

# ALLOCAZIONE DELLA MEMORIA

---

Allocazione della memoria: quando un blocco di memoria RAM viene assegnato ad un programma in esecuzione.

In C abbiamo tre modalità di allocazione della memoria:

- **Allocazione automatica.** Riguarda le variabili (non `static`) con scope locale. La memoria viene automaticamente allocata sullo stack (record di attivazione) e automaticamente deallocata al termine della funzione in cui è dichiarata.

# ALLOCAZIONE DELLA MEMORIA

---

- **Allocazione statica.** Riguarda le variabili globali e le variabili locali dichiarate `static`. La memoria viene allocata prima dell'esecuzione del programma e rilasciata al termine.
- **Allocazione dinamica.** Memoria richiesta a runtime esplicitamente dal programmatore (mediante specifiche funzioni di libreria). Viene allocata dinamicamente nella memoria heap a cui si può accedere solo mediante puntatori. Deve anche essere rilasciata esplicitamente.

# ALLOCAZIONE DINAMICA

---

- Se le variabili sono **definite *staticamente***
    - la loro esistenza deve essere prevista e dichiarata a priori
  - Questo può rappresentare un problema soprattutto ***per variabili di tipo array, in cui dover specificare a priori le dimensioni (costanti) è particolarmente limitativo***
- ➔ Sarebbe molto utile poter ***dimensionare un array “al volo”, dopo aver scoperto quanto grande deve essere***

# 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”**

# ALLOCAZIONE DINAMICA

---

Per chiedere nuova memoria o rilasciarla “al momento del bisogno” si usano opportune funzioni di libreria i cui prototipi sono dichiarati nel file header `stdlib.h` e che “girano” la richiesta al sistema operativo:

**`malloc` e `free`**

# LA FUNZIONE `malloc`

---

La funzione `void * malloc(size_t dim)`:

- chiede al sistema di allocare un'area di memoria contigua grande *dim byte*
- *size\_t rappresenta un tipo di libreria del tipo unsigned (long) int (in dipendenza dall'implementazione)*
- *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
- grazie all'equivalenza fra array e puntatori possiamo accedere alla memoria allocata dinamicamente come se fosse un array (allocato quindi dinamicamente).

# LA FUNZIONE `malloc`

---

Il valore di ritorno è un puntatore all'inizio di un'area di memoria la cui dimensione è definita dall'argomento della **`malloc`**

Il sistema **ricorda, per ogni singola allocazione**, quanta memoria è stata allocata

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

Attenzione: Può essere trasformato in qualsiasi tipo di puntatore, anche in un tipo la cui dimensione non corrisponde al tipo allocato.



## ESEMPIO

---

- Per allocare dinamicamente 12 byte:

```
float *p;
```

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

- Per farsi dare *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

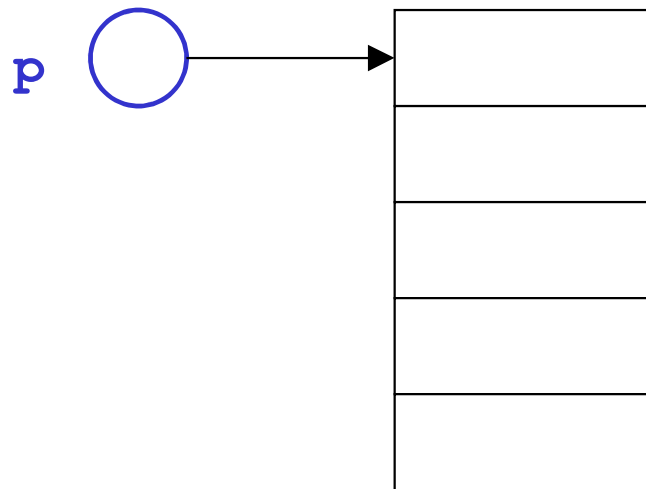
---

**Allocazione:**

```
int *p;
```

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

**Risultato:**



Sono cinque celle contigue,  
adatte a contenere un int

# AREE DINAMICHE: USO

---

L'area allocata è usabile, in maniera equivalente:

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

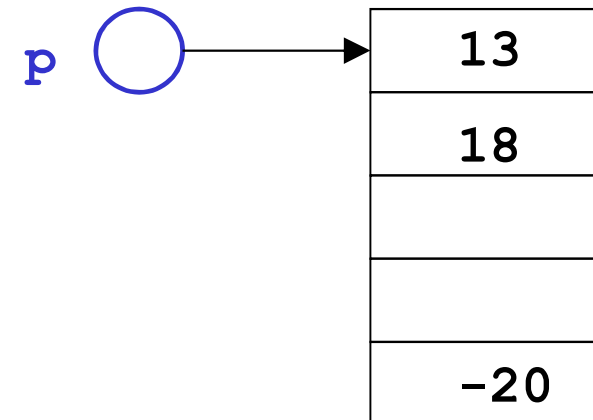
```
int *p;
```

```
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:

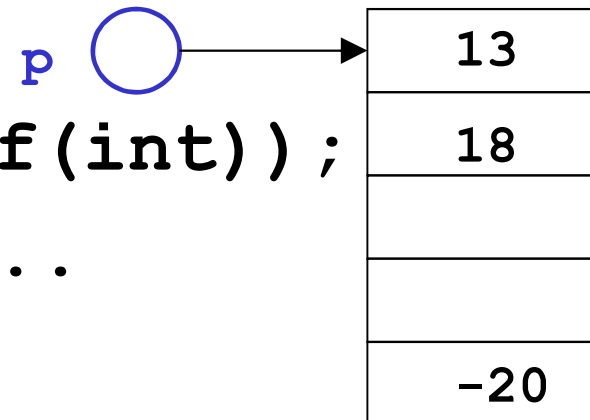
- *non sono determinate a priori*
- *possono essere scelte dal programma in base alle esigenze del momento*
- L'espressione passata a `malloc()` può infatti contenere variabili

```
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`

---

- Deallocazione tramite la procedura:  
`void free(void* p) ;`
- Consente di rilasciare la memoria allocata dinamicamente che si rende disponibile per eventuali nuove allocazioni.
- E' buona norma farlo per non saturare la memoria disponibile.
- Il sistema sa quanta memoria deallocare per quel puntatore (**ricorda** la relativa `malloc`).
- Se la memoria non viene correttamente deallocata → ***memory leaking (consumo di memoria).***

## AREE DINAMICHE: TEMPO DI VITA

---

**Tempo di vita di una area dati 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 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);
}
```

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



## 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));
}
```

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**

***Che cosa invece non si può fare in C:***

```
#include <stdlib.h>
char* alloca(int n) {
    char v[n];
    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 per forza 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, pile, code, ...*)

# Deallocazione della memoria

---

- A differenza di altri linguaggi di programmazione il linguaggio C non dispone di `garbage collector`.
- 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.***
- Se perdiamo il riferimento (indirizzo del primo byte) alla memoria allocata non potremo più accederci e questa resterà inutilizzabile (garbage) per tutta la durata del programma.<sup>20</sup>

## 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 gestiscono automaticamente la deallocazione tramite ***garbage collector***

```

#include <stdio.h>
#include <stdlib.h>
int f(void)
{char* s;
  s = malloc(50); /* ottieni memoria */
  if (s==NULL) return 1; /* non è disponibile */
  else
  { /* è disponibile */
    return 0; /* memory leak - vedi la nota sotto */
  }
  /*
   * La memoria era disponibile e puntata da s, ma non
   * salvata. Dopo questo ritorno di funzione il puntatore
   * viene distrutto e la memoria diventa irraggiungibile
   *
   * Per correggere il codice bisogna aggiungere l'istruzione
   * "free(s) all'else prima di "return 0"
   */
}

int main(void)
{
  /* questo è un loop infinito che richiama il metodo di sopra */
  while (1) f(); /* Questa funzione fallirà prima o poi */
  return 0;
}

```

# Garbage Collection

---

- Nei moderni linguaggi di programmazione, la deallocazione della memoria non è più un problema
- 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

# Reference Counting

---

- Utilizzo di un “semplice” sistema di gestione della memoria, il *reference counting (rc)*, basato sul conteggio “semiautomatico” del numero di puntatori che puntano ad una certa struttura dati
- Ogni volta che un nuovo puntatore punta alla struttura dati, ***incremento di rc*** per la struttura
- Ogni volta che un puntatore smette di puntare alla struttura dati, ***decremento di rc*** per la struttura
- In ogni momento si ha il controllo sul numero di puntatori che puntano alla struttura dati. Se tale numero è zero, la struttura non è puntata da nessuno e quindi è possibile deallocarla.



## Gli esercizi precedenti

---

Come cambiano gli esercizi già visti se si tiene conto della possibilità di allocare la memoria in modo dinamico?

- Merge Sort
- Rubrica
- Lettura da files
- .....