2 - Programmazione modulare

Con programmazione modulare si intende la possibilità di dividere in **sotto-programmi** il programma principale per facilitarne la realizzazione del programma.
Questi piccoli frammenti di codice prendono il nome di **funzione (o procedura)**, definiti diversamente in base al linguaggio utilizzato.
Le funzioni sono un tipo di **astrazione**.
L'astrazione è il processo che porta a definire oggetti/concetti generali dalla conoscenza di oggetti particolari.
La programmazione modulare permette di implementare il paradigma **divide-et-impera**, che aiuta a gestire meglio la complessità dei problemi, suddividendoli in problemi più piccoli di dimensioni ridotte.
L'implementazione di questo paradigma si basa su due metodologie:

- **TOP DOWN**: suddividere il problema in sotto-problemi e implementare un sotto-programma per ciascuno di essi.
- **BOTTOM UP**: partire da frammenti di codice più piccoli e li unisco per ottenere programmi complessi.

Vantaggi della modularità

E' molto utile utilizzare la programmazione modulare poiché andremmo ad eliminare le difficoltà dell'uso di un **unico file sorgente**:

- Difficoltà nel tracciamento di errori logici
- Difficoltà di collaborazione nello stesso codice
- Difficoltà nel riutilizzo dello stesso codice in parti diversi di altri codici.
  I vantaggi che ci garantisce la programmazione modulare sono:
- Leggibilità
- Astrazione
- Riusabilità
- Modularità

Nel linguaggio *C* raggruppiamo le funzioni e le procedure attraverso le **librerie**, includendole attraverso la clausola `#include`. In precedenza ,senza creare direttamente delle funzioni, abbiamo già trovato in programmi 'semplice' dei richiami alla programmazione modulare, quando utilizziamo la direttiva #include stiamo in realtà già applicando i principi della programmazione modulare, perché stiamo **riutilizzando** funzioni non previste nel C standard che
sono state scritte da altri programmatori.

Principi della programmazione

Usufruendo della programmazione modulare usiamo due principi della programmazione modulare:

- Separazione delle competenze: ogni pezzo di codice svolge un ruolo ben preciso e delimitato.
- Information **hiding**: si nascondono i dettagli implementativi, di fatto quando implementiamo una funzione essa avviene in automatico.

Funzioni e procedure

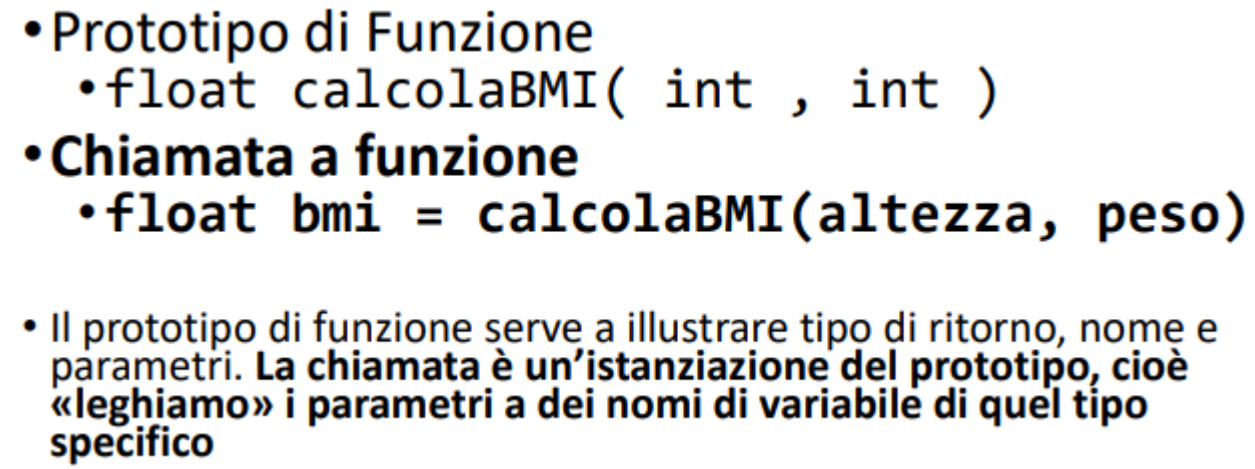
Le funzioni e le procedure sono un meccanismo di **astrazione**, in cui:

- Le funzioni sono un astrazione sui **dati**, perché permettono l'estendibilità degli operatori disponibili nel linguaggio
- Le procedure sono un astrazione sulle **istruzioni**, perché permettono l'estendibilità delle istruzioni primitive disponibili nel linguaggio.

Ogni funzione è divisa in due parti:

- **Prototipo della funzione**
- Implementazione
  Il prototipo della funzione è costituito da:
- nome
- insieme di parametri
- valore di ritorno

Da non confondere il prototipo con la **chiamata**



Passaggio di parametri per valore

Passare i parametri per valore sarebbe come attuare una "*copia*" del valore
della variabile in un'altra locazione di memoria. La funzione lavora sulla **nuova locazione di memoria**, la vecchia non viene mai **modificata**.
Terminata la funzione tutto ciò che era contenuto all'interno sarà **perso**.

```c
int main() {
   int x = 5;
   printf("Fuori: %d\n", x);
   quadrato(x);
   printf("Fuori: %d\n", x);
}

void quadrato (int y) {
    y = y*y
    printf("Dentro: %d", y);
}
```

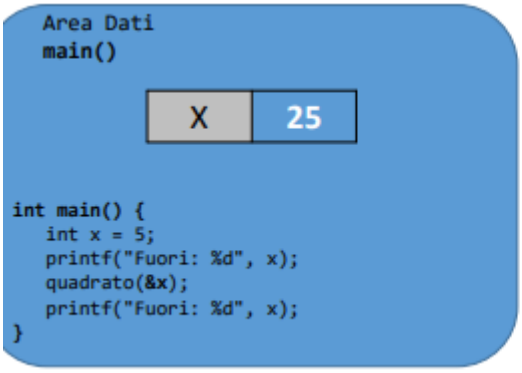
Cosa stampa questo programma?

```
Fuori: 5
Dentro: 25
Fuori: 5
```

Se non definiamo come attuare il passaggio sarà impostato di **default** per valore, tranne per gli **array**, i quali sono solo per **riferimento**.

Passaggio di parametri per riferimento

Sia il parametro formale che il parametro attuale puntano alla **stessa locazione di memoria**. Le modifiche quindi non vengono perse, anzi, vengono **ereditate** dal programma chiamante.



Tutte le modifiche vengono ereditate **dal main**

Cosa stampa questo programma?

```
Fuori: 5
Dentro: 25
Fuori: 25
```

Classi di memoria

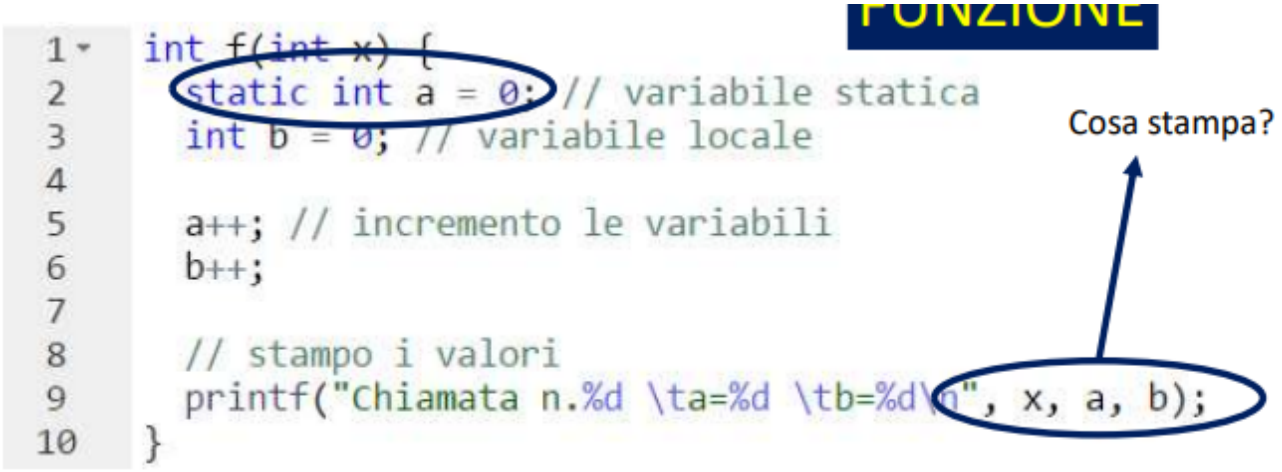
Una **classe di memoria** di un *identificatore* determina il tempo in cui quest'ultimo rimane in memoria, il quale si può classificare in permanenza di tipo **statica** e di tipo **automatica**.
La zona in cui una classe di memoria può essere **menzionata** nel programma viene detta **scope** (campo di visibilità). Il **collegamento** invece determina se l'identificatore è noto solo sul file corrente o in tutti i file del progetto.
Per comprendere se una variabile è permanente in memoria, nel *C*, si usano degli speciali specificatori:

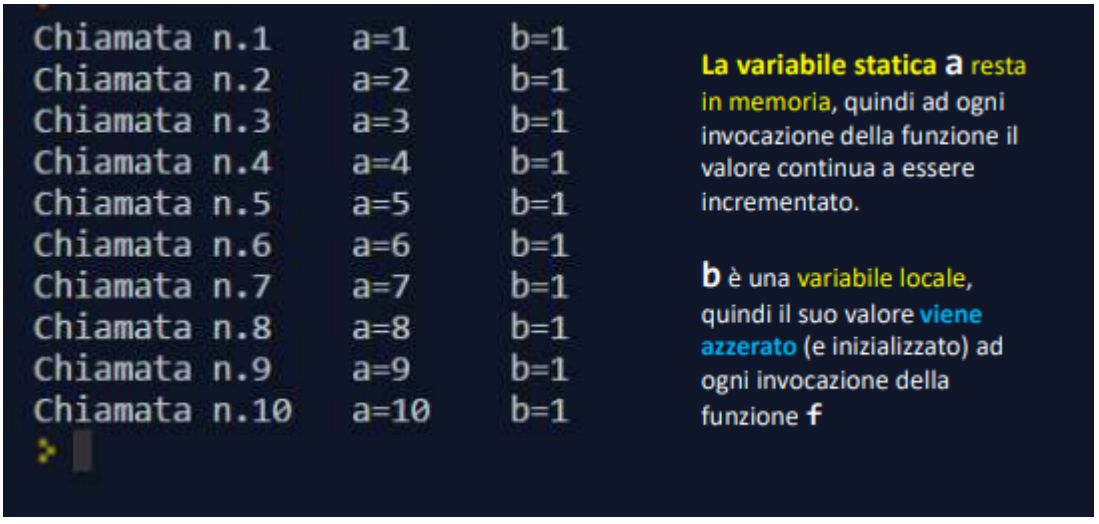
- **Auto**: è utilizzato per le variabili locali, infatti viene dettato di default e non serve specificarlo, la permanenza in memoria è pari al ciclo di vita del blocco.
- **Extern**: è utilizzato di default per le variabili globali, la permanenza in memoria è pari all'intera esecuzione del file.
- **Static**: la permanenza in memoria è pari all'esecuzione della funzione, ma il valore **non viene distrutto** al termine dell'esecuzione di essa.

- **Register**: questo specifica non viene utilizzata, è utile per le variabili a cui si accede spesso, come i contatori ad esempio.
  In generale si preferisce non usare le variabili globali, poiché possono creare effetti indesiderati al modificare del loro valore.

Esempi:

```c
int main(){
    //i variabile con specificatore register
    for(register int i=1;i<=10;i++){ //il register è obsoleto
        f(i);
    }
}
```





Scope

Come detto in precedenza lo *scope* di una variabile è il frammento di codice in cui una variabile è nota al compilatore e può essere utilizzata nel codice senza errori.
Lo scope segue una **regola** generale:
Una variabile è visibile nel **blocco in cui viene definita** e in tutti i **blocchi innestati**,a meno

di ridefinizioni.
Sulla base di questa si definiscono poi diverse tipologie di scope:

- **File scope**: tutti gli identificatori definiti fuori dalle funzioni sono visibili in tutto il file, si usa per i *prototipi di funzione* (devono essere richiamati in ogni momento) e le *variabili globali* (visibili e utilizzabili in tutto il codice sorgente)
- **Function scope**: gli identificatori definiti nel corpo di una funzione sono visibili solo in quella funzione (a meno di ridefinizioni in un blocco)
- **Block** scope: Tutti gli identificatori definiti in un *blocco, sono visibili solo in quel blocco*
- **Function Prototype Scope**: Gli identificatori definiti nel prototipo di una funzione valgono solo in esso. (possono anche essere omessi, non è detto che serva ri-specificarlo).

```c
#include <stdio.h>

// **File scope**: la variabile globale e il prototipo della funzione sono visibili in tutto il file
int globalVar = 10;  // Variabile globale con file scope

void funzione(int param);  // Prototipo della funzione con file scope

int main() {
    // **Function scope**: la variabile locale è visibile solo dentro main()
    int localVar = 20;

    printf("Global: %d, Local: %d\n", globalVar, localVar);

    // **Block scope**: la variabile dentro il blocco if è visibile solo nel blocco
    if (localVar > 10) {
        int blockVar = 30;  // Visibile solo dentro questo if
        printf("Block scope: %d\n", blockVar);
    }
    // printf("%d", blockVar); // Errore! blockVar non è visibile qui

    funzione(50); // Chiamata della funzione

    return 0;
}

// **Function Prototype Scope**: il parametro "param" è visibile solo all'interno della funzione
void funzione(int param) {
    printf("Function parameter: %d\n", param);
    // Il parametro "param" è visibile solo in questa funzione
}
```

Suggerimenti

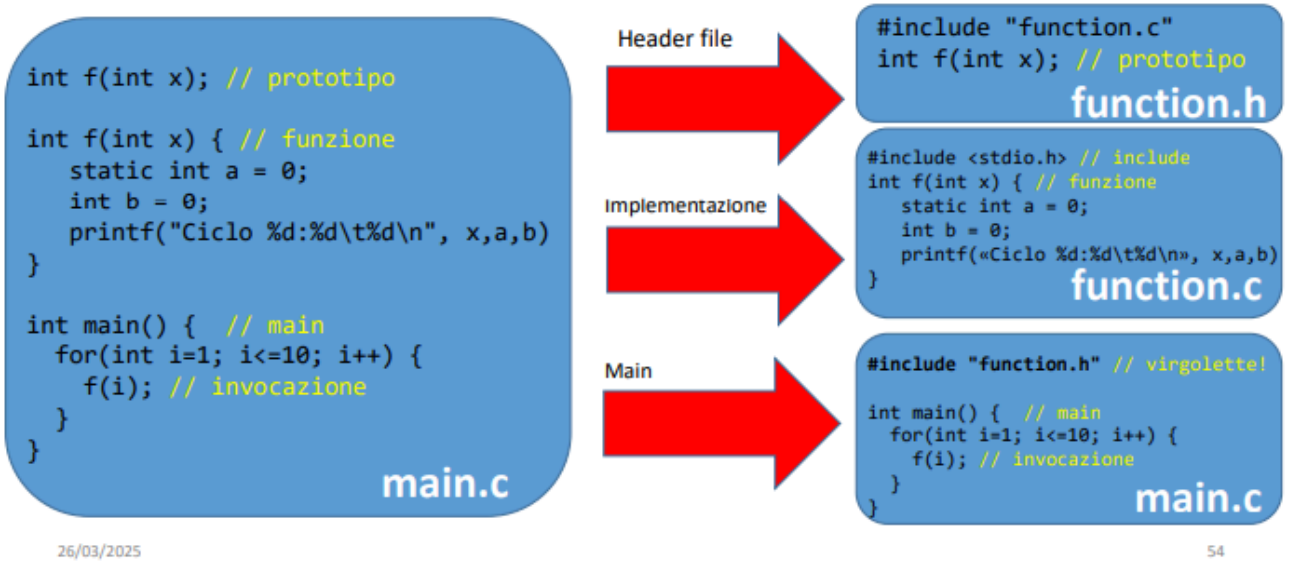
E' importante utilizzare il principio dell'*information Hiding*, poiché al codice dev'essere garantito solamente la possibilità di accedere al compito designato e non andare oltre. Inoltre inserire i dati in un livello più *interno* ci aiuta anche nella risoluzione degli errori.

Divisione delle funzioni

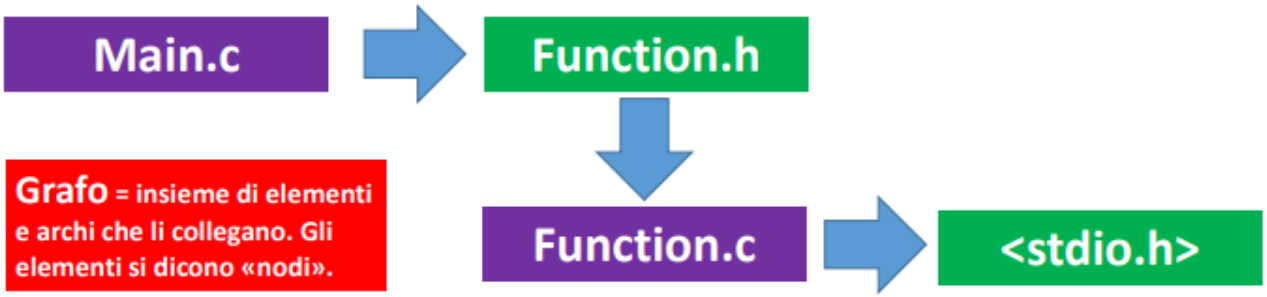
Usare tante funzioni presenti sempre nello stesso file non è comunque la soluzione ottimale e finale, per utilizzare a pieno i principi della programmazione modulare è necessario dividere **fisicamente** il codice sorgente, attraverso le **librerie** e l'**header files**.

Header Files

Gli header files sono dei file di *intestazione* la quale estensione è *.h* e sono tutti seguiti da un corrispettivo file *.c* che possiede l'omonimo nome, il loro scopo è contenere le intestazioni delle funzioni e delle procedure che si vogliono **separare** dal programma madre.
Per far ciò si creano dei nuovi file, all'interno dei quali vengono riportati le funzioni e in base al loro scopo vengono distribuite sui vari file creati. Questo è alla base di ciò che si è già utilizzato nelle **librerie standard**.
Ecco come cambierebbe un programma primitivo utilizzando l'header files:



Come possiamo notare nel file `function.c`, in cui è presente l'implementazione della funzione, si possono richiamare altre librerie e direttive se ciò fosse necessario. Un altra osservazione importante è che al richiamo dell'header files nel main non useremo le parentesi angolari come si usano per le librerie ma si devono implementare le **virgolette**.



L'utilizzo degli header files cambia la struttura dei programmi. Il **`main`** invoca le funzioni implementate in **`function.h`**, che a sua volta nella sua implementazione ha bisogno delle funzioni della **libreria standard**.

Il processo di esecuzione dei programmi **può essere rappresentato attraverso un grafo aciclico** (perché non ci possono essere dipendenze «circolari» tra librerie)

Ogni header file è suddiviso in **sette parti**:

1. **Prologo**:
   Esso è un commento che spiega a cosa serve il file in questione e che informazioni contiene, le informazioni principali che contiene sono: *Autori, Data, Numero di Versione, Riferimenti e Link esterni, Eventuali informazioni su Licenze e Copyright*.
2. **Guardia**:
   La guardia definisce delle condizioni tra le varie definizioni del file per evitare di fornire definizioni multiple.
3. **Inclusioni**:
   Sono tutte le librerie incluse in un modulo per poter far funzionare le loro funzioni.
4. **Costanti**:
   Un header file deve includere anche le definizioni di costanti e macro. E' possibile anche qui usare `#ifndef` (if not defined) per evitare ridefinizioni di costanti già definite nel programma.
5. **Tipi di Dato**:
   Serve a definire in un header file tutti i dati che sono `typedef`, per poter includere anche i nuovi tipi di dati creati da noi.
6. **Variabili Globali**:
   Le variabili definite negli header file hanno visibilità globale, sono visibili da tutte le funzioni o moduli. Per alcuni problemi può essere utile avere delle variabili globali accessibili da tutti e visibili a tutti.
7. **Prototipo di Funzione**:
   Per essere definito tale, ovviamente un header file deve contenere almeno un prototipo di almeno una funzione.

Header Files vs Librerie

E' importante capire a questo punto che header files e librerie sono strettamente collegati, ma non sono la stessa cosa.
L'header file contiene le dichiarazioni e prototipi da condividere con dei file sorgenti, non presenta del codice eseguibile a differenza di una libreria (la quale termina anche con

diverse estensioni) che possiede del codice già **compilato e ri-eseguibile** senza doverlo riscrivere su altri codici distinti.

Caratteristica	Header File (`.h`)	Libreria (`.a` , `.so` , `.lib` , `.dll`)
Contenuto	Dichiarazioni di funzioni, macro, costanti	Implementazione delle funzioni (già compilate)
Eseguibile?	❌ No, solo dichiarazioni	✅ Sì, contiene codice compilato
Scopo	Permette al compilatore di sapere cosa esiste	Contiene il codice reale delle funzioni
Come si usa?	`#include "file.h"` nel codice sorgente	Si compila con `-l` (es. `gcc main.c -lmialibreria`)

Compilazione codice sorgente

Ricapitoliamo in ordine come avviene la compilazione di un codice sorgente, in base a queste nuove nozioni:

- **Editor**
  E' la fase in cui il codice sorgente viene scritto o su un classico file di testo o su un IDE
- **Pre-processor**
  Zona in cui avviene l'elaborazione delle direttive del codice sorgente, dove si specificano le librerie da includere, le costanti e i macro. Il risultato di questa fase è un file sorgente **senza direttive del preprocessore**, pronto per essere compilato.
- **Compiler**
  In questa fase si verifica la correttezza a livello **sintattico** del codice, gli errori logici e matematici non sono individuati dal compilatore, successivamente avviene la costruzione del file **oggetto** (*file.o*) che sarà salvato su disco.
- **Linker**
  Fase in cui si **collegano** tra di loro i vari file oggetto costruiti e unire ad essi le librerie esterne, al fine di generare un unico e solo file **eseguibile**. Nel linker si uniscono il main e i file oggetto delle funzioni (file delle funzioni generati dall'header file) con le loro implementazioni. Il main conosce le varie funzioni che sono presenti al suo interno, grazie ai riferimenti nel file header, solo che sono dimostrati con simboli differenti, il ruolo del LINKER è quello di **linkare** i riferimenti simbolici con la reale implementazione della funzione.
- **Loader**
  Si carica in memoria e si lancia l'eseguibile compilato. Il processo è preso in carico dalla CPU che esegue sequenzialmente le istruzioni ed eventualmente alloca della memoria per creare variabili, file su disco, etc.