ALLOCAZIONE STATICA: LIMITI

- Per quanto sappiamo finora, in C le variabili sono sempre **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 "al momento del bisogno" si usa una funzione di libreria che "gira" la richiesta al sistema operativo:

`malloc()`

La funzione `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()`

La funzione `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***

`malloc()`

void* malloc(size_t dim);

- Il valore di ritorno è un puntatore senza tipo (**void***) → un indirizzo 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
- **Attenzione al void*** Può essere trasformato tramite *cast* in qualsiasi tipo di puntatore → anche in un tipo la cui dimensione non corrisponde al tipo allocato → **PERICOLOSISSIMO!**

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));
```

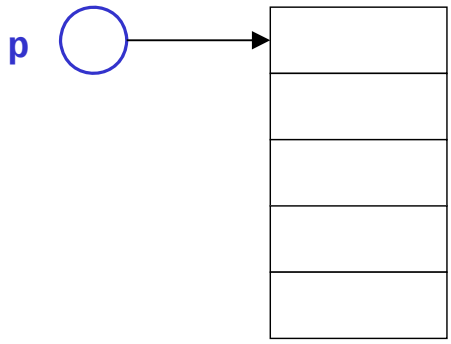
sizeof consente di essere indipendenti dalle scelte dello specifico compilatore/sistema di elaborazione

ESEMPIO

Allocazione:

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

Risultato:



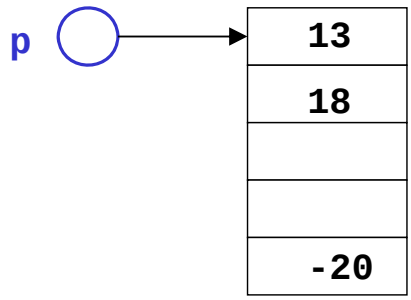
Sono cinque celle contigue, adatte a contenere un int

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;
```



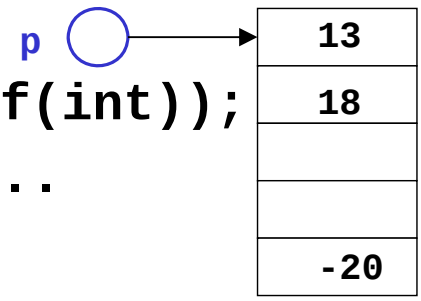
Attenzione a non "eccedere" l'area allocata dinamicamente. Non ci può essere alcun controllo

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*

`free()`

- Deallocazione tramite la procedura:
  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***

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 - 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***

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

- Nel nostro piccolo si può ipotizzare di utilizzare un "semplice" sistema di gestione della memoria, il *reference counting*...

- 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 reference count*** per quella struttura
- Ogni volta che un puntatore smette di puntare alla struttura dati, ***decremento di reference count*** per quella struttura

- In ogni momento si ha il controllo sul numero di puntatori che puntano alla struttura dati – se tale numero è maggiore di zero, la struttura non è puntata da nessuno e quindi è possibile deallocarla

Reference Counting

Come implementare un meccanismo di *reference counting*?

- Predisporre una struttura dati globale che rappresenti una tabella di due colonne e *n* (quante?) righe
- Nella prima colonna sono memorizzati ***gli indirizzi "puri" delle aree di memoria allocate***
- Nella seconda colonna sono memorizzati i ***reference count "rilasciati"***
- Per completare il tutto servono tre funzioni/procedure:
  - Una per **allocare** la memoria e impostare a **uno** il *reference count*
  - Una per copiare un puntatore in un altro → **incrementando** conteggio relativo a un indirizzo dato
  - Una per eliminare un "riferimento" → **decrementando** conteggio relativo a un indirizzo dato

Reference Counting

- Com'è implementata la tabella?
- Com'è implementata la ricerca nella tabella?
- ...e le "regole" di utilizzo?
- L'interfaccia:

```
void* referenceMalloc (size_t size);
void* copyReference(void* p);
void releaseReference(void *p);
```

…

Gli esercizi precedenti

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

- Es Merge Sort
- Rubrica
- Lettura da files