Puntatori e allocazione dinamica della memoria

Array: alcune osservazioni

- L'array nel main ha una dimensione massima (detta ***cardinalità***) e una dimensione utilizzata in un certo momento (detta ***riempimento***)
  - **Utile nel caso in cui il *riempimento* può variare a causa di operazioni di inserimento/rimozione di elementi**
- Dobbiamo prevedere però una cardinalità molto grande, sufficiente per tutte le esecuzioni del programma
  - **Anche nel caso in cui nell'array del nostro programma non si prevede di inserire/rimuovere dinamicamente elementi …**
  - **… con evidente spreco di memoria**
- Come facciamo ad allocare solo la memoria che effettivamente ci serve per un array ?

Puntatori e allocazione dinamica della memoria

- Il C supporta l'***allocazione dinamica della memoria***:
  - Possiamo allocare la memoria durante l'esecuzione del programma e assegnare l'indirizzo del blocco di memoria allocato ad un puntatore
  - **Questo permette di creare strutture dati la cui dimensione varia (aumenta o diminuisce) durante l'esecuzione, in funzione delle necessità**
- L'allocazione dinamica della memoria viene usata spesso per stringhe, array e strutture
- Per allocare dinamicamente la memoria esistono delle funzioni specifiche …

Allocazione dinamica della memoria

- Il file `<stdlib.h>` dichiara tre funzioni per l'allocazione dinamica della memoria:
  - `void *malloc(size_t size);`
    - Alloca un blocco di memoria di `size` bytes senza inizializzarlo e restituisce il puntatore al blocco
    - `size_t` è un tipo intero senza segno definito nella libreria del C
  - `void *calloc(size_t nelements, size_t elementSize);`
    - Alloca un blocco di memoria di di `nelements * elementSize` bytes e lo inizializza a 0 (clear) e restituisce il puntatore al blocco
  - `void *realloc(void *pointer, size_t size);`
    - Cambia la dimensione del blocco di memoria precedentemente allocato puntato da `pointer`
    - Restituisce il puntatore ad una zona di memoria di dimensione `size`, che contiene gli stessi dati della vecchia regione indirizzata da pointer (troncata alla fine nel caso la nuova dimensione sia minore di quella precedente).

Puntatori nulli

- Nel caso in cui non sia possibile allocare la quantità di memoria richiesta la funzione di allocazione restituisce un ***puntatore nullo***
- Un puntatore nullo è un puntatore che ha un valore speciale (macro `NULL`) che si può distinguere da di tutti i valori possibili dei puntatori validi
- Quindi, dopo aver chiamato una funzione di allocazioni dinamica della memoria, dobbiamo controllare se il puntatore restituito dalla funzione è valido oppure è nullo

Liberare la memoria allocata dinamicamente

- Lo spazio di memoria allocato in maniera automatica viene anche liberato automaticamente, all'uscita dalla funzione …

- Lo spazio allocato dinamicamente **non viene deallocato** all'uscita dalla funzione, ma deve essere liberato manualmente quando non è più necessario

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

- Ne faremo uso con l'implementazione delle strutture dati …

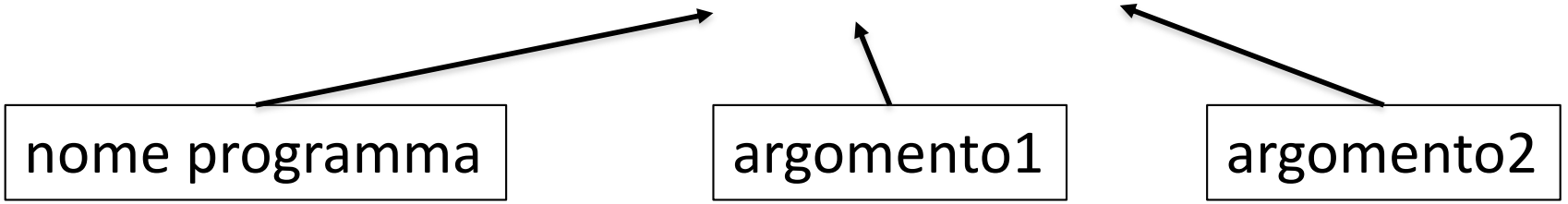
Rivediamo il main del programma per ordinare un array

```
# include <stdio.h>
# include <stdlib.h>
# include "vettore.h"

int main()
{
    int *a, n;

    printf("Inserisci il numero di elementi da ordinare: ");
    scanf("%d", &n);

    a = (int*) calloc(n, sizeof(int));   // in alternativa a = malloc(n*sizeof(int))
    if(a == NULL)
        printf("Memoria insufficiente \n");
    else {
        input_array(a, n);
        ordina_array(a, n);
        printf("Elementi ordinati \n");
        output_array(a, n); }
}
```

Un'alternativa: argomenti sulla linea di comando

- Il `main` è una funzione …
  - invocata dal sistema operativo
  - che può ricevere in input un array di stringhe
  - che corrispondono agli argomenti scritti dall'utente sulla linea di comando
- Esempio di invocazione da linea di comando:

```
MacBook:~ adelucia1$ gcc -c ordina_array.c
```



Argomenti sulla linea di comando

- Il `main` ha due parametri
  - Argument counter: `int argc`
    - Conta il numero di argomenti su linea di comando
  - Argument vector: `char* argv[]`
    - l'array di stringhe corrispondenti agli argomenti
    - NB: argv[0] è il nome del programma
  - Quindi, l'interfaccia completa del main è la seguente
    - `int main(int argc, char* argv[])`
  - Se non servono, argc e argv possono essere omessi:
    - `int main()`

Rivediamo il nostro esempio

```
# include <stdio.h>
# include <stdlib.h>
# include "vettore.h”

// Invocazione programma: nome_programma numero_elementi

int main(int argc, char *argv[])
{
    if(argc < 2)
        printf("Numero di elementi da ordinare mancante \n");
    else {
        int n = atoi(argv[1]);
        int *a = (int*) calloc(n, sizeof(int));
        if(a == NULL)
            printf("Memoria insufficiente \n");
        else {
            input_array(a, n);
            ordina_array(a, n);
            printf(“Elementi ordinati \n”);
            output_array(a, n); }
        }
}
```

Rivediamo alcuni concetti sull'allocazione della memoria in C

- Finora abbiamo assunto che le variabili sono dichiarate all'interno di funzioni
  - **una tale variabile è detta *locale* ed è visibile solo all'interno della funzione in cui è dichiarata**
  - **è possibile dichiarare dati anche in blocchi più interni ...**
  - **tali variabili sono dette *automatiche*, perché vengono allocate in memoria a tempo di esecuzione (dell'istruzione dichiarativa) e deallocate al termine del blocco in cui sono dichiarate**
- Esistono anche variabili globali, dichiarate esternamente alle funzioni ...

Variabili globali

- Una variabile globale è visibile a tutte le funzioni la cui definizione segue la dichiarazione della variabile nel file sorgente
  - tali variabili sono dette *statiche*, perché la loro allocazione in memoria avviene all'atto del caricamento del programma (e la loro deallocazione al termine del programma)
- Le variabili globali possono essere usate per scambiare informazioni tra sottoprogrammi …
  - vedremo in seguito come è possibile usarle in maniera corretta nell'implementazione di astrazioni dati …

Scope, Visibilità e Durata

… tre aspetti importanti di una dichiarazione

- ***Scope***: parte del programma in cui è attiva una dichiarazione (dice quando può essere usato un identificatore)
- ***Visibilità***: dice quali variabili sono accessibili in una determinata parte del programma (non sempre coincide con lo scope …)
- ***Durata***: periodo durante il quale una variabile è allocata in memoria

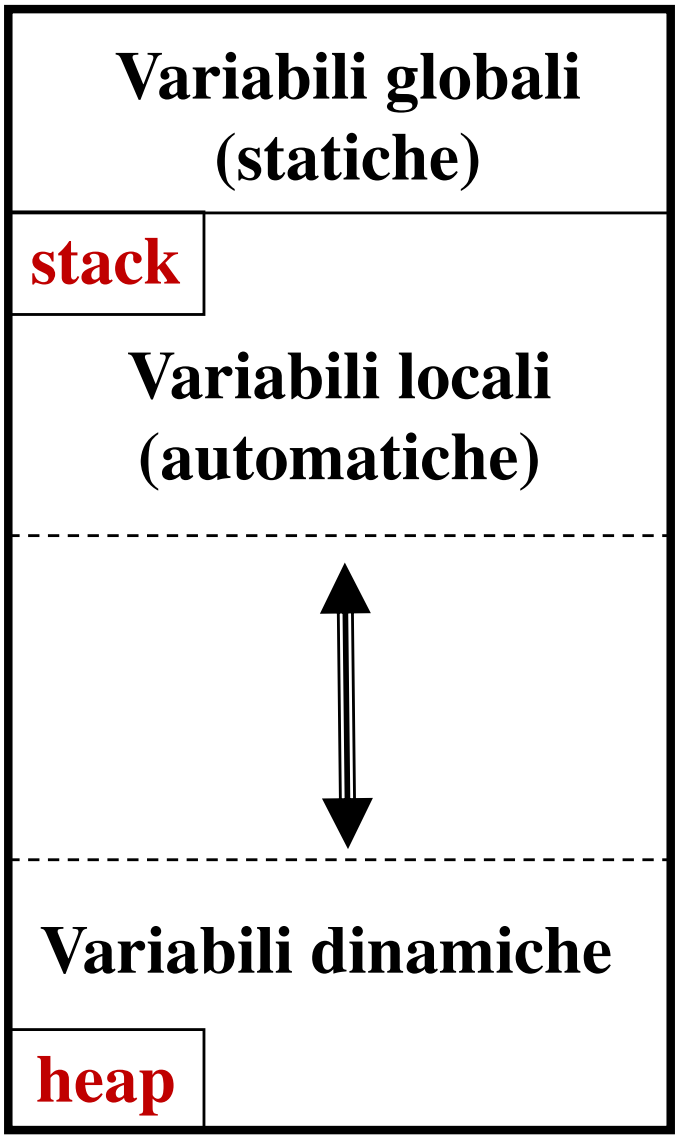
Scope e Visibilità: un esempio

```
int n;
…
int main()
{
   long n;
   …
  {
     double n;
     …
  }
  …
}
```

- int n
  - ***scope*: intero file**
  - ***visibilità*: non è visibile nel main**
- long n
  - ***scope*: main**
  - ***visibilità*: non è visibile nel blocco interno**
- double n
  - ***scope*: blocco interno**
  - ***visibilità*: coincide con lo scope**

Durata: tre aree di allocazione

- Variabili globali (statiche)
  - area di memoria fissata
- Variabili locali (automatiche)
  - allocate all'atto dell'esecuzione del blocco in cui sono dichiarate (e deallocate alla fine dell'esecuzione del blocco)
- Variabili dinamiche
  - operazioni su puntatori per allocazione e deallocazione dinamica della memoria



Dichiarazioni static di funzioni

- Servono a realizzare l'information hiding ...
- In generale una funzione definita in un file .c è utilizzabile in un altro file, anche in assenza della dichiarazioni esplicita del prototipo nell'header file …
  - **ad esempio, per usare una funzione definita in un altro file, basta dichiararne prima il prototipo**
- Dichiarazioni ***static*** di funzioni le rendono private al file in cui sono dichiarate, ossia ne modificano lo ***scope***
  - **Nel modulo vettore, la funzione minimo_i è usata solo localmente al modulo dalla funzione ordina_array**
  - **Per renderla privata al modulo basta aggiungere la dichiarazione static:**

```
static int minimo_i(int a[], int i, int n);
```

Rivediamo il modulo vettore

vettore.h

```
void input_array(int a[], int n);
void output_array(int a[], int n);
void ordina_array(int a[], int n);
int ricerca_array(int a[], int n, int elem);
int minimo_array(int a[], int n);
…
```

vettore.c

```
#include <stdio.h>
#include “utile.h” // contiene funzione scambia

static int minimo_i(int a[], int i, int n); // dichiarazione locale


void input_array(int a[], int n) { … }
void output_array(int a[], int n) { … }
void ordina_array(int a[], int n) { … }
int ricerca_array(int a[], int n, int elem) { … }
int minimo_array(int a[], int n) { … }
static int minimo_i(int a[], int i, int n) { … } // usata da ordina_array
…
```

Dichiarazioni static di variabili globali

- Un discorso analogo può essere fatto per le variabili globali
  - **per usare variabili globali definite in un file F1 basta dichiararle come *extern* nel file F2 in cui si vuole usarle …**
    - ***esempio*: extern int x;**
  - **NB: con la dichiarazione extern non si definisce la variabile, non si alloca memoria …**
- Dichiarazioni ***static*** di variabili globali le rendono private al file in cui sono dichiarate, cambiandone lo ***scope***
    - ***esempio*: static int x;**
  - **Ne vedremo l'uso con l'implementazione di astrazioni dati …**
  - **Forse la scelta della parola chiave static qui non è felice …**

Dichiarazioni static di variabili locali

- Attenzione: in C è possibile anche la dichiarazione ***static*** di variabili locali
- La dichiarazione ***static*** di una variabile locale cambia la classe di allocazione della variabile
  - **Trasforma una variabile automatica in una variabile statica …**
  - **Cambia quindi la durata della variabile, non lo scope che rimane confinato alla funzione (o blocco) in cui la variabile è dichiarata**
- Come effetto, il valore di una variabile "sopravvive" all'esecuzione della funzione in cui è stato definito **…**
  - **non approfondiremo l'uso di dichiarazioni static di variabili locali**