LE FUNZIONI

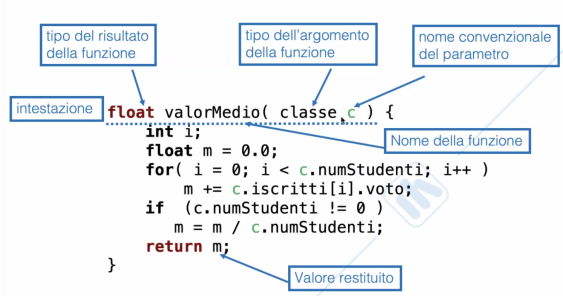
- Problemi: 1) come rendere il codice di programmi complessi più modulare e leggibile? 2) come riutilizzare la funzionalità di codice già scritto? 3) come condividere un codice implementato con altri?

- Principio del *dividi et impera*: un programma complesso viene suddiviso in sottoprogrammi semplici con obiettivi specifici e definiti, rendendo il codice più leggibile e riutilizzabile. come?
  1- modularità: affrontare il codice per raffinamenti successivi
  2- riutilizzabilità: scrivere il codice una sola volta e riutilizzarlo
  3- astrazione: assegnare operazioni specifiche per i tipi di dato definiti dal programmatore

- alcune funzioni sono già implementate in librerie (es <stdio.h>, <string.h>), mentre altre possono essere definite e personalizzate dal programmatore.

- le funzioni sono sottoprogrammi definiti su parametri generici facilmente riutivizzabili in un programma, che possono poi essere invocate per l'utilizzo su parametri specifici. Le funzioni vengono sempre eseguite in ordine di invocazione.

```c
#define MAXSTUDENTI 300;

typedef char stringa[20];
typedef struct {
     int giorno, mese, anno; } data;
typedef struct {
     int matricola, voto;
     stringa   nome, cognome;
     data      dataNascita;  } descrizioneStudente;
typedef struct {
     int   numStudenti;
     descrizioneStudente iscritti[MAXSTUDENTI];  } classe;

classe InfoA, Analisi;
float mediaInfoA, mediaAnalisi;

…
mediaFondamenti = valorMedio(InfoA);
mediaAnalisi = valorMedio( Analisi );
```



- una funzione può essere: implementata e definita prima del programma oppure può essere dichiarata prima del main tramite la sua intestazione (nome, ritorno, variabili) e definita dopo il programma.
  [**dichiarazione**-main(**invocazione**)-**definizione**]
  Quando si vuole invocare una funzione per l'utilizzo bisogna conoscerne il nome e i parametri necessari.

- Struttura di una funzione in C:

```c
#include <stdio.h>

#define DIM 5

int ottieni_max (int []);                    // dichiarazione funzione
int main() {

  int elenco[DIM];
  int max;

  printf("Input 5 numeri interi: ");       // attenzione: no controllo
  scanf("%d %d %d %d %d",
        &elenco[0], &elenco[1], &elenco[2], &elenco[3], &elenco[4]);

  max = ottieni_max(elenco);                 // uso funzione >> produce valore

  printf("Il massimo è %d.\n\n", max);     // stampa

  return 0;
}

int ottieni_max (int valori[]) {           // definizione funzione
  int max = 0;
  for (int i=0; i<DIM; i++)
    if (valori[i] > max)
      max = valori[i];
  return max;                      // ritorno di valore max cercato
}
```

Dichiarazione

Invocazione

Definizione

1) **dichiarazione** (o prototipo)
↓
int main {
2)**invocazione**}
↓
3)**definizione**

1) **dichiarazione**: all'inizio del documento, prima del main, bisogna inserire l'intestazione (header) delle funzioni: i prototipi, che specificano:
   - il nome della funzione
   - il tipo restituito
   - il tipo di ciascuno degli argomenti della funzione (non è necessario specificare i nomi per i parametri, se specificati saranno ignorati brutalmente)
     (es. in questo caso il nome è `ottieni_max`, il ritorno è `int`, il tipo di variabile viene specificato (`int[]`) anche senza il nome della variabile (→ che è però da esplicitare nella definizione))

```
int ottieni_max (int []);
```
Dichiarazione

2) **invocazione**:
   - attraverso l'inserimento del nome della funzione in qualsiasi punto del programma, specificando una variabile per ogni parametro dell'argomento della funzione, assegnando se necessario anche la variabile relativa all'output della funzione. (es. `somma = somma_interi(a,b);` `somma` è la variabile output)

3) **definizione**:
   - intestazione: tipo_di_ritorno nome_funzione (lista parametri con nome e tipo delle variabili esplicitati)
   - corpo della funzione: contiene le istruzioni che permettono alla funzione di svolgere il compito, tra parentesi graffe: è costruita con le stesse regole del main, può avere uno o più `return` ma ad ogni invocazione della funzione se ne esegue solo una (es. `if (a>b) return a; else return b`)

```
int max (int a, int b){
     if (a>b)
           return a;
     else
           return b;
}
```

   - valore di ritorno: return/void, esistono sottoprogrammi che restituiscono valori,
     se invece il sottoprogramma non restituisce valori allora è necessario specificarlo con il particolare tipo "void" che specifica l'assenza di valori di uscita; se il return è void allora posso non specificare il nome del ritorno
     (es. `return;`)

```
void error_line( int line ) {
      printf("Errore alla linea %d\n", line);
}
```

   - lista dei parametri: elenco separato da virgola che specifica tipo e nome di ogni parametro tra parentesi tonda dopo il nome della funzione, all'interno della sua definizione
     (es. `int valori[], etc...` è un parametro)

```
int ottieni_max (int valori[]) {          // definizione funzione
  int max = 0;
  for (int i=0; i<DIM; i++)
    if (valori[i] > max)
      max = valori[i];
  return max;                    // ritorno di valore max cercato
}
```
Definizione

13

- **parametri** formali e attuali:
nella definizione della funzione viene dato un nome convenzionale ai parametri ricevuti in ingresso (non necessario nella dichiarazione)→ **parametri formali**, cui non sono ancora associati valori;
al momento dell'invocazione ad ogni parametro viene assegnata una variabile o un'espressione → **parametro attuale**;
anche se parametri formale e attuali hanno lo stesso nome ciò non comporta un problema in quanto i parametri attuali vengono sempre copiati e duplicati.

- l'assegnamento dei valori avviene attraverso la **corrispondenza posizionale**, e tutti i tipi devono corrispondere (alternativamente vi è un casting implicito ma molto rischioso)

- **tipo del risultato** restituito dalla funzione: possono essere semplici built-in (int, char, double, float, void) o complessi user-defined (struct anche con array), possono essere puntatori a qualsiasi tipo, ma nb! NON possono essere array;
- **tipo dei parametri:** possono essere built-in o user-defined, i parametri attuali originali non sono modificabili dalle funzioni in cui passano, i sottoprogrammi lavorano su copie dei parametri attuali.
  NB! eccezione: gli array, il nome indica solo l'indirizzo della prima variabile per questo la funzione può modificarne il contenuto

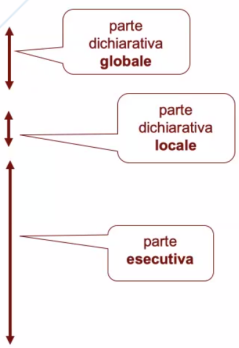
- caso particolare 1: **parametri di tipo array unidimensionale**: array possono essere passati alle funzioni come parametri ma è necessario specificare nella dichiarazione di tale funzione se un parametro è un array (int[]) e nell'invocazione è necessario specificare il nome dell'array. NB! nella dichiarazione di una funzione contenente array non si deve esplicitare la dimensione dell'array, tuttavia posso a) dichiarare la dimensione di un array come variabile globale o come costante (es `#define`), b) passo la dimensione all'interno della funzione come parametro

- caso particolare 2: **parametri di tipo array multidimensionale**: NB! in caso di matrici è necessario esplicitare nella definizione e nel prototipo (dichiarazione) della funzione la dimensione di tutti gli indici successivi al primo, perche, poichè gli array sono memorizzati in riga, il compilatore deve sapere quanti elementi ci sono in una riga per determinare la posizione di essi tramite gli offset (vv. aritmetica dei puntatori).

```c
#include <stdio.h>

#define DIM 5

int ottieni_max (int [], int);            // dichiarazione funzione

int main() {

  int elenco[5];
  int max;

  printf("Input 5 numeri interi: ");       // attenzione: no controllo
  scanf("%d %d %d %d %d",
        &elenco[0], &elenco[1], &elenco[2], &elenco[3], &elenco[4]);

  max = ottieni_max(elenco, 5);            // uso funzione >> produce valore

  printf("Il massimo è %d.\n\n", max);     // stampa

  return 0;
}

int ottieni_max (int valori[], int size) {         // definizione funzione
  int max = 0;
  for (int i=0; i<size; i++)  Uso la dimensione all'interno della funzione
    if (valori[i] > max)
      max = valori[i];
  return max;                        // ritorno di valore max cercato
}
```

- struttura di un programma con funzioni:

```
inclusione librerie
DICHIARAZIONE di variabili globali e funzioni
int main ( ){
        dichiarazione di variabili locali
        istruzione 1;
        istruzione 2;
        istruzione 3;
        ...
        istruzione N;
}
DEFINIZIONE di funzioni
```

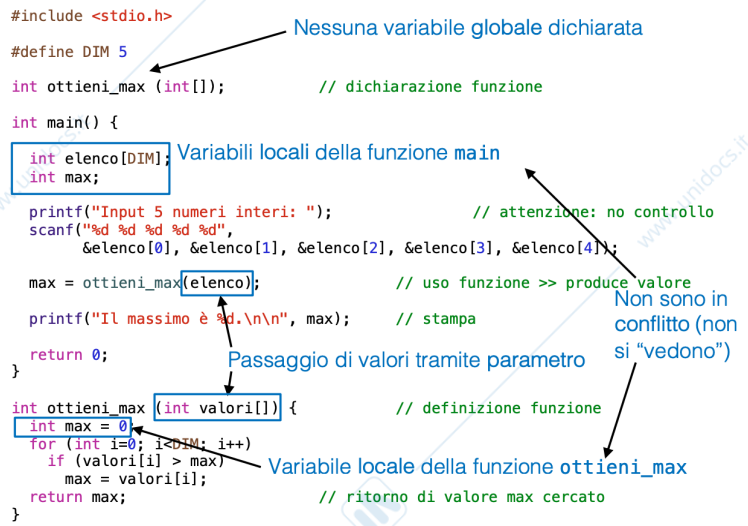


-

- variabili locali e globali:
  - **variabili globali**: variabili costanti in ogni parte del programma, dichiarate prima del main come tutte le altre variabili
  - **variabili locali**: sono visibili solo all'interno del sottoprogramma che le dichiara e nascono e muoiono insieme al sottoprogramma (es. se dichiarate nel main o nella definizione di una funzione);

  quindi ogni sottoprogramma può usare solo: var locali definite al suo interno, var globali, parametri della funzione (= a variabili locali provenienti dal main)

- le variabili non locali e non dichiararate all'interno di un sottoprogramma possono essere utilizzate solo se passate a una funzione come **argomenti**, percmettono la comunicazione tra ambiente chiamante e interno della funzione.



- passaggio di argomenti a funzioni: 1) per valore, 2) per riferimento
  1) il **passaggio per valore** si ottiene semplicemente inserendo nell'invocazione della funzione il nome di una variabile, la funzione ne copierà il contenuto in una variabile locale della funzione che verrà utilizzata, mentr la variabile originale rimarrà immutata (anche in caso di omonimie);
  2) il **passaggio per riferimento** si realizza tramite i puntatori, permette di evitare sovraccarichi e consente di modificare l'argomento nella funzione chiamante: se nell'invocazione un paramentro viene passato tramite il suo indirizzo tramite l'operatore & (es. `number=cubeByReference(&number)`), allora nel suo prototipo e nella sua definizione deve essere definito un parametro puntatore con operatore * in modo da poter ricevere l'indirizzo (es. `int cubeByReference(int *nPtr)`) e per utilizzare il valore all'interno della funzione si usa sempre l'operatore di dereferenziazione * (es. `*nPtr`).
     nb! il compilatore non distingue tra una funzione che riceve un array unidimensionale e una che riceve una variabile passata per indirizzo, per questo va specificato.

esempi: funzione di elevamento al cubo

passaggio per valore

```
int cubeByValue(int n){
  return n*n*n;
}
int main(void) {
  int number=5;
  number = cubeByValue(number);
  printf("il cubo è %d", number);
  return 0;
}
```

passaggio per riferimento

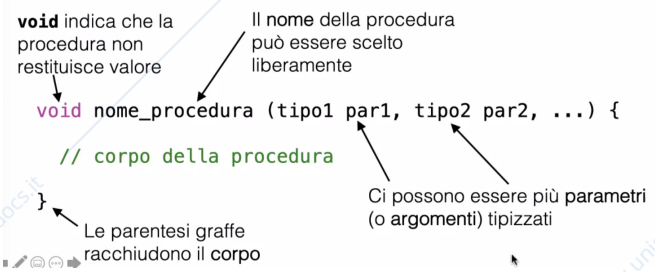
```
int cubeByReference(int *nPtr){
  return *nPtr * *nPtr * *nPtr;
}
int main(void) {
  int number=5;
  number = cubeByReference(&number);
  printf("il cubo è %d", number);
  return 0;
}
```

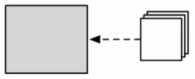
- **Pattern principali** di funzioni:

1- funzioni senza valore di ritorno
2- funzioni con valore di ritorno
3- funzioni con effetti collaterali

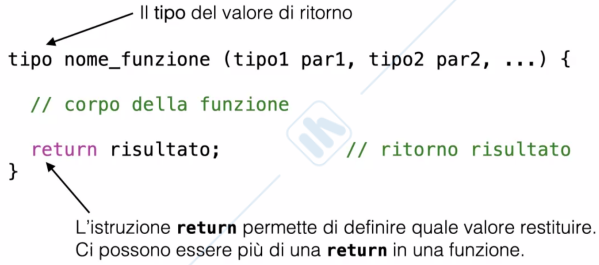
- **Pattern principali** di funzioni:

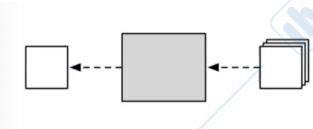
1- **funzioni senza valore di ritorno** o **procedure:** di tipo void, non restituiscono valori ne hanno effetti collaterali, l'ambiente che le invoca non viene modificato (es. somma_interi, Stampa_matrici, printf)



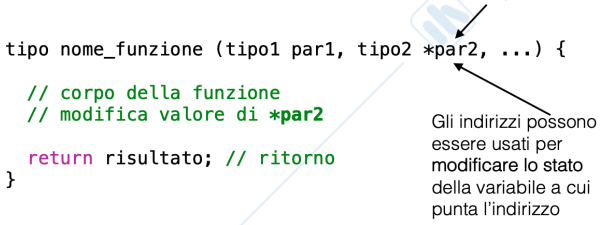


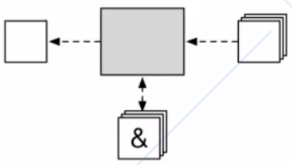
2- **funzioni con valore di ritorno**: producono un valore che può essere usato nell'ambiente chiamante per assegnare nuovi valori a variabili, può avere più return ma ne viene eseguito sempre solo uno per volta (es. occupazione_aule), nb! non possono restituire array.





3- **funzioni con effetti collaterali**: modificano lo stato di alcune variabili dell'ambiente chiamante in quanto per **passare le variabili si utilizzano gli indirizzi dei parametri tramite puntatori**, ogni cambiamento di tali variabili nella funzione si riflette sul main (normalmente verrebbe altrimenti passata per valore, ovvero verrebbe utilizzata una copia locale di tali variabili che rimarrebbero però intatte nel main), nella definizione il parametro deve essere specificato come puntatore (es. `int* p`) mentre nell' invocazione della funzione si deve passare l'indirizzo del parametro attuale da modificare (es. `&varLocale`), nel corpo della funzione si usa l'operatore "`*`" di deferenziazione per riferirsi al valore del parametro (es. `*p`). (es. scanf, somma_interi)
(NB! il parametro attuale deve essere una variabile o un'espressione che restituisce un puntatore, non una generica espressione che altrimenti non sarebbe associata a nessuno spazio di memoria)





- Intercambiabilità tra funzione e procedura, tramite l'utilizzo di puntatori: il ritorno viene sostituito da un nuovo parametro di tipo puntatore inserito negli argomenti della funzione

Funzione

```
int f(int par1) {
  ... (calcola valore
  della variabile
  "risultato") ...
  return risultato;
}
```

Chiamata:

```
y = f(x);
```

Procedura

```
void f(int par1, int *par2) {
  ... (calcola valore di
  variabile "risultato")
  ...
  *par2 = risultato;
}
```

Chiamata:

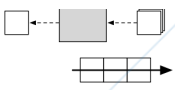
```
f(x, &y);
```

(es: funzione senza ritorno: stampa_matrice)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define DIM 10

void stampa_matrice (int[][DIM]); dichiarazione funzione


int main() {

  int matrice[DIM][DIM];

  srand((int)time(NULL));     // inizializzazione generatore numeri

  for (int i=0; i<DIM; i++)   // inizializzazione con num casuali
    for (int j=0; j<DIM; j++)
      matrice[i][j] = rand() % 100;

  stampa_matrice(matrice);// invocazione funzione di stampa

  return 0;
}

// definizione funzione di stampa
void stampa_matrice (int mat[][DIM]) {

  printf("Matrice:\n\n");

  for (int i=0; i<DIM; i++) {
    for (int j=0; j<DIM; j++)
      printf(" %3d", mat[i][j]);
    printf("\n");
  }
}
```

(es funzione con ritorno: occupazione_aule, vv le diverse versioni)

```c
#include <stdio.h>

#define NUM_AULE 20
#define NUM_ORE 24

typedef int occupazione_aule[NUM_AULE][NUM_ORE];

// dichiarazione funzioni
float calcola_media (int []);
int aula_piu_affollata (occupazione_aule);

int main() {

  occupazione_aule occ;              // + inizializzazione occ...

  printf("L'aula più affollata è l'aula numero %d.\n\n",
         aula_piu_affollata(occ));          // invocazione funzione

  return 0;
}

// definizione funzioni

int aula_piu_affollata (occupazione_aule occ) {

  int risultato = 0;
  float maxMedia = 0.0;

  for (int i = 0; i < NUM_AULE; i++) {      // per tutte le aule i
    if (calcola_media(occ[i]) > maxMedia) {         // tieni massimo
      maxMedia = calcola_media(occ[i]);
      risultato = i;
    }
  }
  return risultato;
}

float calcola_media (int elenco[]) {

  int somma = 0;                    // inizializzazione somma
  for (int j = 0; j < NUM_ORE; j++)
    somma += elenco[j];             // calcola totale di persone

  return ((float) somma) / NUM_ORE;                 // ritorna media
}
```



I due cicli originali ora sono contenuti tutti e due nelle due funzioni, la prima invocata dal `main`, la seconda dalla prima funzione

(es1. funzioni con effetti collaterali: somma_interi con puntatori, fz void

se inizializzo come come un intero: `int a,b,somma;` poi nell'invocazione devo passare l'indirizzo a tale intero con l'operatore di referenziazione & per passare l'indirizzo: `somma_interi(a,b,&somma);` e poi svolgere normalmente la funzione)

```c
#include <stdio.h>
void somma_interi(int addendo1, int addendo2, int *risultato);
       // dichiarazione funzione (i nomi dei parametri verranno
          ignorati)
int main() {
  int a, b, *somma;      somma è un puntatore a intero

  printf("Input 2 numeri interi: "); // attenzione: no controllo
  scanf("%d %d", &a, &b);
  somma_interi(a, b, somma);                    // uso funzione
  printf("La somma è %d.\n\n", *somma);              // stampa
  return 0;                          Per leggere il valore uso la deferenziazione
}

void somma_interi(int addendo1, int addendo2, int *risultato) {//
definizione funzione          Il parametro risultato è un puntatore a intero
  *risultato = addendo1 + addendo2;           // fine funzione
}  Per modificare il valore uso la deferenziazione
```

caso1:

```
int a,b,*somma;
}
somma_interi(a,b,somma);
```

a → addendo1, b → addendo2, somma → *risultato

caso2:

```
int a,b,somma;
}
somma_interi(a,b,&somma);
```

a → addendo1, b → addendo2, &somma → *risultato

(es2. funzione scambio)

```c
#include <stdio.h>

// dichiarazioni funzioni
void swap_sbagliato (int, int);
void swap_corretto  (int *, int *);

int main() {

  int a = 5, b = 9;

  printf("Originale: %d, %d\n", a, b);

  swap_sbagliato(a, b);
  printf("Dopo swap_sbagliato: %d, %d\n", a, b);

  swap_corretto(&a, &b);
  printf("Dopo swap_corretto: %d, %d\n\n", a, b);

  return 0;
}

// continua...

// passaggio parametri per valore
void swap_sbagliato (int primo, int secondo) {

  int temp;                          // variabile ausiliaria

  temp = primo;                      // scambio valori
  primo = secondo;
  secondo = temp;
}  I parametri sono passati per valore. La loro
   modifica non si riflette sul programma principale
//passaggio parametri per indirizzo
void swap_corretto (int *primo, int *secondo) {

  int temp;                          // variabile ausiliaria

  temp = *primo;                     // scambio valori
  *primo = *secondo;
  *secondo = temp;
}
```

(es3. somma di array)

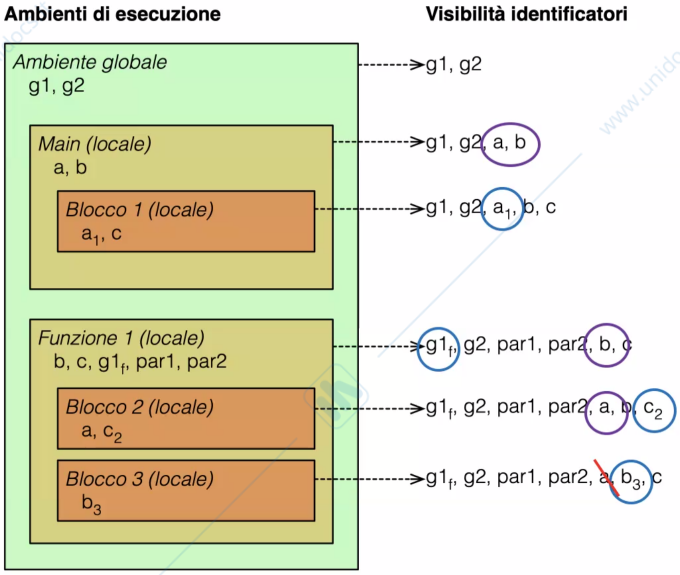
```c
#include <stdio.h>

// dichiarazioni funzioni
double sum_array1 (double [], int);
double sum_array2 (double *, int);

int main() {

  double a[6] = {1,2,3,4,5,6};
  double sum1, sum2;

  sum1 = sum_array1(a, 3);
  printf("Risultato somma con array: %d, %d\n", sum1);

  sum2 = sum_array2(a, 3);
  printf("Risultato somma con puntatore: %d, %d\n", sum2);

  return 0;
}


// funzione che somma i primi n elementi di un array

// funzione con array
double sum_array1 (double a[], int n) {
  int i;
  double ris = 0.0;
  for(i=0; i<n; i++)
    ris += a[i];
  return ris;
}

// funzione equivalente con puntatore
double sum_array2 (double *a, int n) {
  int i;
  double ris = 0.0;
  for(i=0; i<n; i++)
    ris += *(a + i);
  return ris;
}
```

Due notazioni diverse per definire la stessa funzione

Funzioni e modello esecuzione

- **Blocco**: è una porzione di codice definita da parentesi graffe {...} che contiene una dichiarazione di variabili locali (opzionale) e una sequenza di istruzioni. Possono essere:
  annidati: uno all'inderno dell'altro;
  paralleli: uno dopo l'altro all'interno di un terzo blocco;

- **Visibilità** (o campo d'azione) **di un identificatore**: condizioni per cui un identificatore (nome della variabile, funzione, tipo) può essere visto e quindi utilizzato. In base alla visibilità possono essere:
  globali: (→ variabili globali) visibili e accessibili in tutto il codice, sono tutti gli identificatori dichiarati fuori da una specifica funzione (es. prima del main);
  locali: (→ variabili locali) visibili e accessibili solo all'interno della funzione in cui sono stati dichiarati, e in tutti i blocchi annidati in esso → il blocco annidato ha la vista del blocco padre + la propria, nb! la dichiarazione di identificatori omonimi a identità esterne maschera la visibilità di tali entità omonime esterne al blocco.



(se in una funzione e nel main utilizzo identificatori con lo stesso nome non ho problemi)

(NB! la a nel blocco 3 è un errore! il blocco 3 non vede a perchè è parallelo al blocco 2 e non annidato)

- ciclo di vita delle variabili: passaggi di gestione di una variabile dalla creazione (allocazione in memoria) alla sua distruzione (rilascio dello spazione di memoria).
  E' possibile in C specificare la classe di memoria di una variabile che determina la durata della sua permanenza in memoria, classi:
  - **variabili statiche**: vengono create e allocate in memoria una solo volta quando viene creato il programma, perdurano per tutta la durata di esso e vengono distrutte alla sua fine, sono statiche variabili e costanti globali (identificatori esterni) e le variabili locali forzate con il modificatore `static`, esse comunque rimangono conosciute solo nel blocco in cui sono state definite ma mantengono il loro valore anche alla fine dell'esecuzione della funzione (es. `static int count=1`);
    le variabili statiche globali possono essere utilizzare per comunicare tra diverse funzioni (poco elegante), se modificate in una funzione allora vengono modificate in tutto il programma;
    le variabili locali static possono fungere da canale di comunicazione tra invocazioni multiple di una stessa funzione in quanto mantengono il loro valore dopo il termine di ogni funzione;

  - **variabili automatiche** (o dinamiche): sono memorizzate sullo stack, durano soltanto il tempo di esecuzione del blocco in cui sono state dichiarate e quindi sono visibili e sono distrutte al suo termine, servono per ottimizzare la memoria (es. contatori nel `for` ma anche le variabili locali del main);

(es. comunicazione tra variabili globali: se esse vengono modificate in una funzione, allora rimangono modificate in tutto il codice in quanto si accede alla variabile non di un parametro di cui è stata creata una copia)

```c
int var_globale = 3;                        // variabile globale

void funzione1() {                    // funzione 1 scrive valore
  var_globale = 5;
  printf("Funzione 1: %d\n", var_globale);
}

void funzione2() {                          // funzione 2 legge valore
  printf("Funzione 2: %d\n", var_globale);
}

int main () {

  printf("Main: %d\n", var_globale);

  funzione1();
  funzione2();

  return 0;
}
```

Output:

```
Main: 3
Funzione 1: 5
Funzione 2: 5
```

(es. variabili locali statiche: a) nel primo caso la variabile locale dichiarata nella funzione è zero e viene stampata prima dell'incremento che non si noterà in quanto ogni volta che verrà ri-invocata la funzione la variabile satà sempre inizializzata a 0;
b) nel secondo caso nel momento in cui viene avviata l'esecuzione essa viene allocata in memoria e quindi ogni modifica ad essa sarà salvata in memoria

```c
// funzione contatore
void funzione1() {
  //variabile locale
  int contatore = 0;
  printf("Valore variabile: %d\n",
                  contatore);
  contatore++;
}

int main () {
  funzione1();
  funzione1();
  funzione1();
  funzione1();
  return 0;
}
```

Output:

```
Valore variabile: 0
Valore variabile: 0
Valore variabile: 0
Valore variabile: 0
```

```c
//funzione contatore (var. statica)
void funzione2() {
  //variabile statica
  static int contatore = 0;
  printf("Valore variabile: %d\n",
                  contatore);
  contatore++;
}

int main () {
  funzione2();
  funzione2();
  funzione2();
  funzione2();
  return 0;
}
```

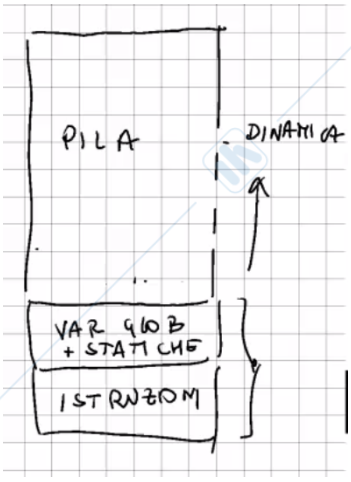
Output:

```
Valore variabile: 0
Valore variabile: 1
Valore variabile: 2
Valore variabile: 3
```

- **modello di esecuzione**: immaginiamo l'esistenza di una macchina dedicata all'esecuzione del programa main, e che siano create altre macchine dedicate ciascuna per la sua funzione; tutte queste sottoparti del calcolatore hanno un ambiente dedicato in memoria in cui vengono salvate variabili locali, valori dei parametri ricevuti, valori dei parametri da restituire

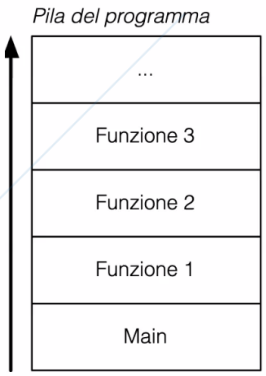
Quando viene chiamato un sottopragramma con una funzione il programma/sottoprogramma in corso viene sospeso ed il controllo passa al sottoprogramma invocato fino al termine di tale funzione.

a livello di macchina in C:
1- salvataggio del **program counter PC** e del **contesto** del programma →2- **assegnazione** al PC **dell'indirizzo** del sottoprogramma → 3- **esecuzione** del sottoprogramma; → 4- ritorno al programma chiamante e **ripristino del suo contesto**;



a livello della memoria: suddivisa in 4 parti:
1) **istruzioni** del programma: codice compilato,
2) **variabili statiche**: variabili sia globali che locali static già allocate in memoria;
3) **Pila** o **stack**: serve per salvare all'occorrenza le variabili dinamiche (record di attivazione) alla fine del programma è sempre vuota,
4)

- **Record di attivazione**: associato a ogni funzione, contiene tutti i dati locali del sottoprogramma (variabili e parametri), l'eventuale ritorno, l'indirizzo di ritorno nel programma chiamante (l'istruzione da cui ricominciare al termine del sottoprogramma), l'indirizzo del record di attivazione precedente, → ad ogni attivazione di un sottoprogramma si crea un nuovo record che viene eliminato al suo termine.
  Record di attivazione = ambiente di memoria

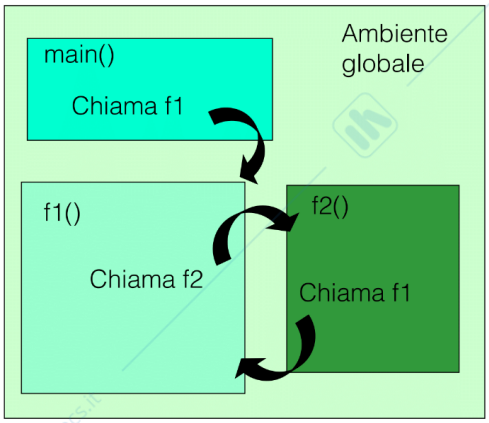


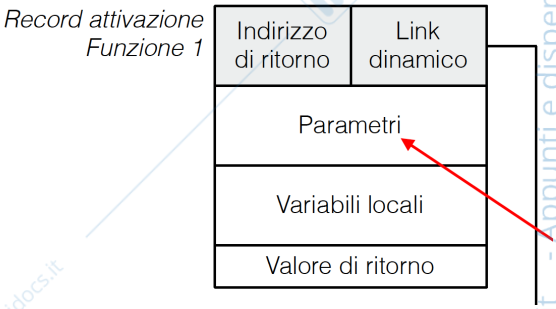
- la porzione di memoria dedicata alle variabili dinamiche è gestita come una **pila** (es. pila di piatti) : quando viene generato un record di attivazione esso viene allocato nella pila sopra il precedente → **push** quando un record di attivazione viene rimosso la porzione di memoria viene liberata rimuovendola dalla parte più alta → **pull**

  Tale sistema operativo di gestione delle chiamate ai sottoprogrammi è detto **LIFO = Last In First Out**, che ricalca l'ordine di invocazione delle funzioni

- **Stack pointer**: puntatore al registro della CPU che contiene l'indirizzo della parola di memoria occupata dal top dello stack.

- se una funzione richiama se stessa (**ricorsione**): possono esistere più istanze addormentate di una stessa funzione in attesa della terminazione di una gemella per riprendere → un compilatore non può sapere a priori quanto spazio di memoria allocare per tutte le variabili nei vari ambienti.
  (es. gestione di memoria di una pila con crescita ricorsiva di sue funzioni che si richiamano infinitamente)





(es. programma per moltiplicazione e potenza)

```c
#include <stdio.h>


int moltiplica (int x, int y) {
  int ris = x * y;
  return ris;
}


int potenza (int b, int e) {
  int i, p = 1;
  for(i = 1; i <= e; i++)
    p = moltiplica(p,b);
  return p;
}


int main() {

  int a = 2, b = 3, c;

  c = moltiplica(a,b);
  c = potenza(a,b);

  return 0;
}
```