

## ALLOCAZIONE STATICA: LIMITI

---

- Per quanto sappiamo finora, in C le variabili sono sempre dichiarate *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

---

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

*la funzione di libreria `malloc( )`*

La funzione `malloc( )`:

- chiede al sistema di allocare **un’area di memoria grande *tanti byte quanti*** ne desideriamo
- ***restituisce l’indirizzo*** dell’area di memoria allocata

## LA FUNZIONE `malloc()`

---

La funzione `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:

```
int *p;
```

```
p = (int*) 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));
```

L'operatore `sizeof` consente di essere indipendenti dalle scelte dello specifico compilatore e della specifica macchina

## ESEMPIO

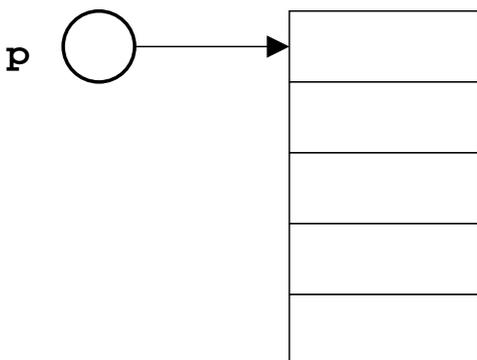
---

**Allocazione:**

```
int *p;
```

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

**Risultato:**



Sono cinque celle adatte a contenere un int

## AREE DINAMICHE: USO

L'area allocata è usabile, in maniera equivalente:

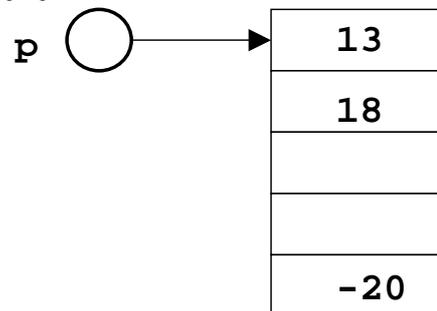
- o tramite la notazione a puntatore ( `*p` )
- o tramite la notazione ad array ( `[ ]` )

```
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 c'è alcun controllo

## AREE DINAMICHE: USO

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

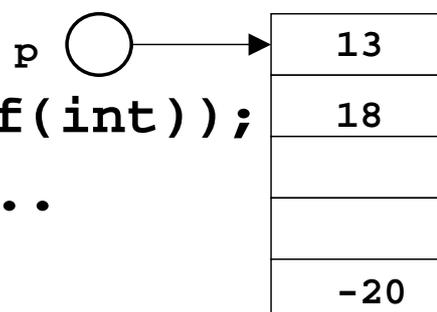
- *non sono determinate a priori*
- *possono essere scelte dal programma in base alle esigenze del momento*
- L'espressione passata a `malloc()` può infatti contenere variabili

```
int *p;
```

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

## AREE DINAMICHE: TEMPO DI VITA

---

- Il 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 variabile 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>
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 e alberi*)

## 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*