**) e deallocate (**`free`**) esplicitamente dal programmatore (libreria **`stdlib.h`**)
- Non hanno un identificatore associato, ma possono essere ***riferite soltanto*** attraverso un ***puntatore*** ***→ sarà simile per gli oggetti Java***
- ***Il tempo di vita*** delle variabili dinamiche è l'intervallo di tempo che intercorre tra l'allocazione e la deallocazione (che sono stabilite dal programmatore)

ALLOCAZIONE DINAMICA

Per allocare nuova memoria "al momento del bisogno" si usa una funzione di libreria che "gira" la richiesta al sistema operativo:

`malloc()`

La funzione di memory allocation, `malloc()`:

- chiede al sistema di allocare **un'area di memoria grande *tanti byte quanti*** ne desideriamo (tutti i byte sono contigui)
- ***restituisce l'indirizzo*** dell'area di **memoria allocata**

LA FUNZIONE `malloc()`

- **Ha signature:**

```
void * malloc(size_t dim)
```

- **chiede al sistema di allocare un'area di memoria grande `dim` *byte***
- ***restituisce l'indirizzo* dell'area di memoria allocata** (**`NULL`** se, per qualche motivo, l'allocazione non è stata possibile)
  - è sempre opportuno controllare il risultato di **`malloc()`** prima di usare la memoria fornita
- Il sistema operativo preleva la memoria richiesta **dall'*area heap***

LA FUNZIONE `malloc()`

Praticamente, occorre quindi:

- **specificare quanti byte si vogliono, come parametro passato a `malloc()`**
- ***mettere in un puntatore il risultato fornito da* `malloc()` *stessa***

Attenzione:

- `malloc()` restituisce **un *puro indirizzo*,** ossia **un puntatore "senza tipo"**
- per assegnarlo a uno *specifico puntatore* **occorre *un cast esplicito***

ESEMPIO

- Per allocare dinamicamente 12 byte:

```
float *p;
p = (float*) malloc(12);
```

- Per allocare *lo spazio necessario per 5 interi* (qualunque sia la rappresentazione usata per gli interi):

```
int *p;
p = (int*) malloc(5*sizeof(int));
```

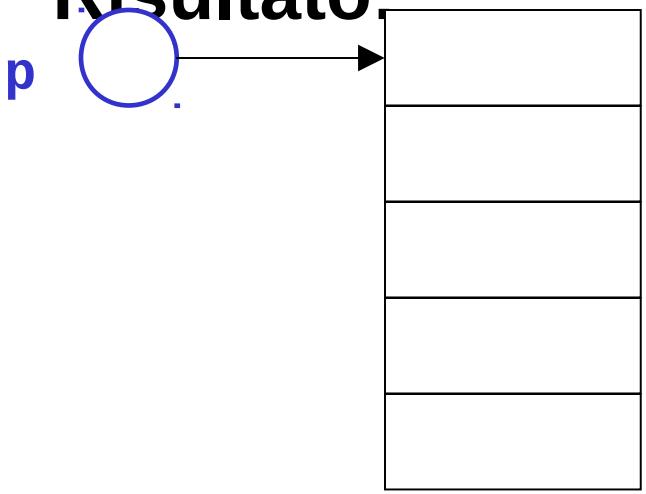
`sizeof` consente di essere indipendenti dalle scelte dello specifico compilatore/sistema di elaborazione

ESEMPIO

Allocazione:

```
int *p;
p = (int*) malloc(5*sizeof(int));
p[4]=7;
```

Risultato:



Sono cinque celle contigue, prelevate dalle celle libere dello HEAP, e ciascuna adatta a contenere un `int`

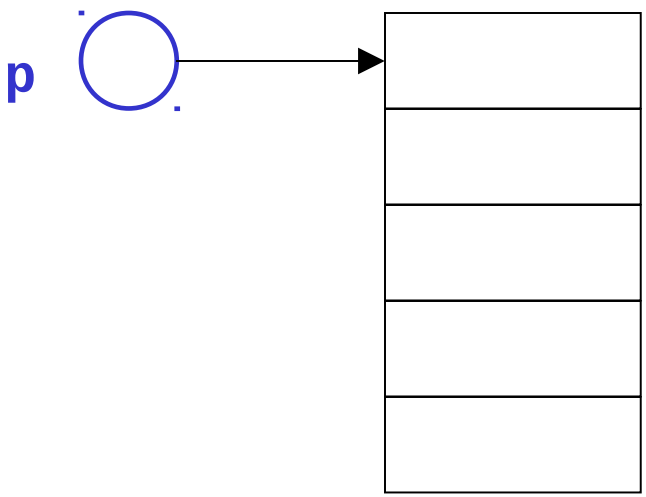
Cosa stiamo allocando?

ESEMPIO (cont)

Allocazione:

```
int *p;
p = (int*) malloc(5*sizeof(int));
```

Risultato:



E' un array, la cui dimensione è stabilita dinamicamente tramite una espressione calcolata

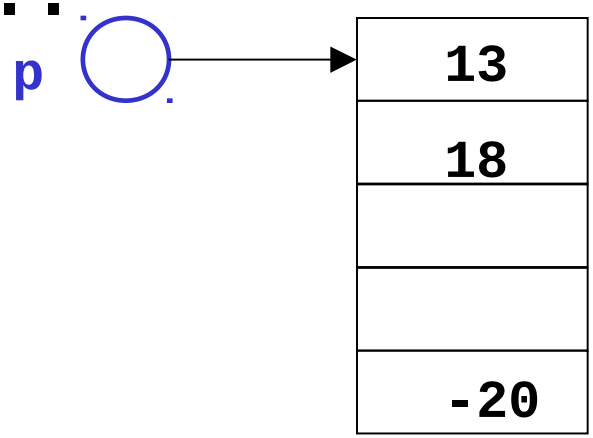
Otteniamo maggiore dinamicità !
Anche se l'array non è "espandibile" …

AREE DINAMICHE: USO

L'area allocata è usabile, in maniera equivalente:

- **o tramite la notazione a puntatore (*p)**
- **o tramite la notazione ad array ([ ])**

```
int *p;
p=(int*)malloc(5*sizeof(int));
p[0] = 13; p[1] = 18;...
*(p+4) = -20;
```



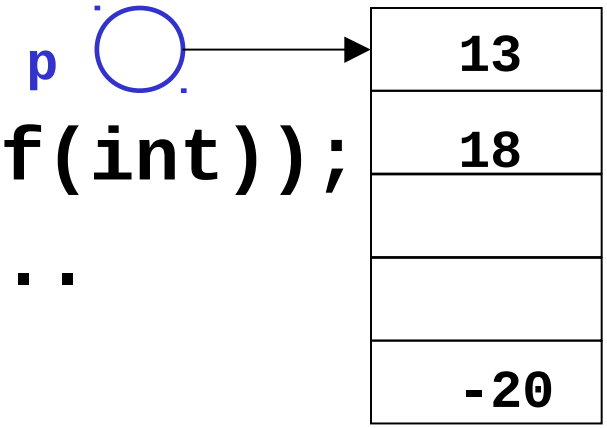
Attenzione a non "eccedere"
l'area allocata dinamicamente.
Non ci può essere alcun controllo

AREE DINAMICHE: USO

Abbiamo costruito un *array dinamico*, le cui dimensioni:

- *non sono determinate <u>a priori</u>*
- *possono essere scelte dal programma <u>in base alle esigenze del momento</u>*
- L'espressione passata a **`malloc()`** può infatti contenere variabili

```
int *p, n=5;
p=(int*)malloc(n*sizeof(int));
p[0] = 13; p[1] = 18;...
*(p+4) = -20;
```



AREE DINAMICHE: DEALLOCAZIONE

Quando non serve più, l'area allocata deve essere *esplicitamente deallocata*

- ciò segnala al sistema operativo che quell'area è da considerare nuovamente disponibile per altri usi

La deallocazione si effettua mediante la *funzione di libreria* `free()`

```
int *p=(int*)malloc(5*sizeof(int));
   ...
free(p);
```

Non è necessario specificare la dimensione del blocco da deallocare, perché *il sistema la conosce già dalla malloc() precedente*

LA FUNZIONE `free()`

- Ha signature:

```
void free(void* p)
```

- Il sistema sa quanta memoria deallocare per quel puntatore (**ricorda** la relativa malloc)
- Se la memoria non viene correttamente deallocata → ***memory leaking***
- In caso di strutture dati condivise, come si decide quando deallocare la memoria?
- In ogni momento occorre sapere **chi** ha in uso una certa struttura condivisa per ***deallocare solamente quando più nessuno ne ha un riferimento***

ESERCIZIO 1

Creare un array di float **di dimensione specificata dall'utente**

```
#include <stdio.h>
#include <stdlib.h>
int main(){
 float *v; int n;
 printf("Dimensione: ");
 scanf("%d",&n);
 v = (float*) malloc(n*sizeof(float));
   ... uso dell'array ...
 free(v);v=NULL;
}
```

malloc() e free() sono dichiarate in **stdlib.h**

AREE DINAMICHE: TEMPO DI VITA

Tempo di vita di una variabile dinamica *non* è *legato a quello delle funzioni*

- in particolare, non è legato al tempo di vita della funzione che l'ha creata

Quindi, *una area dati dinamica può sopravvivere anche dopo che la funzione che l'ha creata è terminata*

Ciò consente di

- creare un'area dinamica in una funzione...
- ... usarla in un'altra funzione...
- ... e distruggerla in una funzione ancora diversa

ESERCIZIO 2

- Scrivere una funzione che, dato un intero, **allochi e restituisca una stringa di caratteri della dimensione specificata**

```
#include <stdlib.h>
char* alloca(int n){
  return (char*) malloc(n*sizeof(char));
}
```

- Il cliente:

```
char *p;
p=alloca(12);
```

- NOTA: dentro alla funzione *non* deve comparire la **`free()`**, in quanto scopo della funzione è proprio ***creare un array che sopravviva alla funzione stessa***

ESERCIZIO 2 - CONTROESEMPIO

Scrivere una funzione che, dato un intero,
allochi e restituisca una stringa di caratteri della dimensione specificata

Cosa non si può fare in C:

```
#include <stdlib.h>
  char* alloca(int n){
     char v[n]; /* n deve essere definito */
     return v;
  }
```

ARRAY DINAMICI

- **Un array ottenuto per allocazione dinamica è "dinamico" poiché *le sue dimensioni possono essere decise al momento della creazione*, e non staticamente a priori**
- *Non significa che l'array possa essere "espanso" secondo necessità:* una volta allocato, l'array ha dimensione *fissa*
- **Strutture dati espandibili dinamicamente secondo necessità esistono, ma *non sono array*** (*vedi* lezioni successive su ***liste, code …***)

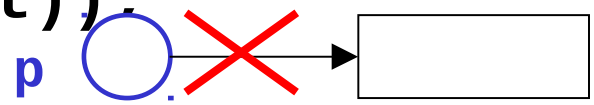
DEALLOCAZIONE - NOTE

- Il modello di gestione della memoria dinamica del C richiede che ***l'utente si faccia esplicitamente carico*** anche della ***deallocazione della memoria***
- ***È un approccio pericoloso:*** **molti errori sono causati proprio da un'errata deallocazione**
  - rischio di puntatori che puntano ad aree di memoria ***non più esistenti*** → ***dangling reference***
- **Altri linguaggi (ad esempio Java) gestiscono automaticamente la deallocazione tramite *garbage collector***

Dangling Reference

- Possibilità di fare riferimento ad aree di memoria heap non più allocate
- Esempio:

```
int *p;
p = (int *) malloc(sizeof(int));
...
free(p);p=NULL;

*p = 100;       /*  Da non fare!  */
```



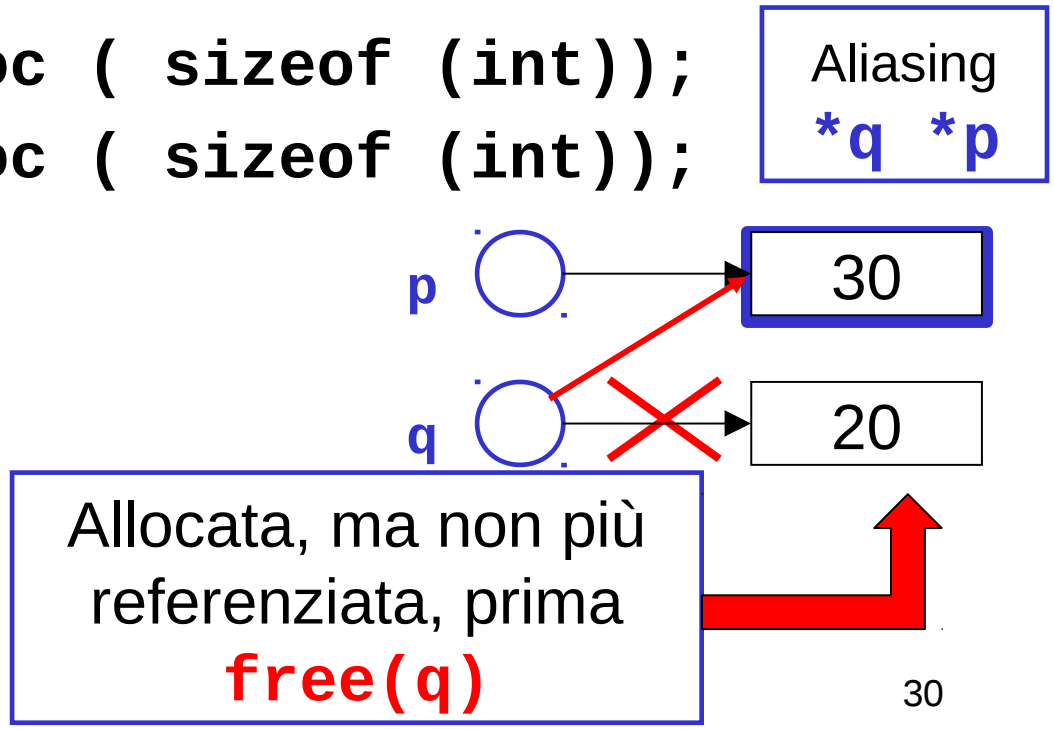
Garbage Collection

- Nei moderni linguaggi di programmazione, la deallocazione della memoria è gestita da un opportuno algoritmo
- Esistono ***sistemi automatici di recupero della memoria allocata ma non più usata*** → allocazione esplicita, ***deallocazione automatica***
- Il sistema sa sempre quanti e quali puntatori puntano ad una certa area di memoria → quando un'area di memoria non è più puntata da nessuno, viene recuperata tramite opportuna deallocazione
- Più facile per il programmatore, ma il recupero della memoria richiede una sospensione dell'esecuzione del programma → inadatto a sistemi real time.

Aree inutilizzate

- Possibilità di perdere il riferimento ad aree di memoria heap allocate
- Esempio:

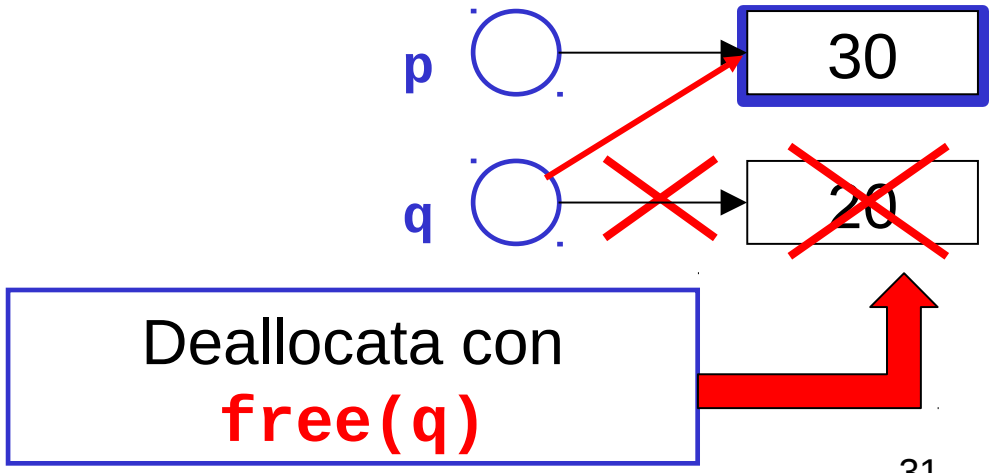
```
int *p,*q;
p = (int *) malloc ( sizeof (int));
q = (int *) malloc ( sizeof (int));
*p = 30;
*q = 20;

q = p;
```



Come evitare aree inutilizzate

- Possibilità di perdere il riferimento ad aree di memoria heap allocate
- Esempio:

```
int *p,*q;
p = (int *) malloc ( sizeof (int));
q = (int *) malloc ( sizeof (int));
*p = 30;
*q = 20;
free(q);
q = p;
```



Esempio con free e dangling reference

- Esempio, rivisto:

```
int *p,*q, *r;
p = (int *) malloc ( sizeof (int));
q = (int *) malloc ( sizeof (int));
r=q;
*p = 30;
*q = 20;
free(q);
q = p;
*r=10;
```