

Laboratorio di Informatica

Programmazione Modulare

(Parte 2)

docente: Cataldo Musto

cataldo.musto@uniba.it

Programmazione Modulare

- **Funzioni e Procedure sono un meccanismo di astrazione**
 - **Le funzioni sono un esempio di astrazione sui dati**, perchè ci permettono di estendere gli **operatori** disponibili nel linguaggio
 - **Le procedure sono un esempio di astrazione sulle istruzioni**, perchè ci permettono di estendere **le istruzioni primitive** disponibili nel linguaggio

Programmazione Modulare

- **Funzioni e Procedure** sono un **meccanismo di astrazione**
 - **Le funzioni sono un esempio di astrazione sui dati**, perchè ci permettono di estendere gli **operatori** disponibili nel linguaggio

```
65 // programma principale
66 int main() {
67     float BMI[5] = {0.0}; // dichiaro l'array dei BMI
68
69     acquireInput(BMI, DIMENSION); // metodo di acquisizione dell'input
70     float avg = calculateAverage(BMI, DIMENSION); // metodo per il calcolo della media
71     int max = calculateMaximum(BMI, DIMENSION); // metodo per il calcolo del massimo
72     printOutput(BMI, DIMENSION, avg, max); // Stampa dell'output
73 }
```

Programmazione Modulare

- **Funzioni e Procedure** sono un **meccanismo di astrazione**
 - **Le funzioni sono un esempio di astrazione sui dati**, perchè ci permettono di estendere gli **operatori** disponibili nel linguaggio

```
65 // programma principale
66 int main() {
67     float BMI[5] = {0.0}; // dichiaro l'array dei BMI
68
69     acquireInput(BMI, DIMENSION); // metodo di acquisizione
70     float avg = calculateAverage(BMI, DIMENSION); // calcolo la media
71     int max = calculateMaximum(BMI, DIMENSION); // calcolo il massimo
72     printOutput(BMI, DIMENSION, avg, max); // stampa i risultati
73 }
```

Le funzioni introducono **nuovi meccanismi per elaborare i dati**.

Prendono in input dei dati e ne producono altri.

Così come altre funzioni che abbiamo già usato, es: `sqrt(99.0)` oppure `isdigit('a')`;

Programmazione Modulare

- **Funzioni e Procedure** sono un **meccanismo di astrazione**
 - Le procedure sono un esempio di astrazione sulle istruzioni, perchè ci permettono di estendere le **istruzioni primitive** disponibili nel linguaggio

```
65 // programma principale
66 int main() {
67     float BMI[5] = {0.0}; // dichiaro l'array
68
69     acquireInput(BMI, DIMENSION); // metodo per acquisire i dati
70     float avg = calculateAverage(BMI, DIMENSION); // calcolo la
71     int max = calculateMaximum(BMI, DIMENSION); // calcolo il massimo
72     printOutput(BMI, DIMENSION, avg, max); // stampo i risultati
73 }
```

Le procedure non introducono nuovi dati, ma introducono **nuove istruzioni** non originariamente disponibili nel linguaggio.

Le procedure tipicamente si concentrano sulla gestione dell'input/output, così come `printf()` o `scanf()`

media
simo

Programmazione Modulare - Scope

Ogni variabile è descritta da due caratteristiche. **Lo scope** (la sua visibilità) **e la sua permanenza in memoria** (statica oppure automatica).

Programmazione Modulare - Scope

- **Lo scope (visibilità) di una variabile è il frammento di codice in cui una variabile è «visibile»**
 - **Visibile** → è nota al compilatore e può essere utilizzata nel codice senza produrre errori.

Programmazione Modulare - Scope

- **Lo scope (visibilità) di una variabile è il frammento di codice in cui una variabile è «visibile»**
 - **Visibile** → è nota al compilatore e può essere utilizzata nel codice senza produrre errori.
- **Regola Generale di Scope**
 - Una variabile è visibile **nel blocco in cui viene definita e in tutti i blocchi innestati**, a meno di ridefinizioni
 - Sulla base di questa regola si definiscono quattro **diverse tipologie di scope**

Programmazione Modulare - Scope

- **Tipologie di scope**
 - File Scope
 - Function Scope
 - Block Scope
 - Function Prototype Scope

Programmazione Modulare - Scope

- **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** (che devono poter essere richiamati in ogni momento) e **le variabili globali** (che sono utilizzabili e visibili in tutto il codice sorgente)

Programmazione Modulare - Scope

- **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** (che devono poter essere richiamati in ogni momento) e **le variabili globali** (che sono utilizzabili e visibili 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)
 - Si usa per le **variabili (locali).**

Programmazione Modulare - Scope

- **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** (che devono poter essere richiamati in ogni momento) e **le variabili globali** (che sono utilizzabili e visibili 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)
 - Si usa per le **variabili (locali).**

- **Block Scope**

- Tutti gli identificatori **definiti in un blocco**, sono visibili solo in quel blocco;

Programmazione Modulare - Scope

- **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** (che devono poter essere richiamati in ogni momento) e **le variabili globali** (che sono utilizzabili e visibili 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)
 - Si usa per le **variabili (locali).**

- **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.

Programmazione Modulare - Scope

```
void function(int var1);  
int var2;  
  
int main() {  
    int var3=0;  
  
    if(var3==0) {  
        int var4 = 1;  
    }  
}
```

Programmazione Modulare - Scope

```
void function(int var1); // var1 = ???  
int var2;                // var2 = ???  
  
int main() {  
    int var3=0;           // var3 = ???  
  
    if(var3==0) {  
        int var4 = 1;    // var4 = ???  
    }  
}
```

Programmazione Modulare - Scope

```
void function(int var1); // var1 = visibilità nel prototipo
int var2;                // var2 = visibilità globale

int main() {
    int var3=0;           // var3 = visibilità locale

    if(var3==0) {
        int var4 = 1;    // var4 = visibilità nel blocco
    }
}
```


Programmazione Modulare - Scope

```
void function(int var1); // var1 = visibilità nel prototipo
int var2;                // var2 = visibilità globale

int main() {
    int var3=0;           // var3 = visibilità locale
    var2=1;               // è una istruzione valida?
    if(var3==0) {
        int var4 = 1;    // var4 = visibilità nel blocco
        var3 = 2;
    }
    var1 = 3;
    var4 = 4;
}
```

Programmazione Modulare - Scope

```
void function(int var1); // var1 = visibilità nel prototipo
int var2;                // var2 = visibilità globale

int main() {
    int var3=0;           // var3 = visibilità locale
    var2=1;               // si, var2 è globale
    if(var3==0) {
        int var4 = 1;     // var4 = visibilità nel blocco
        var3 = 2;
    }
    var1 = 3;
    var4 = 4;
}
```

Programmazione Modulare - Scope

```
void function(int var1); // var1 = visibilità nel prototipo
int var2;                // var2 = visibilità globale

int main() {
    int var3=0;           // var3 = visibilità locale
    var2=1;               // sì, var2 è globale
    if(var3==0) {
        int var4 = 1;     // var4 = visibilità nel blocco
        var3 = 2;         // è una istruzione valida?
    }
    var1 = 3;
    var4 = 4;
}
```

Programmazione Modulare - Scope

```
void function(int var1); // var1 = visibilità nel prototipo
int var2;                // var2 = visibilità globale

int main() {
    int var3=0;           // var3 = visibilità locale
    var2=1;               // sì, var2 è globale
    if(var3==0) {
        int var4 = 1;    // var4 = visibilità nel blocco
        var3 = 2;        // sì, var3 è locale
    }
    var1 = 3;
    var4 = 4;
}
```

Programmazione Modulare - Scope

```
void function(int var1); // var1 = visibilità nel prototipo
int var2;                // var2 = visibilità globale

int main() {
    int var3=0;           // var3 = visibilità locale
    var2=1;               // sì, var2 è globale
    if(var3==0) {
        int var4 = 1;    // var4 = visibilità nel blocco
        var3 = 2;        // sì, var3 è locale
    }
    var1 = 3;
    var4 = 4;
}
```

Programmazione Modulare - Scope

```
void function(int var1); // var1 = visibilità nel prototipo
int var2;                // var2 = visibilità globale

int main() {
    int var3=0;           // var3 = visibilità locale
    var2=1;               // sì, var2 è globale
    if(var3==0) {
        int var4 = 1;     // var4 = visibilità nel blocco
        var3 = 2;        // sì, var3 è locale
    }
    var1 = 3; // errore, var1 è visibile solo nel prototipo
    var4 = 4; // errore, var4 è visibile solo nel blocco
}
```

Programmazione Modulare - Scope

```
void function(int var1); // var1 = visibilità nel prototipo
int var2;                // var2 = visibilità globale

int main() {
    int var3=0;           // var3 = visibilità locale
    var2=1;               // sì, var2 è globale
    if(var3==0) {
        int var4 = 1;    // var4 = visibilità nel blocco
        var3 = 2;        // sì, var3 è locale
    }
    var1 = 3; // errore, var1 è visibile solo nel prototipo
    int var4 = 4; // ?
}
```

Programmazione Modulare - Scope

```
void function(int var1); // var1 = visibilità nel prototipo
int var2;                // var2 = visibilità globale

int main() {
    int var3=0;           // var3 = visibilità locale
    var2=1;               // sì, var2 è globale
    if(var3==0) {
        int var4 = 1;     // var4 = visibilità nel blocco
        var3 = 2;        // sì, var3 è locale
    }
    var1 = 3; // errore, var1 è visibile solo nel prototipo
    int var4 = 4; // OK! Perché una volta terminato il blocco la
                  // variabile var4 può essere ri-definita.
}
```


Programmazione Modulare - Scope

- In Linguaggio C esistono degli specificatori per «definire» la permanenza in memoria di una variabile
 - Auto
 - Extern
 - Static
 - Register

Programmazione Modulare - Scope

- In Linguaggio C esistono degli specificatori per «definire» la permanenza in memoria di una variabile
 - **Auto**
 - Valore di default per le variabili locali (non serve indicarlo). **Imposta la permanenza in memoria pari al ciclo di vita del blocco.**

Programmazione Modulare - Scope

- **In Linguaggio C esistono degli specificatori per «definire» la permanenza in memoria di una variabile**
 - **Auto**
 - Valore di default per le variabili locali (non serve indicarlo). **Imposta la permanenza in memoria pari al ciclo di vita del blocco.**
 - **Extern**
 - Valore di default per le variabili globali. **La permanenza in memoria è pari all'intera esecuzione del file.**

Programmazione Modulare - Scope

- **In Linguaggio C esistono degli specificatori per «definire» la permanenza in memoria di una variabile**
 - **Auto**
 - Valore di default per le variabili locali (non serve indicarlo). **Imposta la permanenza in memoria pari al ciclo di vita del blocco.**
 - **Extern**
 - Valore 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 di una funzione, **ma il suo valore non viene distrutto quando termina l'esecuzione della funzione**

Programmazione Modulare - Scope

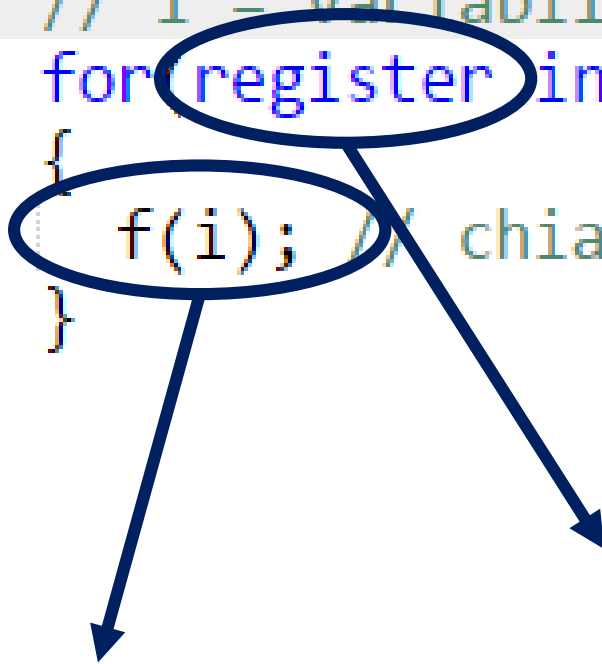
- **In Linguaggio C esistono degli specificatori per «definire» la permanenza in memoria di una variabile**
 - **Auto**
 - Valore di default per le variabili locali (non serve indicarlo). **Imposta la permanenza in memoria pari al ciclo di vita del blocco.**
 - **Extern**
 - Valore di default per le variabili globali. **La permanenza in memoria è pari all'intera esecuzione del file.**
 - **Static**
 - La permanenza in memoria è pari all'esecuzione di una funzione, **ma il suo valore non viene distrutto quando termina l'esecuzione della funzione**
 - **Register**
 - Memorizza la variabile in registri ad alta velocità, per velocizzarne l'accesso. **Utile per variabili cui si accede spesso, es. i contatori – NB) OBSOLETO, non utilizzato.**

Specificatori - Esempio

```
12 ▾ int main() {  
13     // i = variabile register|  
14     for(register int i=1; i<=10; i++) // ciclo  
15 ▾ {  
16         f(i); // chiamo una funzione  
17     }  
18 }
```

Specificatori - Esempio

```
12 ▾ int main() {  
13     // i = variabile register  
14     for(register int i=1; i<=10; i++) // ciclo  
15 ▾ {  
16         f(i); // chiamo una funzione  
17     }  
18 }
```

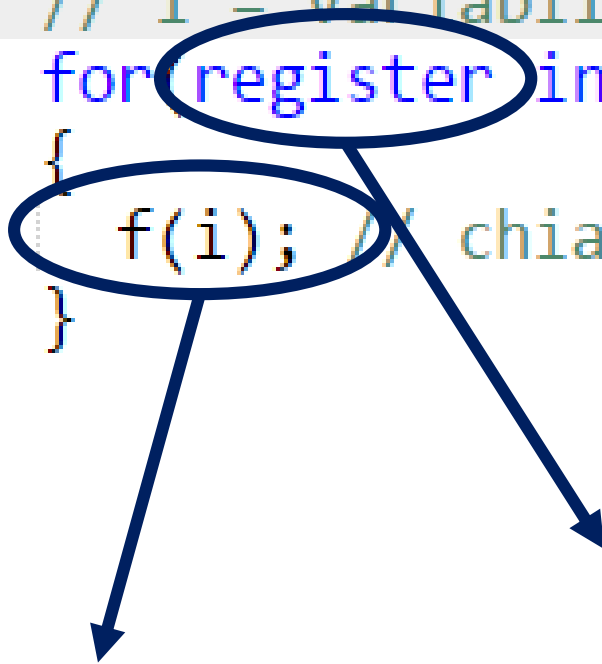


Come viene passato il parametro?

Specificatore utile **per le variabili cui si accede spesso**. Velocizza l'accesso inserendole nei registri. Obsoleto!

Specificatori - Esempio

```
12 ▾ int main() {  
13     // i = variabile register  
14     for(register int i=1; i<=10; i++) // ciclo  
15 ▾ {  
16         f(i); // chiamo una funzione  
17     }  
18 }
```



Parametro passato
per valore.

Specificatore utile **per le variabili cui si accede spesso**. Velocizza l'accesso inserendole nei registri. Obsoleto!

Specificatori - Esempio

```
1 int f(int x) {  
2     static int a = 0; // variabile statica  
3     int b = 0; // variabile locale  
4  
5     a++; // incremento le variabili  
6     b++;  
7  
8     // stampo i valori  
9     printf("Chiamata n.%d \ta=%d \tb=%d\n", x, a, b);  
10 }
```

Specificatori - Esempio

```
1 int f(int x) {  
2     static int a = 0; // variabile statica  
3     int b = 0; // variabile locale  
4  
5     a++; // incremento le variabili  
6     b++;  
7  
8     // stampo i valori  
9     printf("Chiamata n.%d \ta=%d \tb=%d\n", x, a, b);  
10 }
```

Cosa stampa?

Specificatori – Esempio (Output)

```
Chiamata n.1    a=1    b=1
```

Specificatori – Esempio (Output)

```
Chiamata n.1    a=1    b=1  
Chiamata n.2    a=2    b=1
```

Specificatori – Esempio (Output)

```
Chiamata n.1    a=1    b=1
Chiamata n.2    a=2    b=1
Chiamata n.3    a=3    b=1
Chiamata n.4    a=4    b=1
Chiamata n.5    a=5    b=1
Chiamata n.6    a=6    b=1
Chiamata n.7    a=7    b=1
Chiamata n.8    a=8    b=1
Chiamata n.9    a=9    b=1
Chiamata n.10   a=10   b=1
```



La variabile statica `a` resta in memoria, quindi ad ogni invocazione della funzione il valore continua a essere incrementato.

`b` è un parametro **passato per valore**, quindi il suo valore **viene azzerato** (e inizializzato) ad ogni invocazione della funzione **`f`**

Progettazione Modulare

Progettazione Modulare

- L'utilizzo delle funzioni e delle procedure **risolve parzialmente il** problema della programmazione modulare
- **Perché?**

Progettazione Modulare

- L'utilizzo delle funzioni e delle procedure **risolve parzialmente** il problema della programmazione modulare
- **Perché?**
 - Il codice sorgente è comunque tutto aggregato in un unico file (anche se «spacchettato» in diverse funzioni.
 - Per implementare totalmente i principi della programmazione modulare è necessario anche dividere «**fisicamente**» il codice sorgente
 - **Come?**

Progettazione Modulare

- L'utilizzo delle funzioni e delle procedure **risolve parzialmente** il problema della programmazione modulare
- **Perché?**
 - Il codice sorgente è comunque tutto aggregato in un unico file (anche se «spacchettato» in diverse funzioni.
 - Per implementare totalmente i principi della programmazione modulare è necessario anche dividere «**fisicamente**» il codice sorgente
 - **Come?**
 - **Header Files**
 - **Librerie Statiche**

Progettazione Modulare

	Header Files	Librerie Statiche
Repl.it	SI	NO
Eclipse	SI	SI

Progettazione Modulare

	Header Files	Librerie Statiche
Repl.it	SI	NO
Eclipse	SI	SI

Eclipse da la possibilità di implementare la programmazione modulare sia attraverso l'utilizzo delle **librerie statiche** che attraverso la definizione degli **header files**.

Repl.it permette solo di dividere il codice sorgente in più file più piccoli, utilizzando gli **header files**

Entrambe le modalità sono accettate.

Progettazione Modulare – Header Files

- Gli **header files** sono ‘files di intestazione’ (tradotto letteralmente)
 - Hanno estensione `.h` (esempio: `functions.h`)
 - Sono accompagnati da un file `.c` con lo stesso nome (`functions.c`)
 - **Contengono le intestazioni delle funzioni e delle procedure che vogliamo separare dal programma principale**

Progettazione Modulare – Header Files

- Gli **header files** sono ‘files di intestazione’ (tradotto letteralmente)
 - Hanno estensione `.h` (esempio: `functions.h`)
 - Sono accompagnati da un file `.c` con lo stesso nome (`functions.c`)
 - **Contengono le intestazioni delle funzioni e delle procedure che vogliamo separare dal programma principale**
- Si creano uno o più nuovi files e si inseriscono le funzioni in questi files
 - **Le funzioni vengono aggregate in diversi header files**, in base al loro scopo (es. tutte le funzioni che si occupano di input/output, tutte le funzioni per operazioni matematiche, etc.)
 - Un po’ come avviene nelle funzioni della libreria standard del C (es. `<string.h>` `<ctype.h>` etc.)
- **Come cambia la struttura dei programmi?**

Progettazione Modulare – Header Files

```
void f(int x); // prototipo

int f(int x) { // funzione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}

int main() { // main
    for(int i=1; i<=10; i++) {
        f(i); // invocazione
    }
}
```

main.c

Progettazione Modulare – Header Files

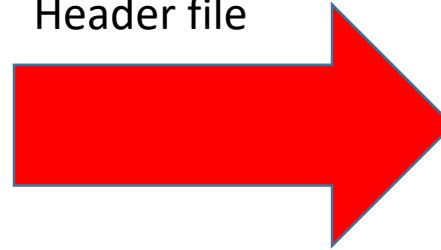
```
void f(int x); // prototipo

int f(int x) { // funzione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}

int main() { // main
    for(int i=1; i<=10; i++) {
        f(i); // invocazione
    }
}
```

main.c

Header file



```
void f(int x); // prototipo
function.h
```

Progettazione Modulare – Header Files

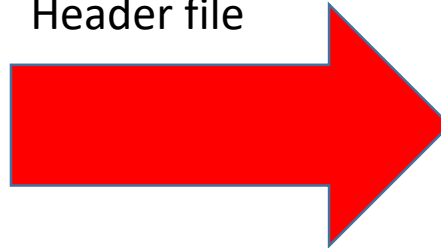
```
void f(int x); // prototipo

int f(int x) { // funzione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}

int main() { // main
    for(int i=1; i<=10; i++) {
        f(i); // invocazione
    }
}
```

main.c

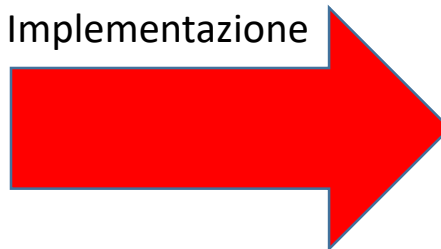
Header file



```
void f(int x); // prototipo
```

function.h

Implementazione



```
#include <stdio.h> // include
int f(int x) { // funzione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}
```

function.c

Progettazione Modulare – Header Files

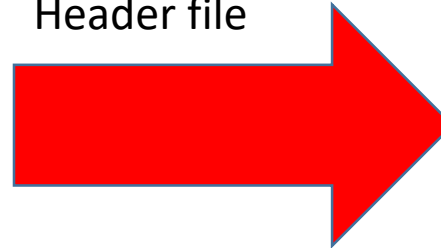
```
void f(int x); // prototipo

int f(int x) { // funzione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}

int main() { // main
    for(int i=1; i<=10; i++) {
        f(i); // invocazione
    }
}
```

main.c

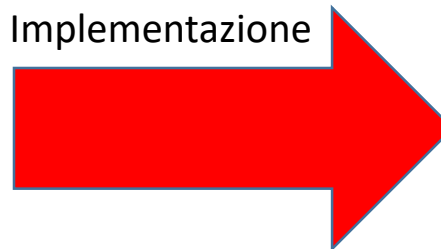
Header file



```
void f(int x); // prototipo
```

function.h

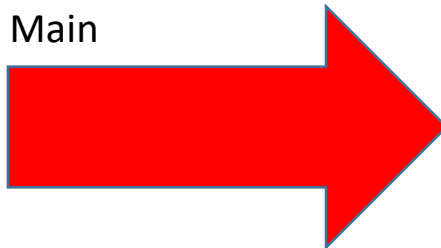
Implementazione



```
#include <stdio.h> // include
int f(int x) { // funzione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}
```

function.c

Main



```
#include 'function'.h // virgolette!

int main() { // main
    for(int i=1; i<=10; i++) {
        f(i); // invocazione
    }
}
```

main.c

Progettazione Modulare – Header Files

```
void f(int x); // prototipo
```

function.h

```
#include <stdio.h> // include
int f(int x) { // funzione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}
```

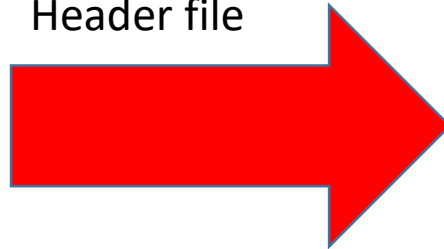
function.c

```
#include 'function'.h // virgolette!

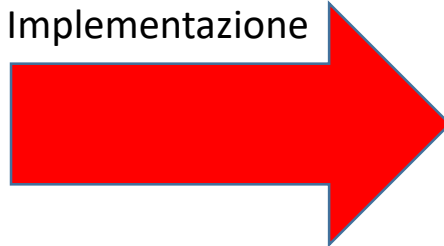
int main() { // main
    for(int i=1; i<=10; i++) {
        f(i); // invocazione
    }
}
```

main.c

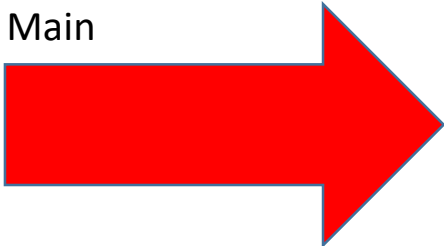
Header file



Implementazione



Main



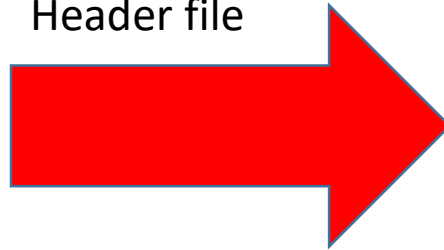
Progettazione Modulare – Header Files

```
void f(int x); // prototipo  
function.h
```

```
#include <stdio.h> // include  
int f(int x) { // funzione  
    static int a = 0;  
    int b = 0;  
    printf(«Ciclo %d:%d\t%d\n», x,a,b)  
}  
function.c
```

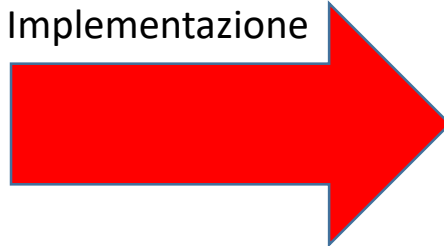
```
#include 'function'.h // virgolette!  
  
int main() { // main  
    for(int i=1; i<=10; i++) {  
        f(i); // invocazione  
    }  
}  
main.c
```

Header file

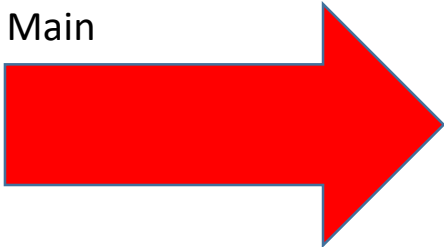


Nell'header file **andiamo a inserire solo i prototipi di funzione**

Implementazione



Main



Progettazione Modulare – Header Files

```
void f(int x); // prototipo  
function.h
```

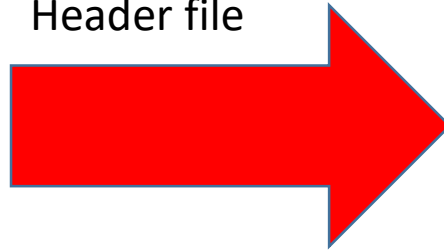
```
#include <stdio.h> // include  
int f(int x) { // funzione  
    static int a = 0;  
    int b = 0;  
    printf(«Ciclo %d:%d\t%d\n», x,a,b)  
}
```

function.c

```
#include 'function'.h // virgolette!  
  
int main() { // main  
    for(int i=1; i<=10; i++) {  
        f(i); // invocazione  
    }  
}
```

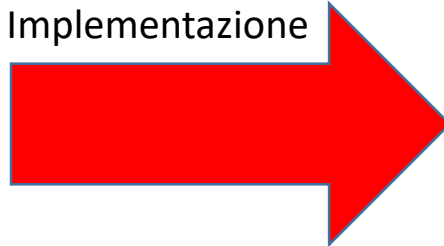
main.c

Header file



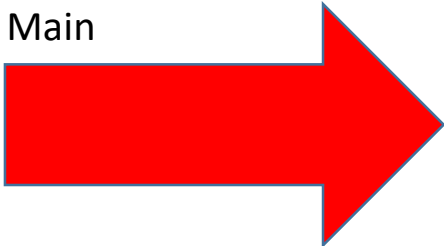
Nell'header file **andiamo a inserire solo i prototipi di funzione**

Implementazione



Nel file di implementazione **(con lo stesso nome!)** inseriamo **l'implementazione dei prototipi**. Se necessario, i file di implementazione possono avere a loro volta delle direttive **#include**

Main



Progettazione Modulare – Header Files

```
void f(int x); // prototipo
```

function.h

```
#include <stdio.h> // include
int f(int x) { // funzione
    static int a = 0;
    int b = 0;
    printf(«Ciclo %d:%d\t%d\n», x,a,b)
}
```

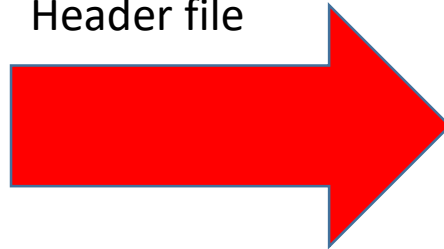
function.c

```
#include 'function'.h // virgolette

int main() { // main
    for(int i=1; i<=10; i++) {
        f(i); // invocazione
    }
}
```

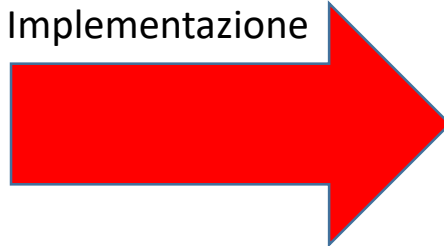
main.c

Header file



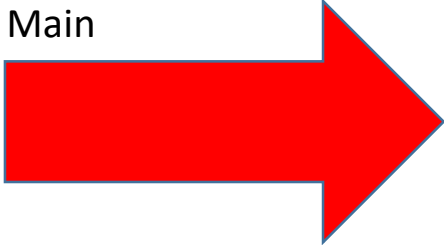
Nell'header file **andiamo a inserire solo i prototipi di funzione**

Implementazione



Nel file di implementazione **(con lo stesso nome!)** inseriamo **l'implementazione dei prototipi**. Se necessario, i file di implementazione possono avere a loro volta delle direttive **#include**

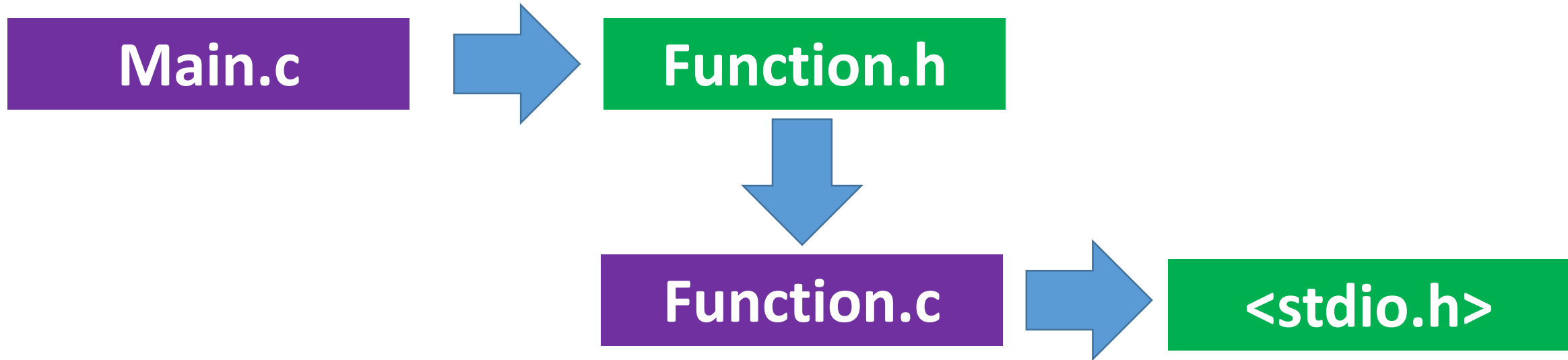
Main



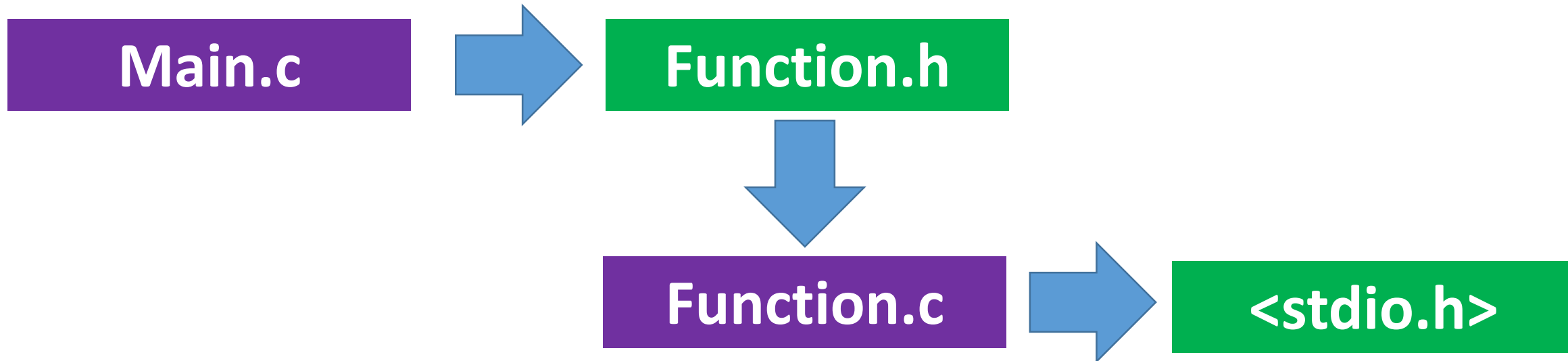
Nel main includiamo il nostro nuovo header files, così come se fosse una delle librerie standard del C, e possiamo utilizzarne le funzioni.

Importante: doppie virgolette, non parentesi angolari!

Progettazione Modulare – Header Files



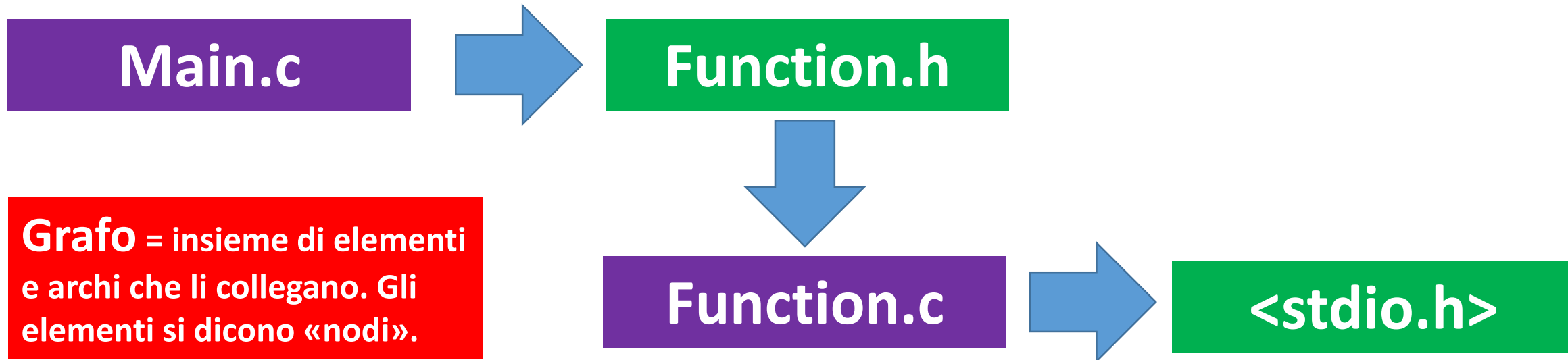
Progettazione Modulare – Header Files



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**.

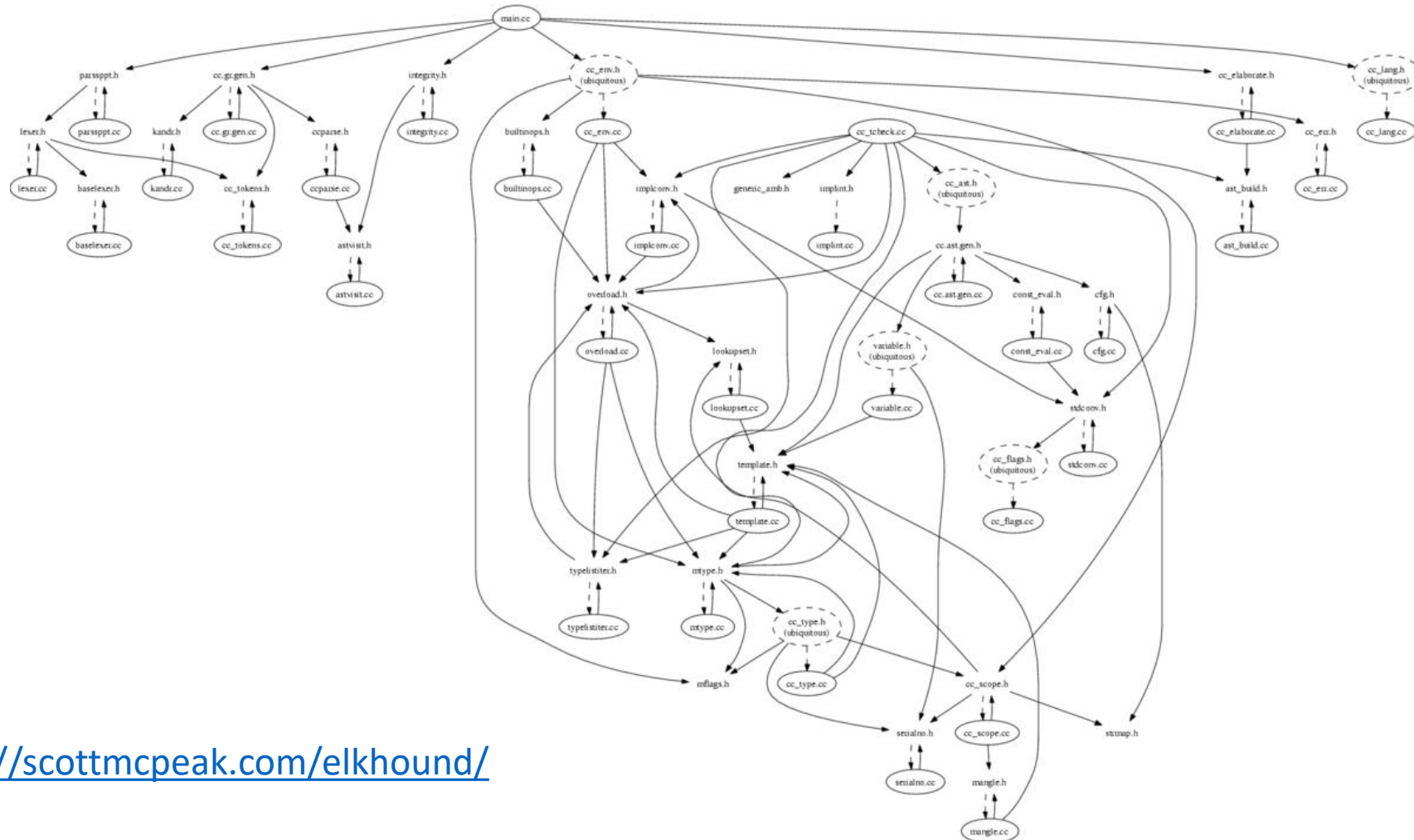
Il processo di esecuzione dei programmi **può essere rappresentato attraverso un grafo aciclico** (perché non ci possono essere dipendenze «circolari» tra librerie)

Progettazione Modulare – Header Files



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)



57

Progettazione Modulare – Header Files

- **Ogni header file è diviso in sette parti**

1. Prologo
2. Guardia
3. Direttive di inclusione
4. Costanti
5. Tipi di Dato
6. Variabili Globali
7. Prototipi di Funzione

Progettazione Modulare – Header Files

- **Ogni header file è diviso in sette parti**

1. Prologo

- Si tratta di un «commento», descrive a cosa serve il file e che tipo di informazioni contiene
- Autori, Data, Numero di Versione, Riferimenti e Link esterni, Eventuali informazioni su Licenze e Copyright

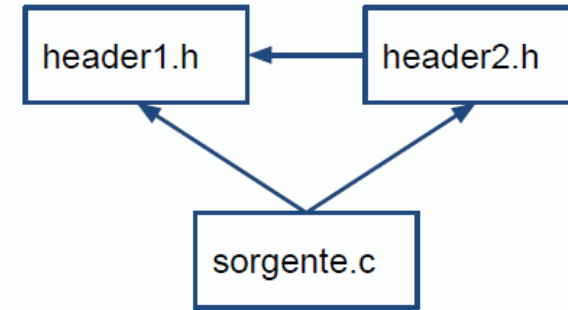
```
/*  
 * CUnit - A Unit testing  
 * framework library for C.  
 * Copyright (C) 2004-2006 Jerry  
 * St.Clair  
 *  
 * This library is free software; ...  
 * License as published by the  
 * Free Software Foundation;  
 ...  
 * 11-Aug-2004 Initial  
 * implementation of basic test  
 * runner interface. (JDS)  
 */
```

Progettazione Modulare – Header Files

- **Ogni header file è diviso in sette parti**

2. Guardia

- La struttura a «grafo» delle dipendenze può portare ad avere dipendenze multiple

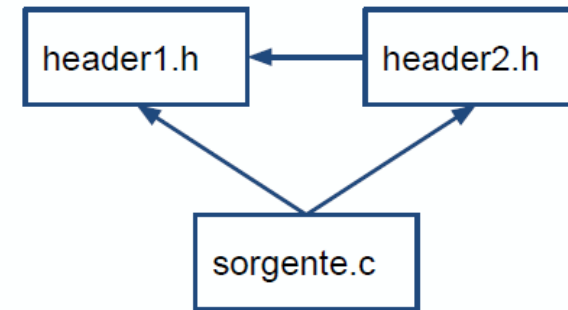


Progettazione Modulare – Header Files

- **Ogni header file è diviso in sette parti**

2. Guardia

- La struttura a «grafo» delle dipendenze può portare ad avere dipendenze multiple
- **La guardia serve a inserire delle condizioni** nella definizione delle informazioni che sono contenute nell'header file, **per evitare di fornire definizioni multiple.**



```
#ifndef CUNIT_BASIC_H_SEEN
#define CUNIT_BASIC_H_SEEN
... codice
#endif
```

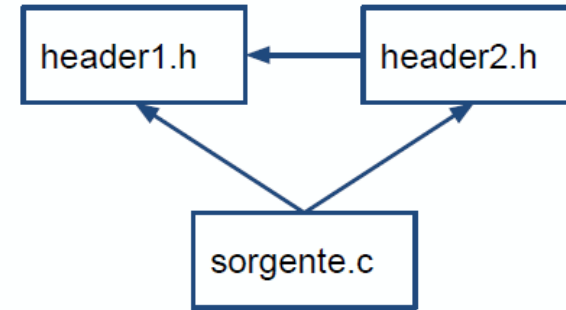
Thanks to Corrado Mencar

Progettazione Modulare – Header Files

- **Ogni header file è diviso in sette parti**

2. Guardia

- La struttura a «grafo» delle dipendenze può portare ad avere dipendenze multiple
- **La guardia serve a inserire delle condizioni** nella definizione delle informazioni che sono contenute nell'header file, **per evitare di fornire definizioni multiple.**



```
#ifndef CUNIT_BASIC_H_SEEN  
#define CUNIT_BASIC_H_SEEN  
... codice  
#endif
```

#ifndef → if not defined

*(Se il file non è già stato incluso,
continua a leggere)*

Progettazione Modulare – Header Files

- **Ogni header file è diviso in sette parti**

- 3. Inclusioni

- Un modulo può implementare delle funzioni **che a loro volta abbiano bisogno di altre librerie**
 - **Nel nostro esempio includevamo `<stdio.h>` per poter utilizzare l'istruzione `printf()`.**

```
/* System includes */  
#include <stdio.h>  
#include <stdlib.h>
```

```
/* Local includes */  
// Unit test framework  
#include "CUnit/Basic.h"  
// libreria di manipolazione  
matrici  
#include "math/matrix.h"
```

Progettazione Modulare – Header Files

- **Ogni header file è diviso in sette parti**

- 4. Definizioni

- **Un header file** deve includere anche le definizioni di costanti e macro.
 - **E' possibile anche qui usare `#ifndef` per evitare ridefinizioni di costanti già definite nel programma**

```
#define CU_VERSION "2.1-2"

#define
CU_MAX_TEST_NAME_LENGTH 256

#ifndef CU_TRUE
    #define CU_TRUE 1
#endif

#   define CU_MAX(a,b)
    (((a) >= (b)) ? (a) : (b))
```


Progettazione Modulare – Header Files

- **Ogni header file è diviso in sette parti**

- 5. Tipi di dato

- **Un header file** deve includere anche le definizioni di tipi di dato
 - **Come abbiamo visto, e' utile definire nuovi tipi di dato con il comando `typedef`. Tutti i nuovi tipi di dato sono da includere nell'header file.**
 - **Utilità**
 - Astrazione
 - Leggibilità

```
typedef enum {  
    CU_BRM_NORMAL = 0,  
    CU_BRM_SILENT,  
    CU_BRM_VERBOSE  
} CU_BasicRunMode;
```

Progettazione Modulare – Header Files

- **Ogni header file è diviso in sette parti**

- 6. Variabili

- **Le variabili definite negli header file hanno visibilità globale**, sono visibili da tutte le funzioni o da tutti i moduli.
 - Per alcuni problemi può essere utile avere delle variabili globali accessibili da tutti e visibili a tutti.

```
int global_variable;  
// visibile a livello  
// globale. EVITARE  
  
static int mod_var;  
// visibile solo  
// a livello di  
// modulo
```

Progettazione Modulare – Header Files

- **Ogni header file è diviso in sette parti**

- 7. Prototipi di funzione

- **A livello minimale, un header file deve contenere almeno un prototipo di funzione**

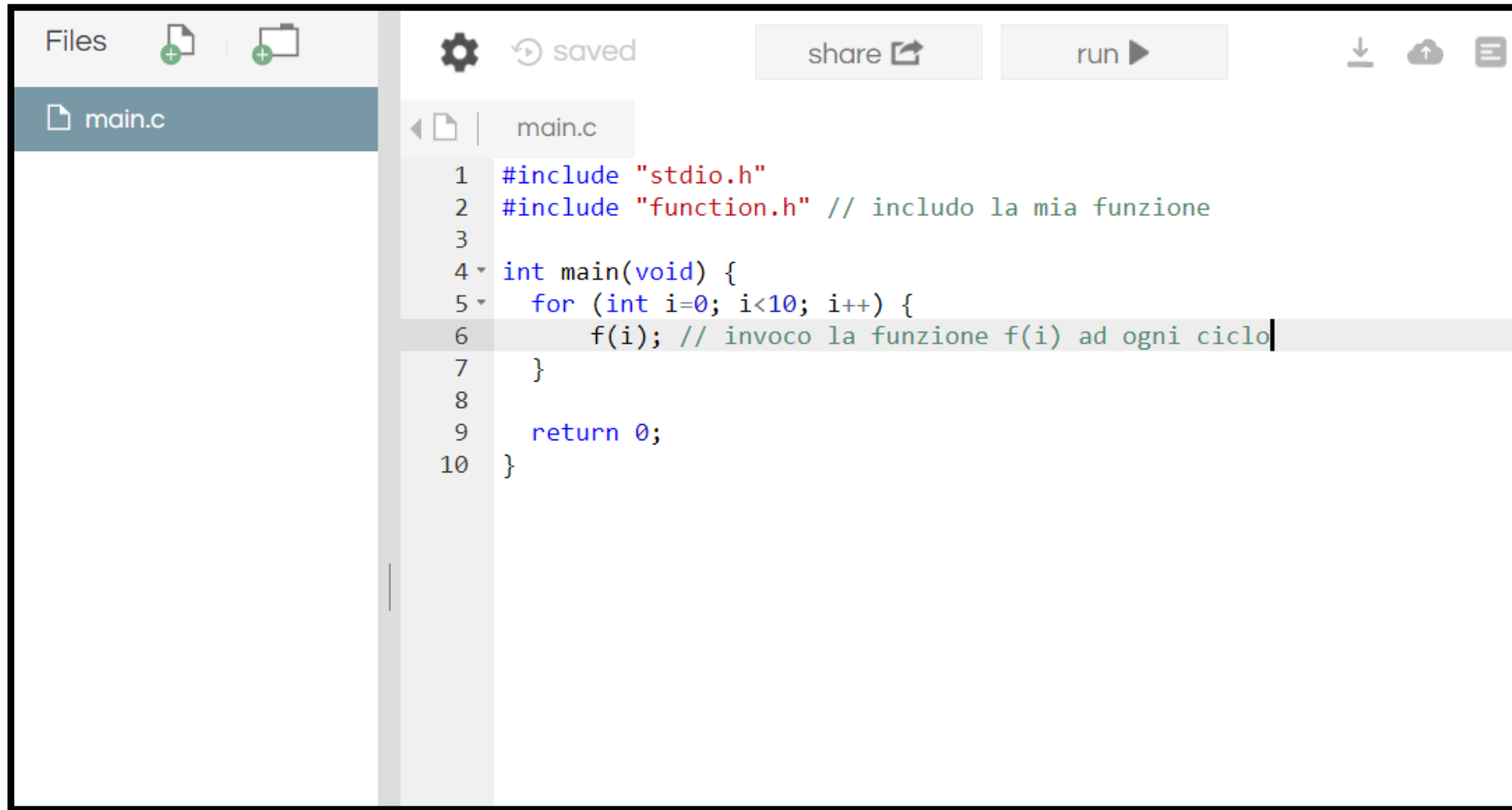
```
CU_ErrorCode CU_basic_run_suite(CU_pSuite pSuite);  
/**<  
 *   Runs all tests for a specific suite in the  
 *   basic interface...
```



Programmazione Modulare

su Repl ed Eclipse

Progettazione Modulare – Header Files (in Repl)



The screenshot shows a web-based IDE interface. On the left, a 'Files' sidebar lists 'main.c'. The main editor area displays the following C code:

```
1 #include "stdio.h"
2 #include "function.h" // includo la mia funzione
3
4 int main(void) {
5     for (int i=0; i<10; i++) {
6         f(i); // invoco la funzione f(i) ad ogni ciclo
7     }
8
9     return 0;
10 }
```

The interface includes a top bar with icons for settings, saved status, share, run, download, upload, and a menu. A line number margin is visible on the left side of the code editor.

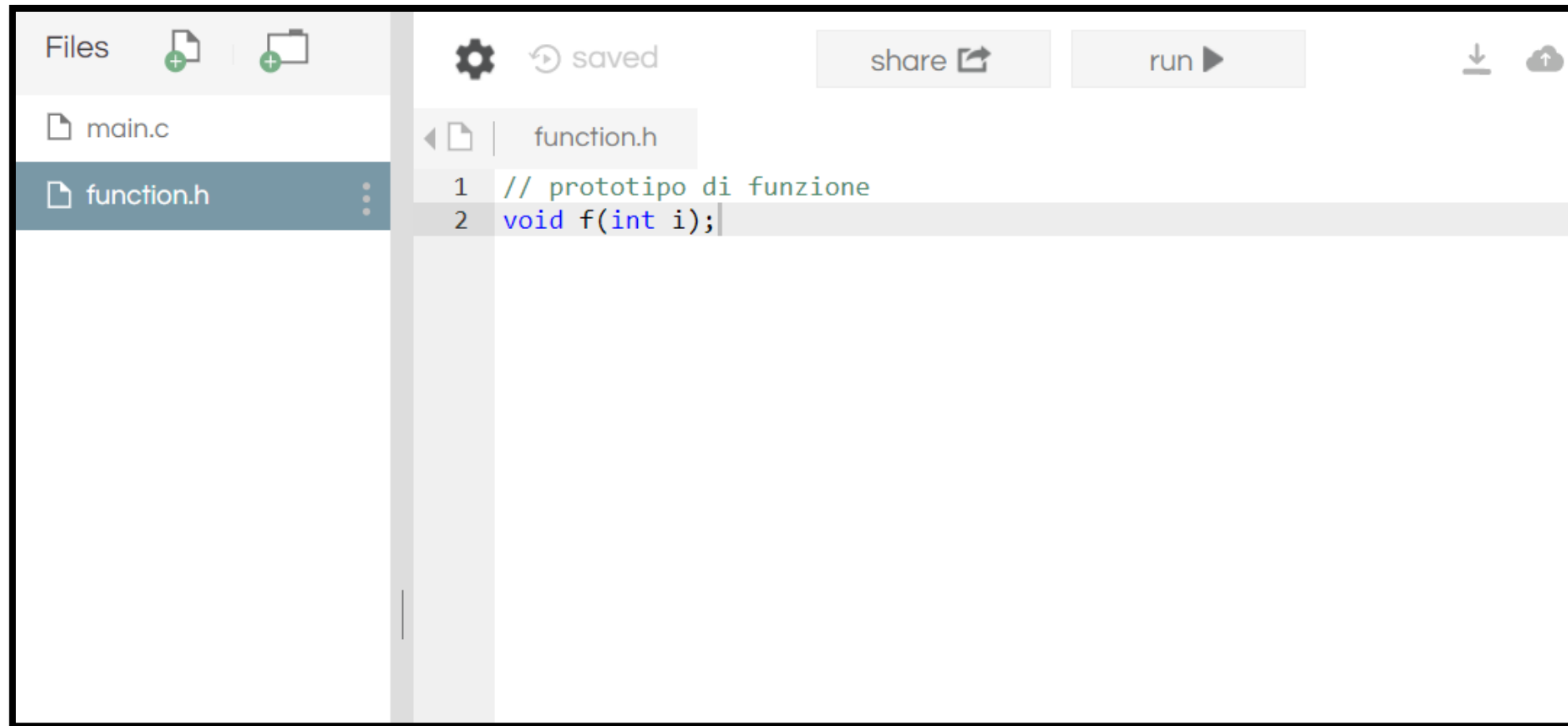
Progettazione Modulare – Header Files (in Repl)



Progettazione Modulare – Header Files (in Repl)



Progettazione Modulare – Header Files (in Repl)



The screenshot shows a web-based IDE interface. On the left, a file explorer lists 'main.c' and 'function.h', with 'function.h' selected. The main editor area shows the content of 'function.h' with two lines of C code: a comment and a function prototype. The interface includes a top bar with icons for settings, saving, sharing, and running, as well as download and upload buttons.

```
1 // prototipo di funzione
2 void f(int i);
```

Progettazione Modulare – Header Files (in Repl)



The screenshot shows a Repl IDE interface. On the left, a file explorer lists three files: `main.c`, `function.c` (selected), and `function.h`. The main editor area displays the contents of `function.c`. The code is as follows:

```
1 // implementazione della funzione
2 #include <stdio.h>
3
4 void f(int i) {
5     static int a = 10; // variabile statica
6
7     printf("%d\t%d\n", a++, i); // stampa il valore di a e di i
8 }
9
```

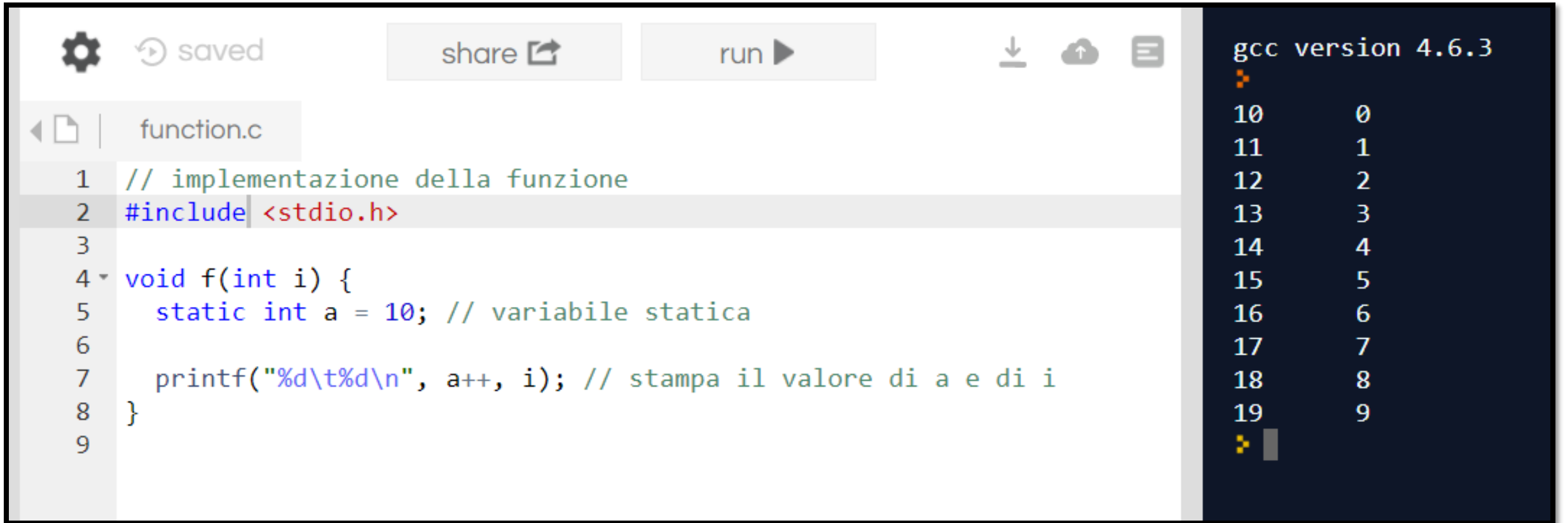
Progettazione Modulare – Header Files (in Repl)

The screenshot shows a Repl IDE interface. On the left, a file explorer lists 'main.c', 'function.c' (selected), and 'function.h'. The main editor displays the contents of 'function.c' with the following code:

```
1 // implementazione della funzione
2 #include <stdio.h>
3
4 void f(int i) {
5     static int a = 10; // variabile statica
6
7     printf("%d\t%d\n", a++, i); // stampa il valore di a e di i
8 }
9
```

Below the code editor, a red rectangular box contains the text: **Cosa stampa in output?**

Progettazione Modulare – Header Files (in Repl)



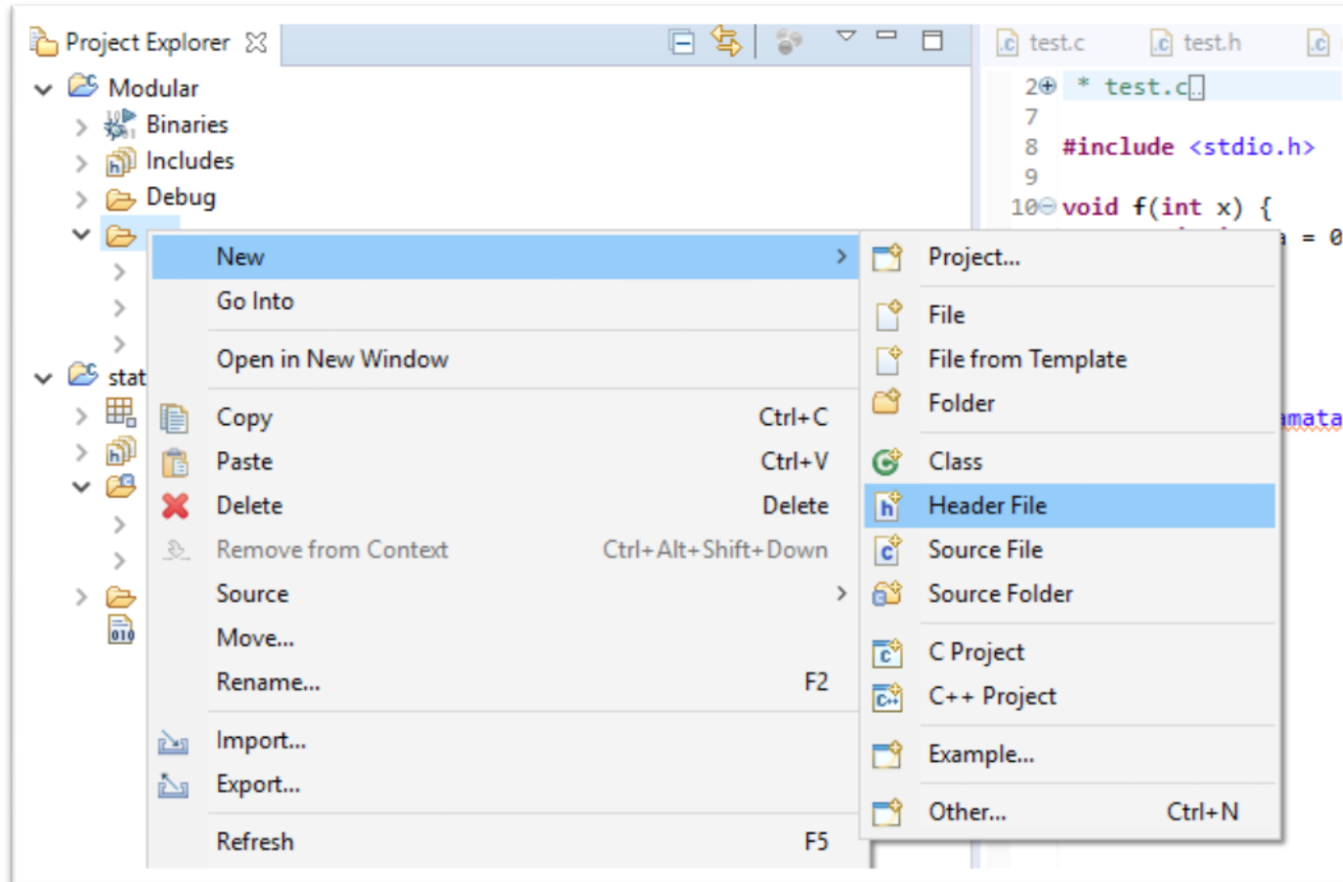
The screenshot shows a Repl environment with a code editor on the left and a terminal on the right. The code editor displays a C program named `function.c` with the following content:

```
1 // implementazione della funzione
2 #include <stdio.h>
3
4 void f(int i) {
5     static int a = 10; // variabile statica
6
7     printf("%d\t%d\n", a++, i); // stampa il valore di a e di i
8 }
9
```

The terminal on the right shows the output of the program, which is a table of values for `a` and `i`:

```
gcc version 4.6.3
10      0
11      1
12      2
13      3
14      4
15      5
16      6
17      7
18      8
19      9
```

Progettazione Modulare – Header Files (Eclipse)



Creazione dell'header file

Creazione Progetto

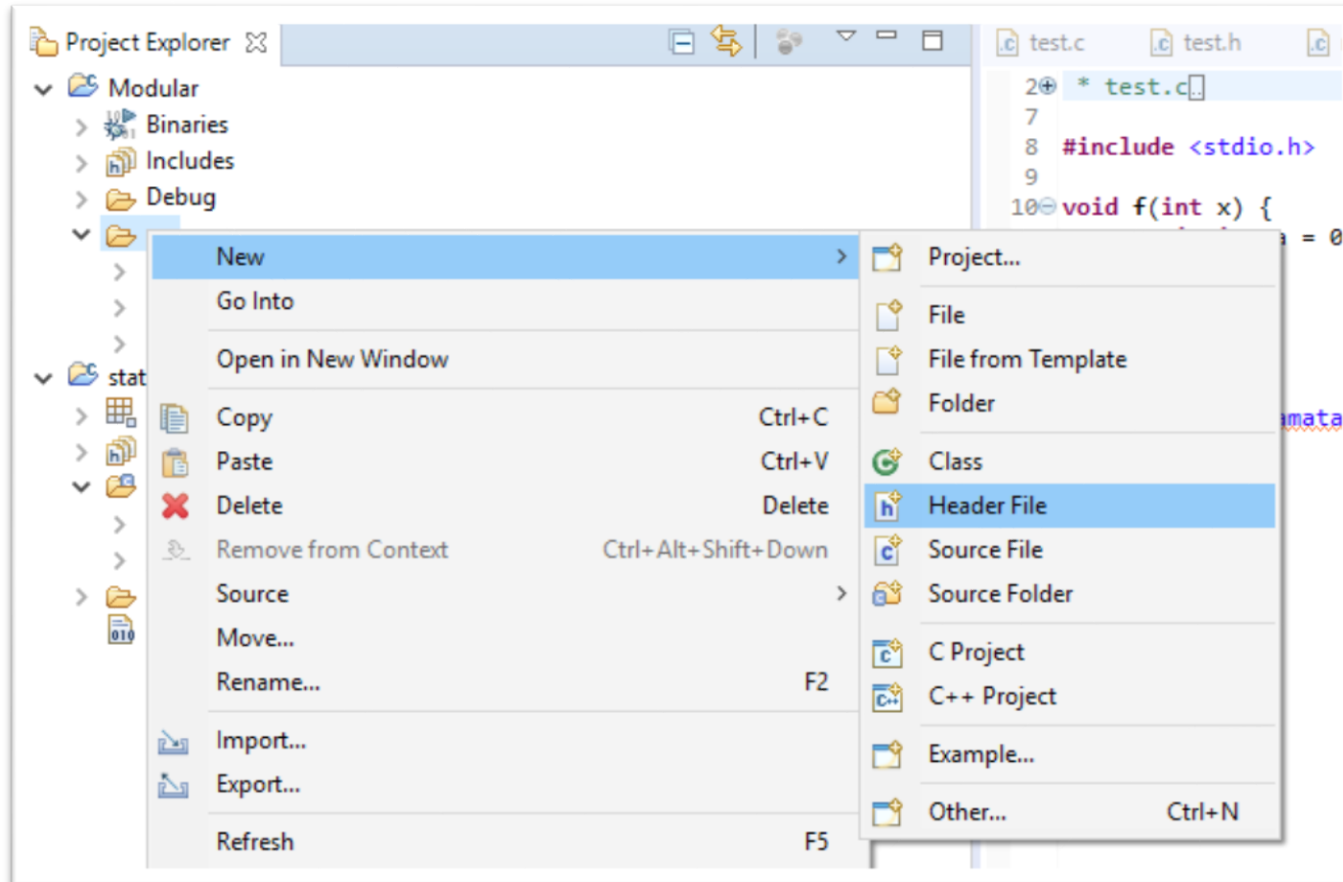
→ Tasto Destro sulla cartella dei sorgenti

→ New

→ Header File

(importante, non dimenticare l'estensione del file!)

Progettazione Modulare – Header Files (Eclipse)



Creazione del file implementazione

Creazione Progetto

→ Tasto Destro sulla
cartella dei sorgenti

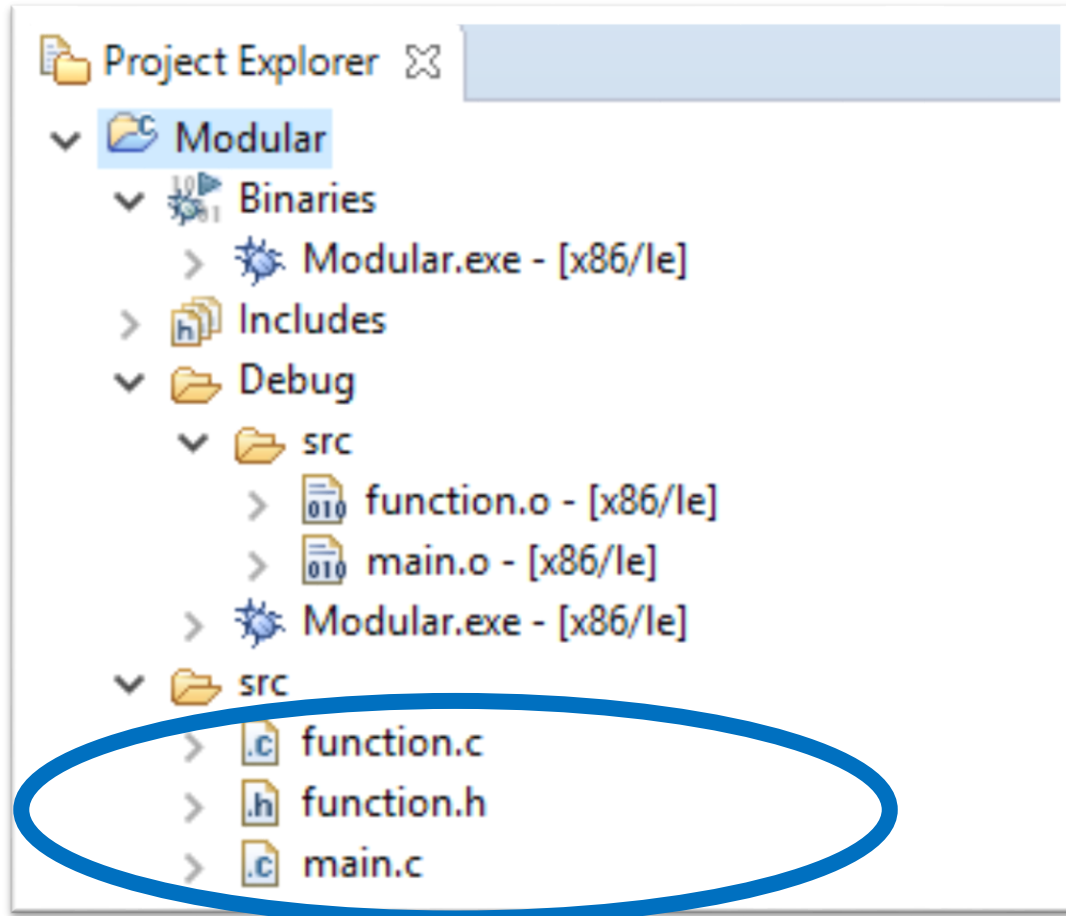
→ New

→ Source File

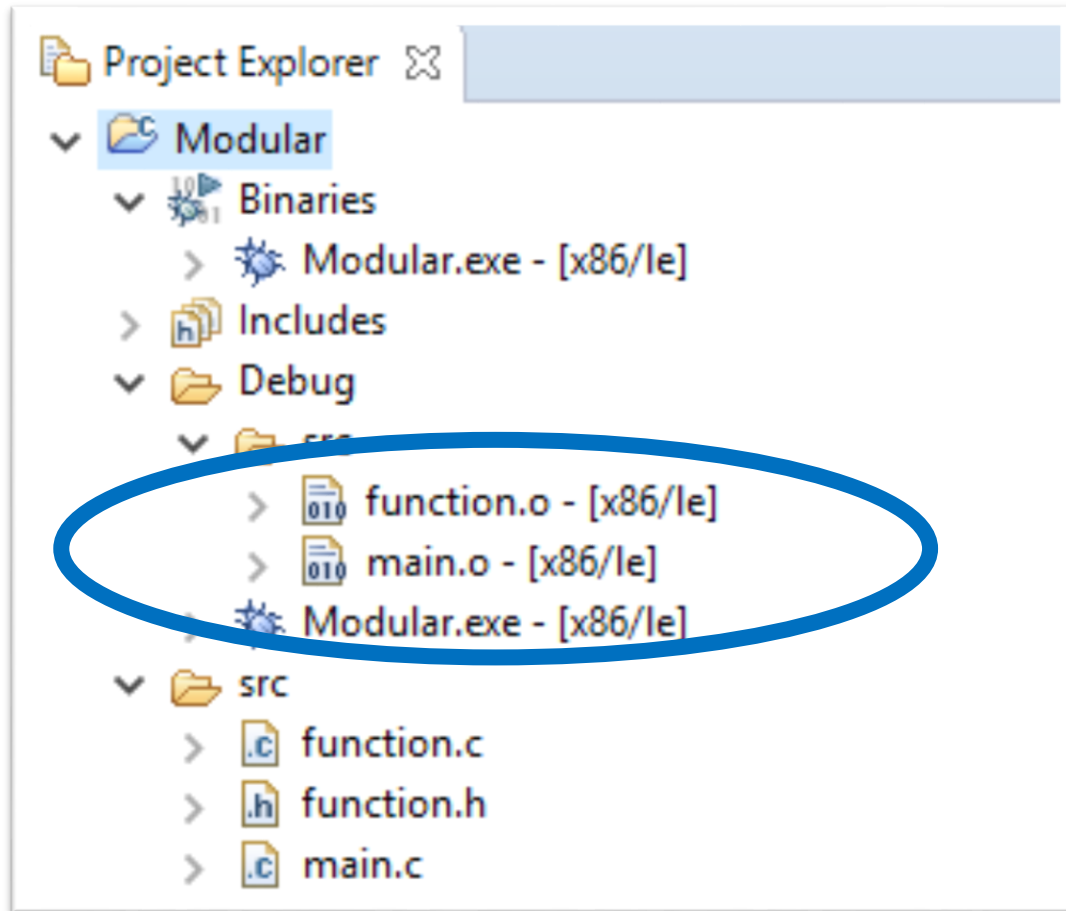
(importante, non dimenticare
l'estensione del file!)

Progettazione Modulare – Header Files (Eclipse)

Struttura del progetto



Progettazione Modulare – Header Files (Eclipse)

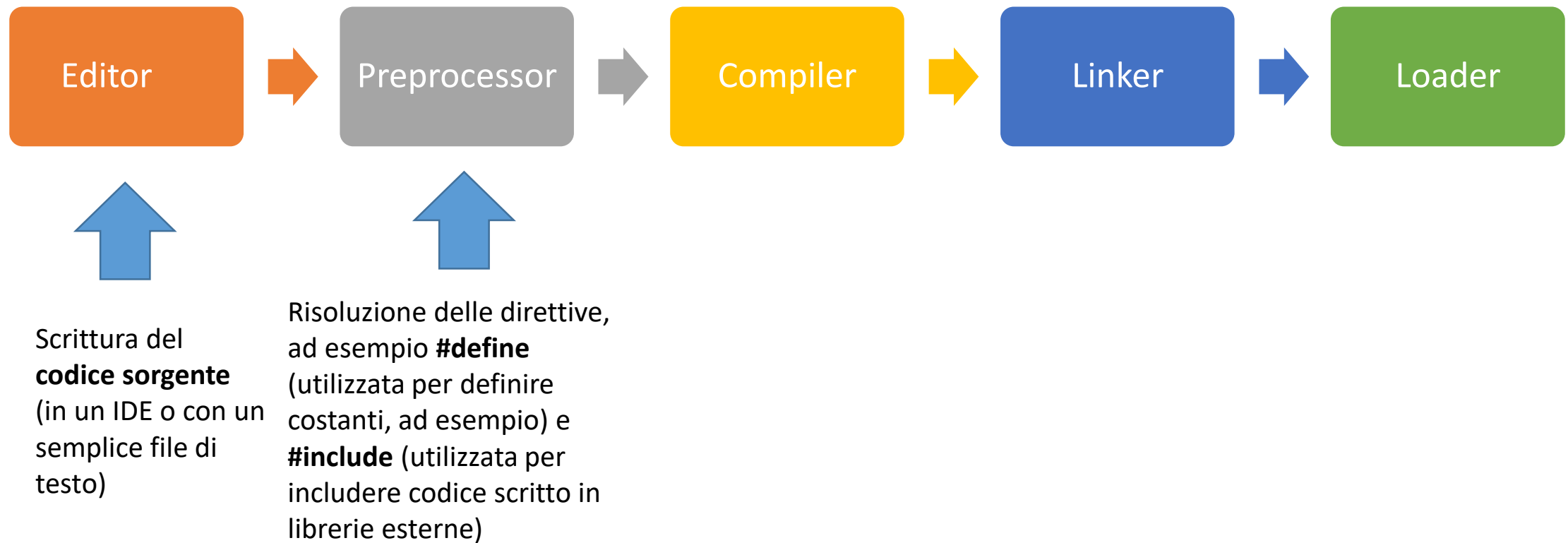


Struttura del progetto

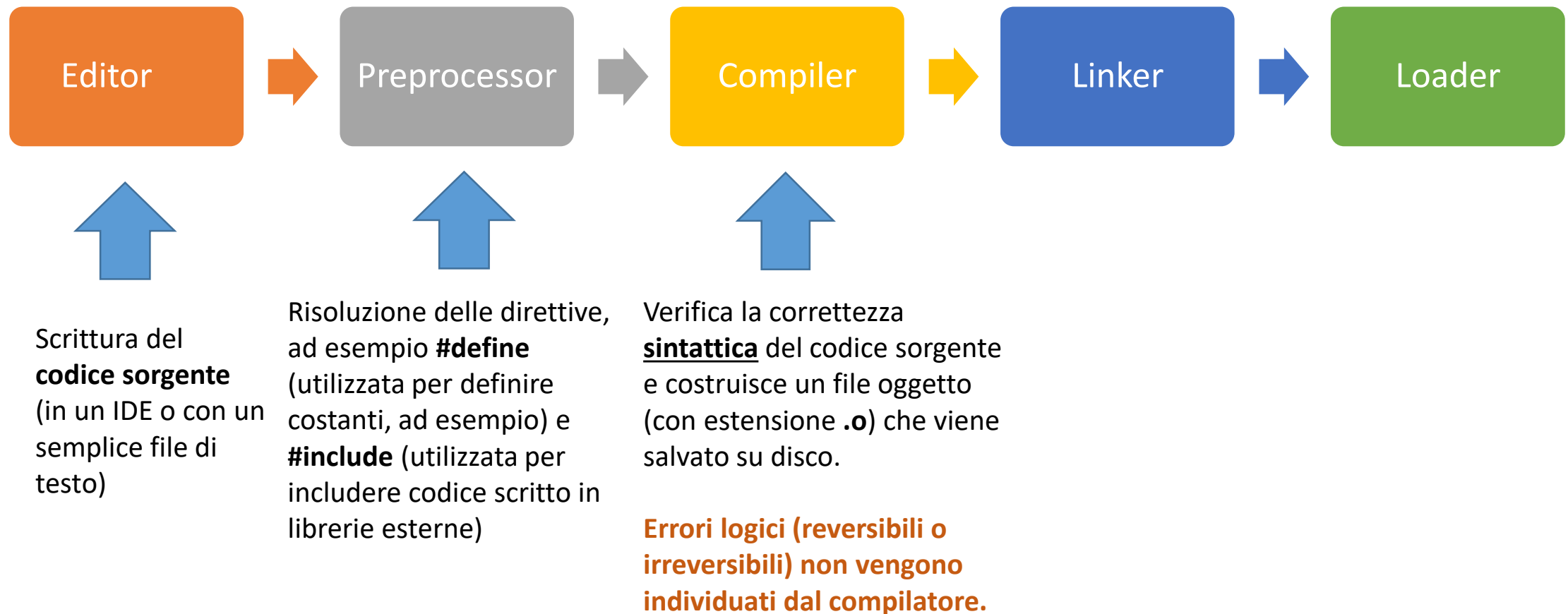
E' interessante notare che quando compiliamo un progetto **anche il file di implementazione viene compilato e crea il suo file oggetto.**

Perché?

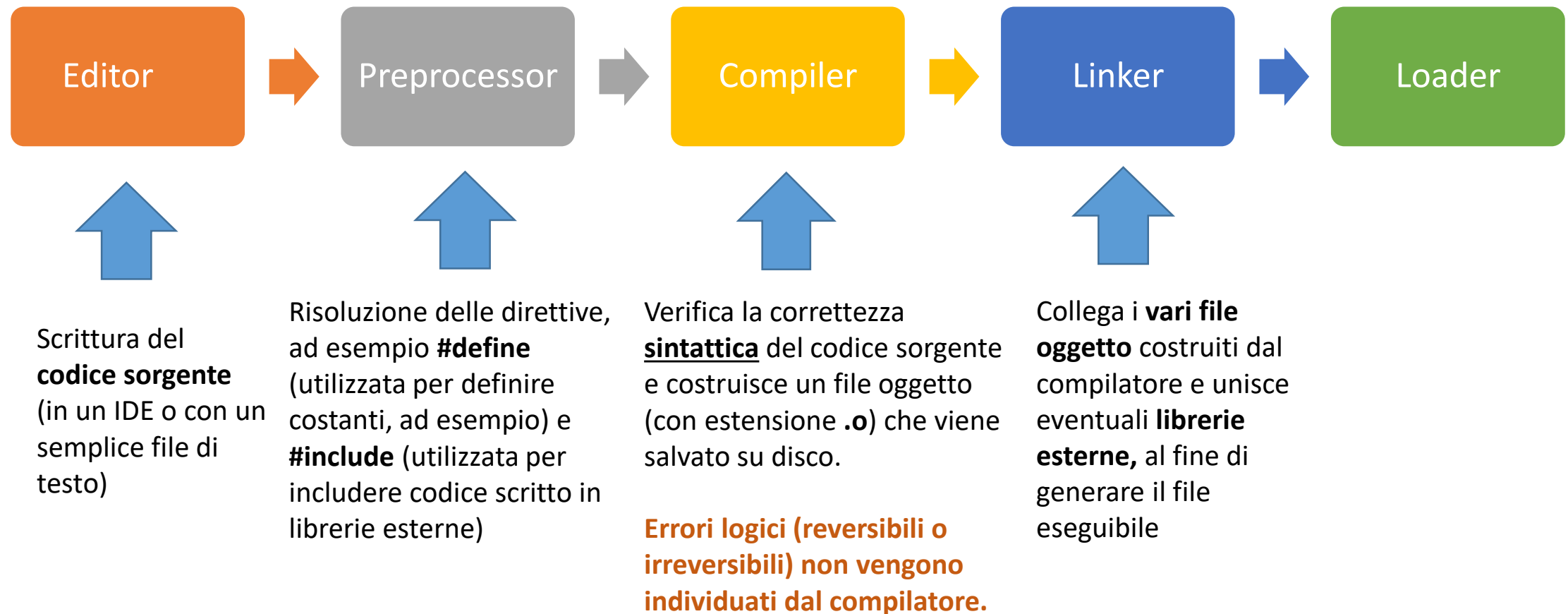
Recap: compilazione codice sorgente



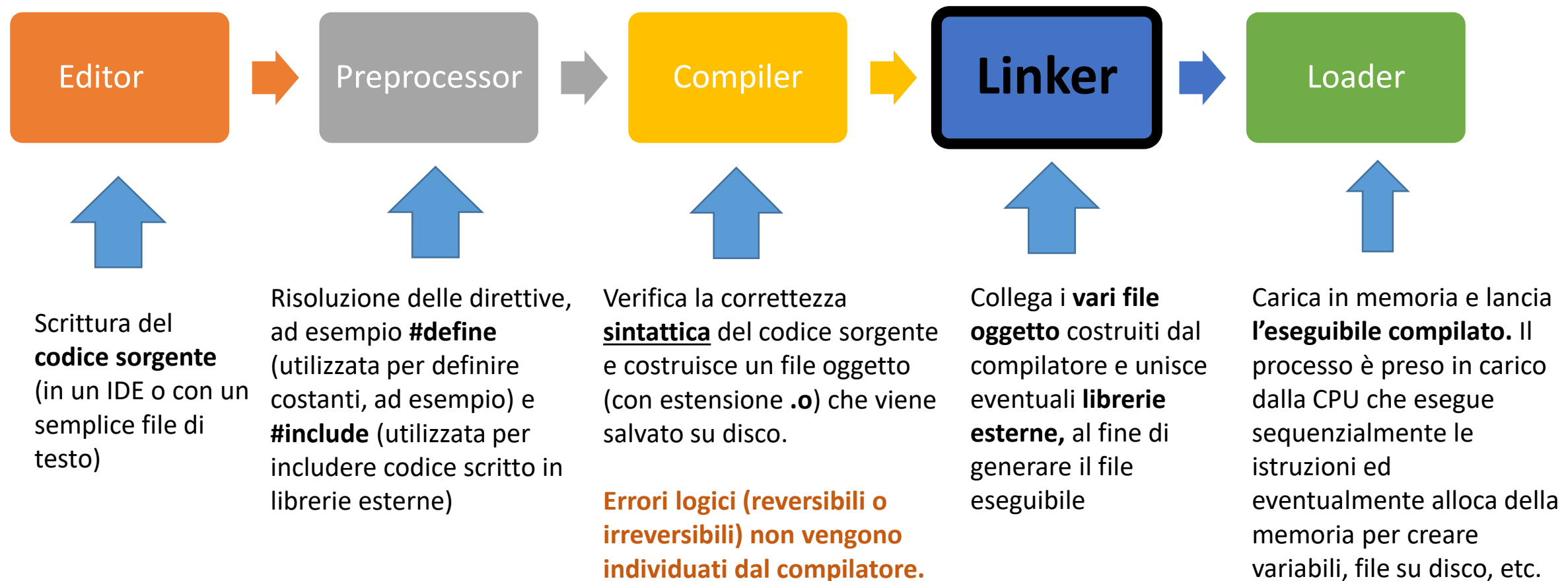
Recap: compilazione codice sorgente



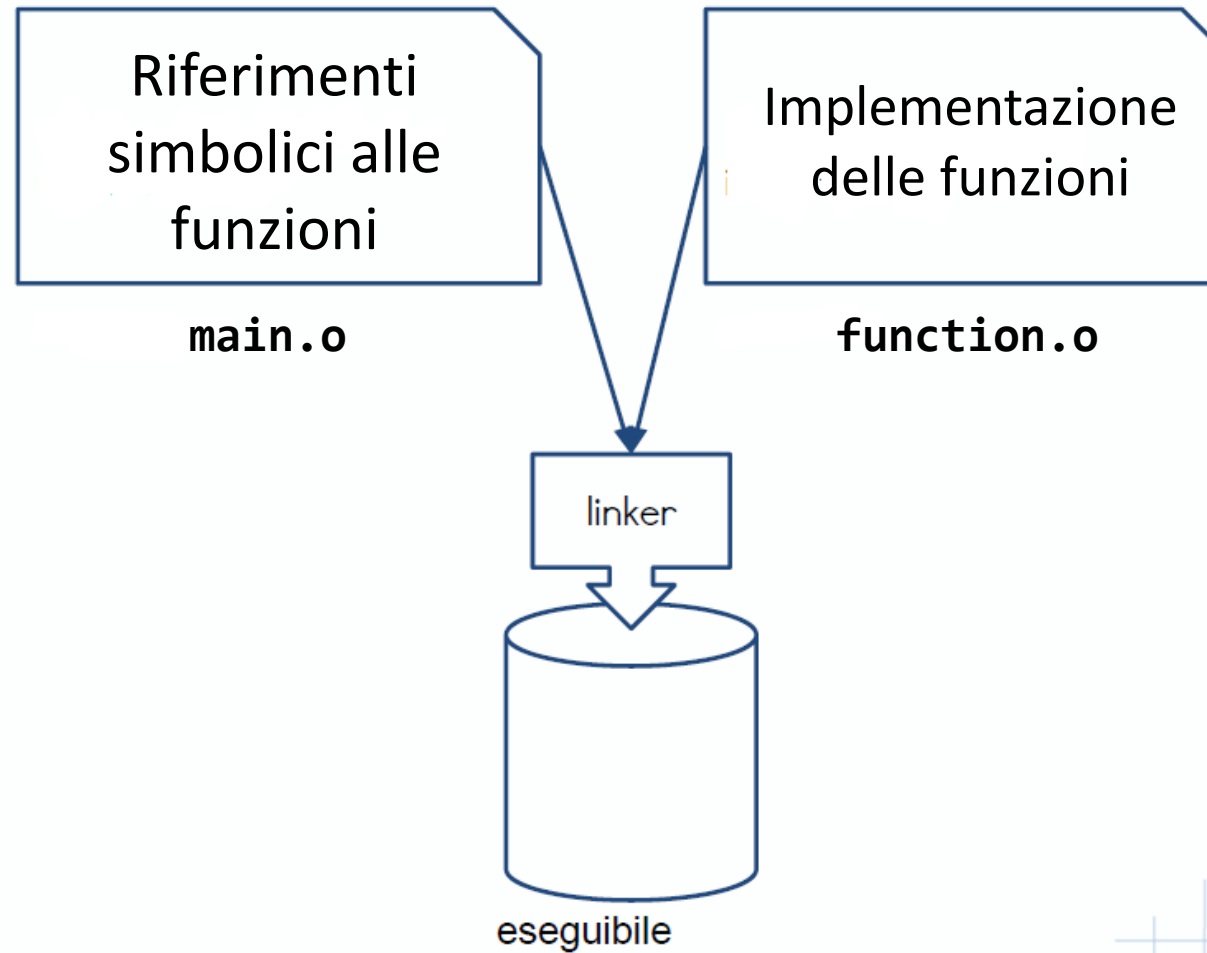
Recap: compilazione codice sorgente



Recap: compilazione codice sorgente

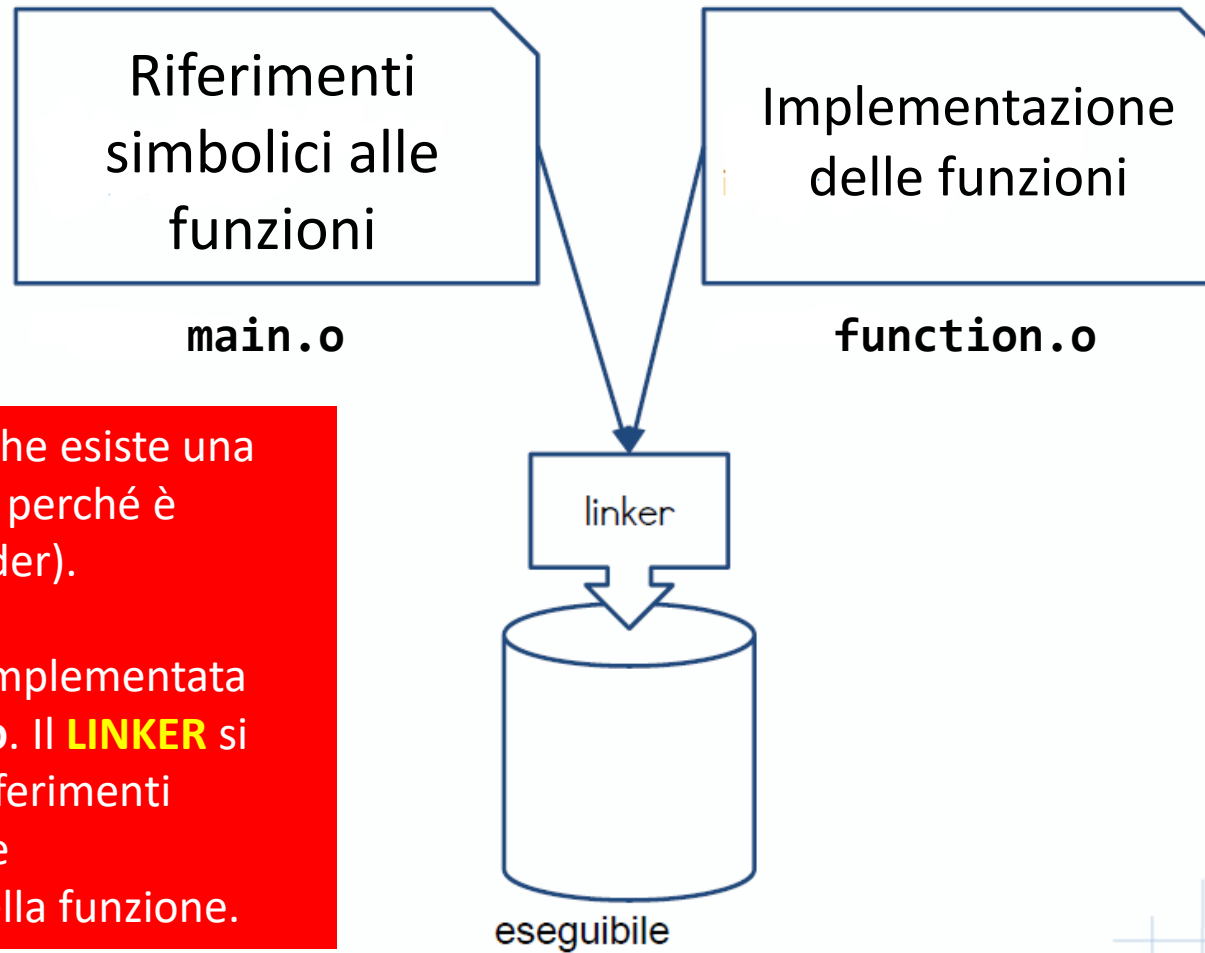


Recap: compilazione codice sorgente



Thanks to Corrado Mencar

Recap: compilazione codice sorgente



Il file **main.o** «sa» che esiste una **funzione f** (lo sa perché è indicato nel file header).

La funzione però è implementata nel file **function.o**. Il **LINKER** si occupa di linkare i riferimenti simbolici con la reale implementazione della funzione.

Thanks to Corrado Mencar

Esercizio 7.1

- Implementare il seguente programma, utilizzando i principi della programmazione modulare

1. Costruire una libreria di funzioni matematiche `math.h`, che implementi le seguenti funzioni

- | | |
|--------------------------------------|---------------------------------------|
| • <code>int somma(int,int)</code> | <code>int sottrazione(int,int)</code> |
| • <code>int prodotto(int,int)</code> | <code>float divisione(int,int)</code> |
| • <code>int quadrato(int)</code> | <code>int cubo(int)</code> |
| • <code>int dispari(int)</code> | <code>int pari(int)</code> |

Esercizio 7.1

- Implementare il seguente programma, utilizzando i principi della programmazione modulare

2. Invocare la libreria in un file **main.c**, che generi **random** due numeri **a** e **b** e calcoli il valore della seguente espressione (attenti all'utilizzo delle parentesi e alla priorità tra gli operatori)

$$(a^2 * b) - (a + b^3)$$

3. Stampare un messaggio diverso a seconda che il risultato sia pari o dispari, utilizzando la funzione implementata
4. Implementare la soluzione sia su Repl o su Eclipse (attraverso la divisione in file .h e .c). **Implementare controlli sull'input! Attenzione a commenti e stile di programmazione.**

Esercizio 7.1 (Soluzione)

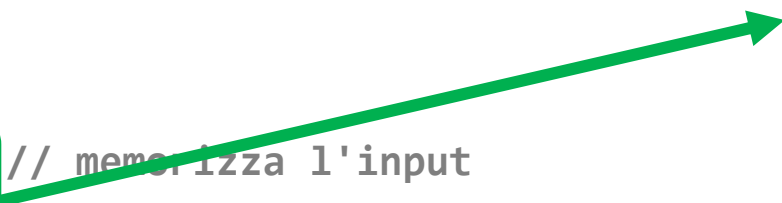
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "math.h" // // importo la libreria appena creata
#define SIZE 64 // fisso la dimensione massima dell'input

int main( ) {
    char input_A[SIZE] = {0}; // memorizza l'input
    char input_B[SIZE] = {0}; // memorizza l'input
    char* remainingInput_A = NULL; // serve a memorizzare quello che resta dell'input
    char* remainingInput_B = NULL; // serve a memorizzare quello che resta dell'input
    int inputCorrect = 0; // serve a uscire dal ciclo appena l'input è corretto
```

Esercizio 7.1 (Soluzione)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "math.h" // // importo la libreria appena creata
#define SIZE 64 // fisso la dimensione massima dell'input

int main( ) {
    char input_A[SIZE] = {0}; // memorizza l'input
    char input_B[SIZE] = {0}; // memorizza l'input
    char* remainingInput_A = NULL; // serve a memorizzare quello che resta dell'input
    char* remainingInput_B = NULL; // serve a memorizzare quello che resta dell'input
    int inputCorrect = 0; // serve a uscire dal ciclo appena l'input è corretto
```



**Leggo delle stringhe,
non degli interi!**

Esercizio 7.1 (Soluzione – cont.)

```
do {
    printf("\nInserire due valori numerici in input, separati da spazio: " );
    scanf("%20s %20s",input_A, input_B);

    int a = (int) strtol(input_A, &remainingInput_A, 0);
    int b = (int) strtol(input_B, &remainingInput_B, 0);

    // l'input è corretto quando la parte rimanente della stringa è vuota
    if( strcmp(remainingInput_A, "") == 0 && strcmp(remainingInput_B, "") == 0 )
        inputCorrect = 1;

    // calcolo l'espressione (a^2*b)-(a+b^3) e stampo il risultato
    if ( inputCorrect ) {
        int result = sottrazione( prodotto( quadrato(a), b), somma( a, cubo(b)));
        printf("Risultato: %d", result);
    }
} while( inputCorrect == 0 ); // cicla finchè l'input non è corretto
}
```

Esercizio 7.1 (Soluzione – cont.)

```
do {  
    printf("\nInserire due valori numerici in input, separati da spazio: " );  
    scanf("%20s %20s",input_A, input_B);  
  
    int a = (int) strtol(input_A, &remainingInput_A, 0);  
    int b = (int) strtol(input_B, &remainingInput_B, 0);  
  
    // l'input è corretto quando la parte rimanente della stringa è vuota  
    if( strcmp(remainingInput_A, "") == 0 && strcmp(remainingInput_B, "") == 0 )  
        inputCorrect = 1;  
  
    // calcolo l'espressione (a^2*b)-(a+b^3) e stampo il risultato  
    if ( inputCorrect ) {  
        int result = sottrazione( prodotto( quadrato(a), b), somma( a, cubo(b)));  
        printf("Risultato: %d", result);  
    }  
} while( inputCorrect == 0 ); // cicla finchè l'input non è corretto  
}
```

Utilizzo `strtol()` per convertire la stringa in intero (serve il cast perché restituisce un long)

Esercizio 7.1 (Soluzione – cont.)

```
do {  
    printf("\nInserire due valori numerici in input, separati da spazio: " );  
    scanf("%20s %20s",input_A, input_B);  
  
    int a = (int) strtol(input_A, &remainingInput_A, 0);  
    int b = (int) strtol(input_B, &remainingInput_B, 0);  
  
    // l'input è corretto quando la parte rimanente della stringa è vuota  
    if( strcmp(remainingInput_A, "") == 0 && strcmp(remainingInput_B, "") == 0 )  
        inputCorrect = 1;  
  
    // calcolo l'espressione (a^2*b)-(a+b^3) e stampo il risultato  
    if ( inputCorrect ) {  
        int result = sottrazione( prodotto( quadrato(a), b), somma( a, cubo(b)));  
        printf("Risultato: %d", result);  
    }  
} while( inputCorrect == 0 ); // cicla finchè l'input non è corretto  
}
```

La parte numerica viene memorizzata in a oppure b, la parte rimanente viene memorizzata nella stringa remainingInput

Esercizio 7.1 (Soluzione – cont.)

```
do {  
    printf("\nInserire due valori numerici in input, separati da spazio: " );  
    scanf("%20s %20s",input_A, input_B);  
  
    int a = (int) strtol(input_A, &remainingInput_A, 0);  
    int b = (int) strtol(input_B, &remainingInput_B, 0);  
  
    // l'input è corretto quando la parte rimanente della stringa è vuota  
    if( strcmp(remainingInput_A, "") == 0 && strcmp(remainingInput_B, "") == 0 )  
        inputCorrect = 1;  
  
    // calcolo l'espressione (a^2*b)-(a+b^3) e stampo il risultato  
    if ( inputCorrect ) {  
        int result = sottrazione( prodotto( quadrato(a), b), somma( a, cubo(b)));  
        printf("Risultato: %d", result);  
    }  
} while( inputCorrect == 0 ); // cicla finchè l'input non è corretto  
}
```

Esempio:
Input_A = «456aa»
A = 456
remainingInput_A = «aa»

Esercizio 7.1 (Soluzione – cont.)

```
do {  
    printf("\nInserire due valori numerici in input, separati da spazio: " );  
    scanf("%20s %20s",input_A, input_B);  
  
    int a = (int) strtol(input_A, &remainingInput_A, 0);  
    int b = (int) strtol(input_B, &remainingInput_B, 0);  
  
    // l'input è corretto quando la parte rimanente della stringa è vuota  
    if( strcmp(remainingInput_A, "") == 0 && strcmp(remainingInput_B, "") == 0 )  
        inputCorrect = 1;  
  
    // calcolo l'espressione (a^2*b)-(a+b^3) e stampo il risultato  
    if ( inputCorrect ) {  
        int result = sottrazione( prodotto( quadrato(a), b), somma( a, cubo(b)));  
        printf("Risultato: %d", result);  
    }  
} while( inputCorrect == 0 ); // cicla finchè l'input non è corretto  
}
```

Esempio:
Input_A = «118»
A = 118
remainingInput_A = «»

Esercizio 7.1 (Soluzione – cont.)

Se remainingInput è «vuota», allora vuole dire che quello che ho letto è corretto!

```
do {
    printf("\nInserire due valori numerici in input, separati da spazio: " );
    scanf("%20s %20s",input_A, input_B);

    int a = (int) strtol(input_A, &remainingInput_A, 0);
    int b = (int) strtol(input_B, &remainingInput_B, 0);

    // l'input è corretto quando la parte rimanente della stringa è vuota
    if( strcmp(remainingInput_A, "") == 0 && strcmp(remainingInput_B, "") == 0 )
        inputCorrect = 1;

    // calcolo l'espressione (a^2*b)-(a+b^3) e stampo il risultato
    if ( inputCorrect ) {
        int result = sottrazione( prodotto( quadrato(a), b), somma( a, cubo(b)));
        printf("Risultato: %d", result);
    }
} while( inputCorrect == 0 ); // cicla finchè l'input non è corretto
}
```


Esercizio 7.1 (Soluzione – cont.)

Se l'input è corretto, posso calcolare il valore dell'espressione e stamparlo

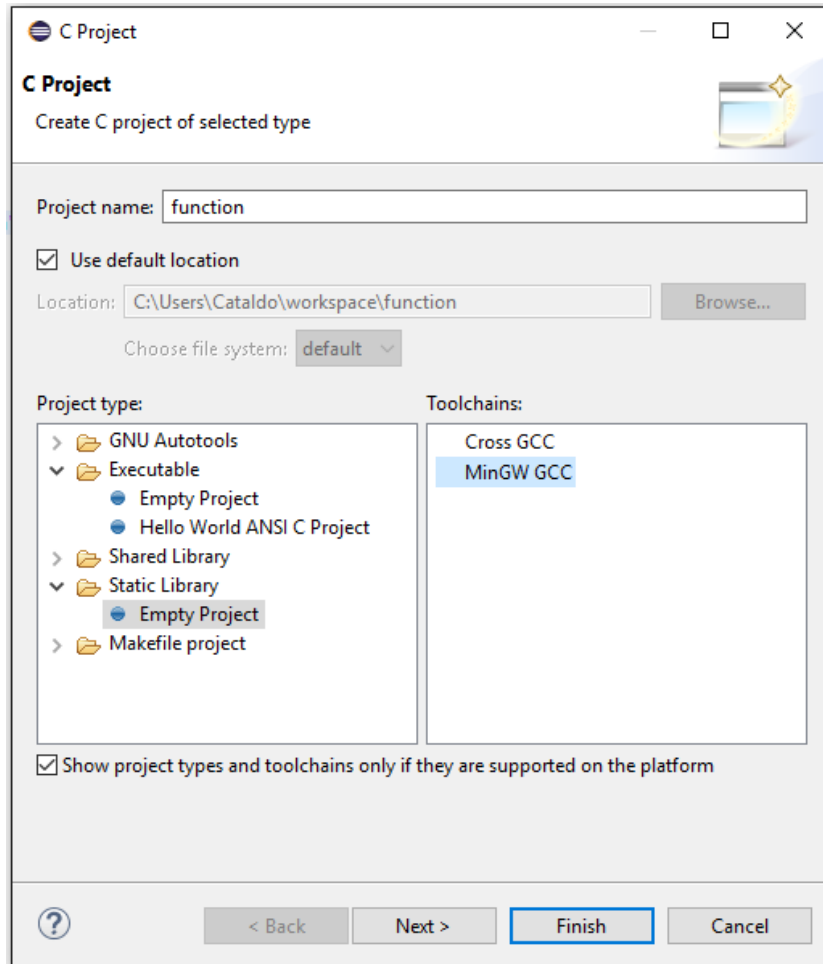
```
do {  
    printf("\nInserire due valori numerici in input, separati da spazio: " );  
    scanf("%20s %20s",input_A, input_B);  
  
    int a = (int) strtol(input_A, &remainingInput_A, 0);  
    int b = (int) strtol(input_B, &remainingInput_B, 0);  
  
    // l'input è corretto quando la parte rimanente della stringa è vuota  
    if( strcmp(remainingInput_A, "") == 0 && strcmp(remainingInput_B, "") == 0 )  
        inputCorrect = 1;
```

```
    // calcolo l'espressione (a^2*b)-(a+b^3) e stampo il risultato  
    if ( inputCorrect ) {  
        int result = sottrazione( prodotto( quadrato(a), b), somma( a, cubo(b)));  
        printf("Risultato: %d", result);  
    }
```

```
} while( inputCorrect == 0 ); // cicla finché l'input non è corretto
```

```
}
```

Progettazione Modulare – Librerie Statiche (in Eclipse)



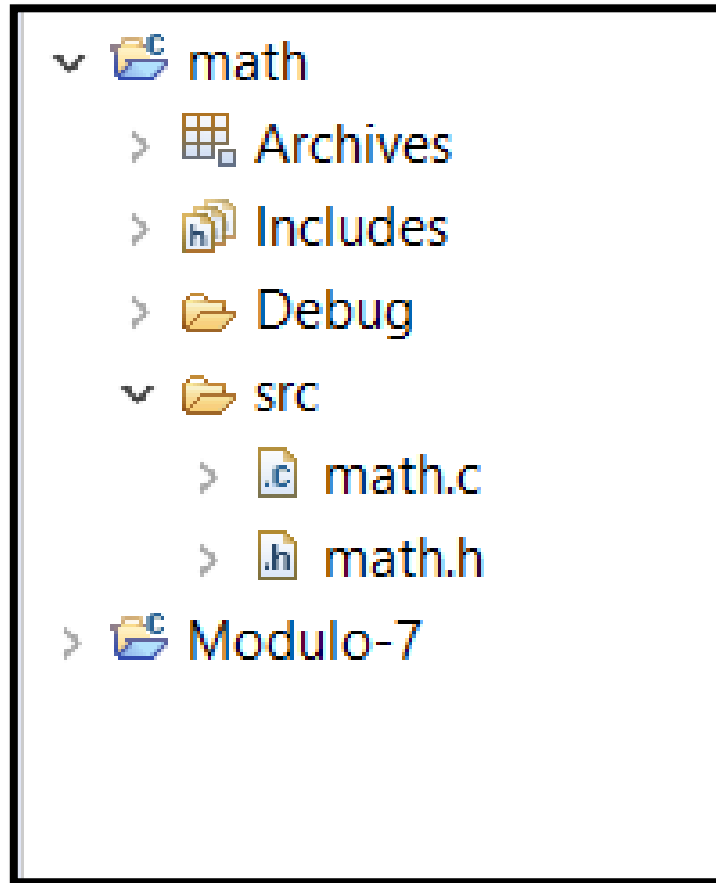
Modalità alternativa

per lo sviluppo di librerie esterne

**New → C Project → Static Library
→ Empty Project**

Dare un nome e selezionare **'MiniGW GCC'** come
Toolchains.

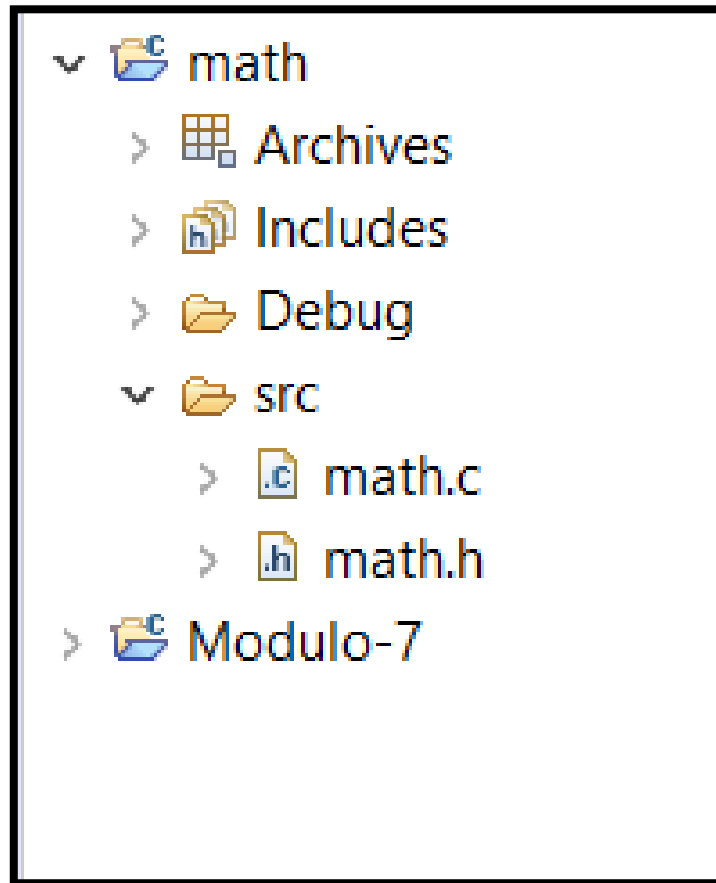
Progettazione Modulare – Librerie Statiche (in Eclipse)



Creare una cartella in cui inserire i file .h e .c

Tasto dx sul progetto → New →
Source Folder → dare un nome (es.
src)

Progettazione Modulare – Librerie Statiche (in Eclipse)



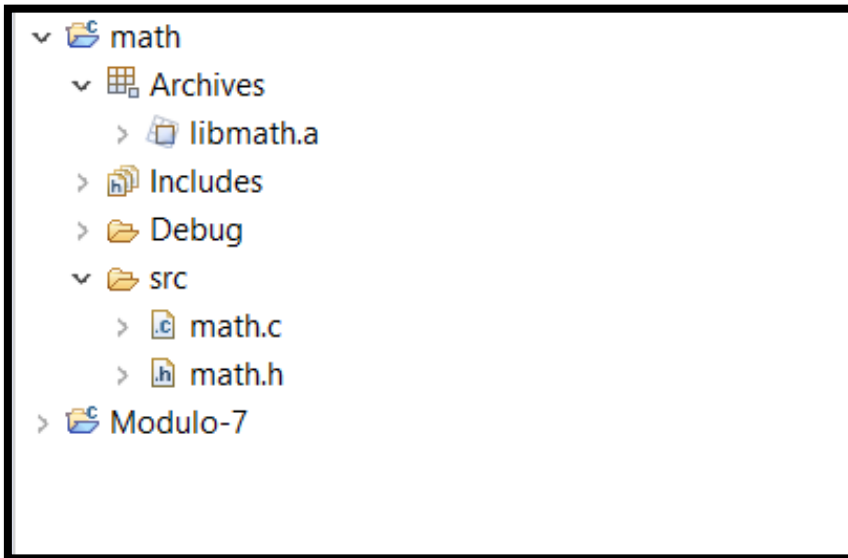
Creare una cartella in cui inserire i file .h e .c

Tasto dx sul progetto → New → Source Folder → dare un nome (es. src)

Si implementano i file **.h** e **.c** esattamente come nel caso precedente.

Terminata l'implementazione, si fa il **«build»** della **libreria statica (simbolo del martello)**

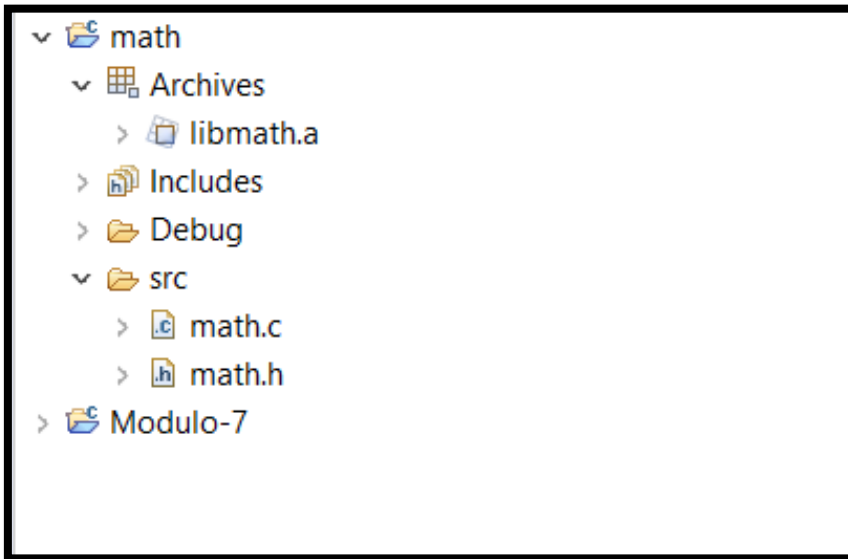
Progettazione Modulare – Librerie Statiche (in Eclipse)



Il build genera un file **.a** (archive) che rappresenta la **libreria statica**.

La libreria statica si può integrare in altri progetti

Progettazione Modulare – Librerie Statiche (in Eclipse)



Il build genera un file **.a** (archive) che rappresenta la **libreria statica**.

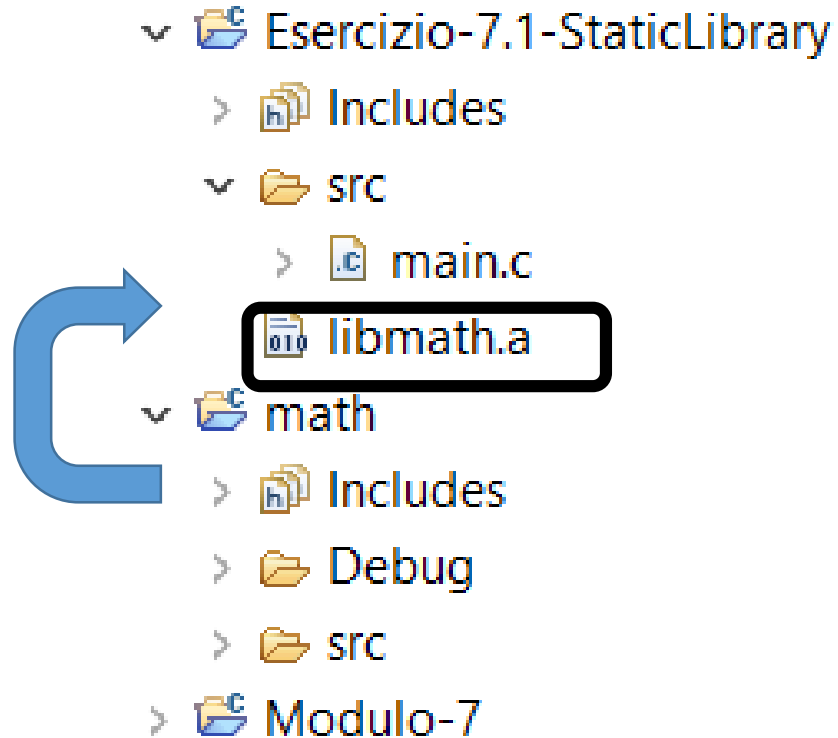
La libreria statica si può integrare in altri progetti

Come?

- 1) Si copia la libreria statica nel progetto**
- 2) Si aggiungono i riferimenti al file .h**

In questo caso non la libreria **viene sviluppata in modo totalmente separato dagli altri file**, e si integra semplicemente **«copiando» il file archivio** all'interno del progetto.

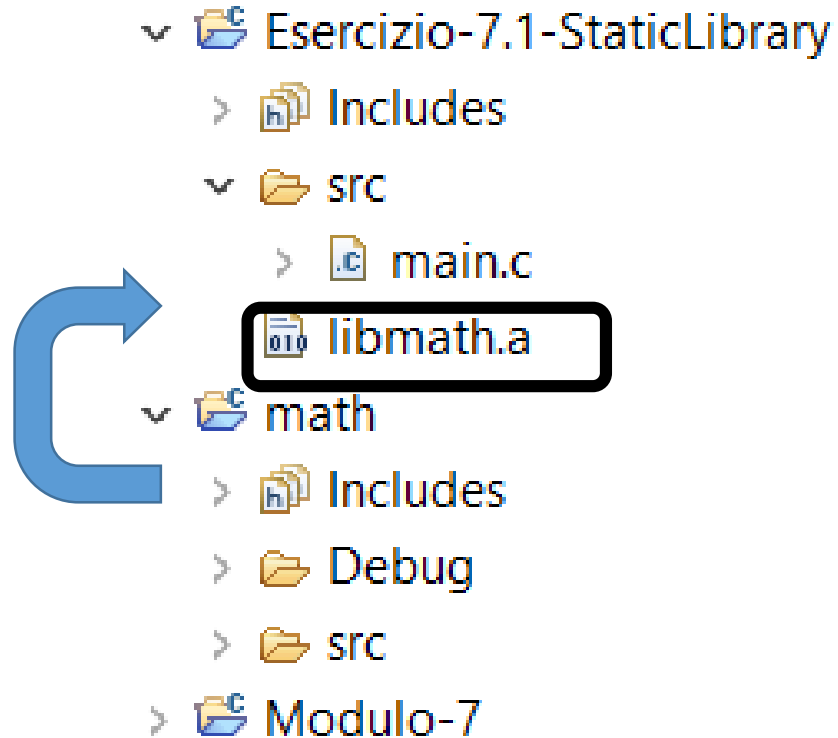
Progettazione Modulare – Librerie Statiche (in Eclipse)



Come?

1) Si copia la libreria statica nel progetto

Progettazione Modulare – Librerie Statiche (in Eclipse)



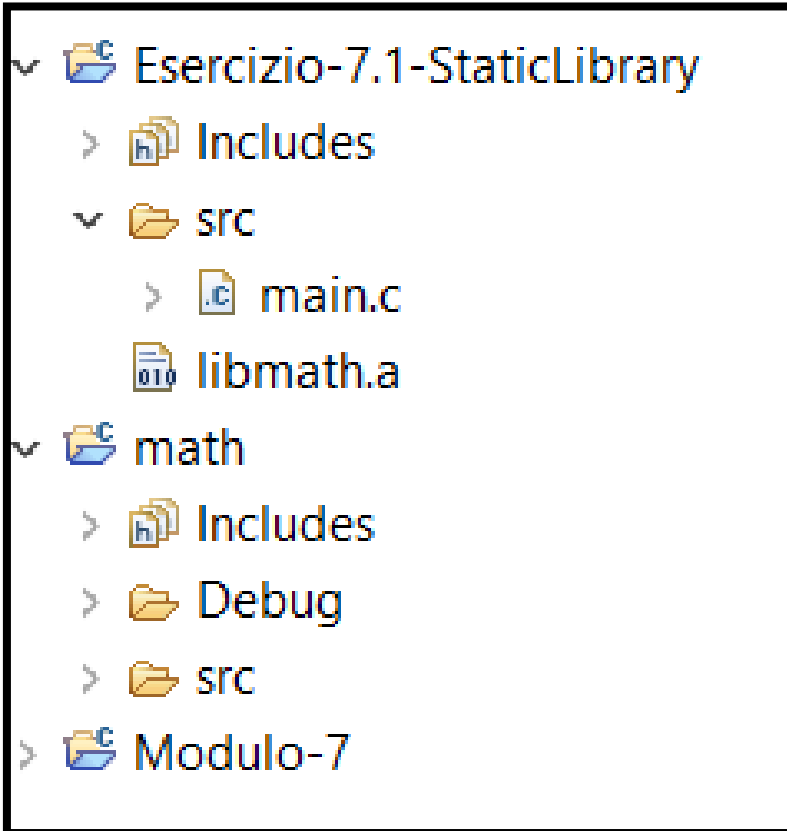
Come?

1) Si copia la libreria statica nel progetto

Abbiamo due progetti, uno con il file **main.c** (non ci sono più i file con la libreria esterna!) e una **libreria statica** implementata come progetto esterno.

L'archivio con estensione .a, output della libreria statica, viene copiato nel primo progetto

Progettazione Modulare – Librerie Statiche (in Eclipse)



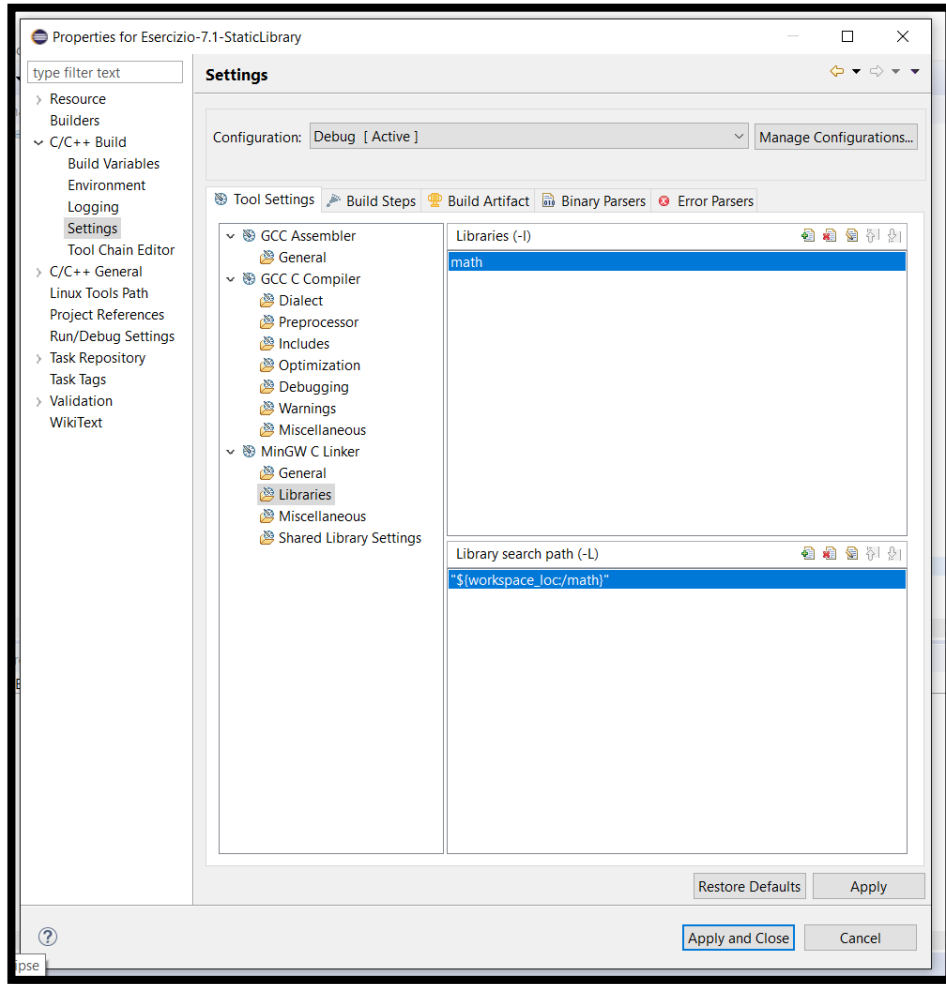
Cosa manca?

2) Si aggiungono i riferimenti al file .h

2.1 Bisogna far sapere al compilatore che «esiste» questa libreria statica

2.2 Bisogna far sapere al compilatore che funzioni implementa!

Progettazione Modulare – Librerie Statiche (in Eclipse)



Cosa manca?

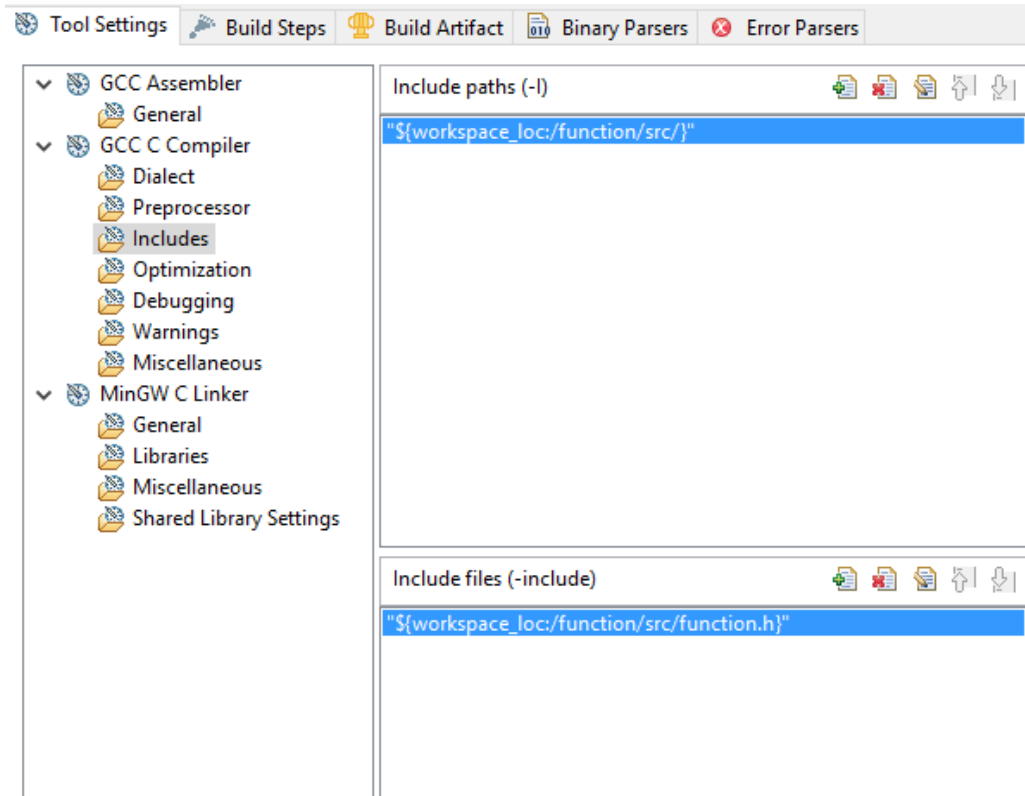
2) Si aggiungono i riferimenti al file .h

2.1 Bisogna far sapere al compilatore che «esiste» questa libreria statica

Tasto Dx sul progetto → Properties → C/C++ Build → Settings → MinGW C Linker → Libraries

In **Libraries** si aggiunge (cliccando sul +) il **nome della libreria**, mentre in **Library Search path** si clicca su **Workspace** e si seleziona la cartella di progetto in cui è presente la libreria che abbiamo importato

Progettazione Modulare – Librerie Statiche (in Eclipse)



Cosa manca?

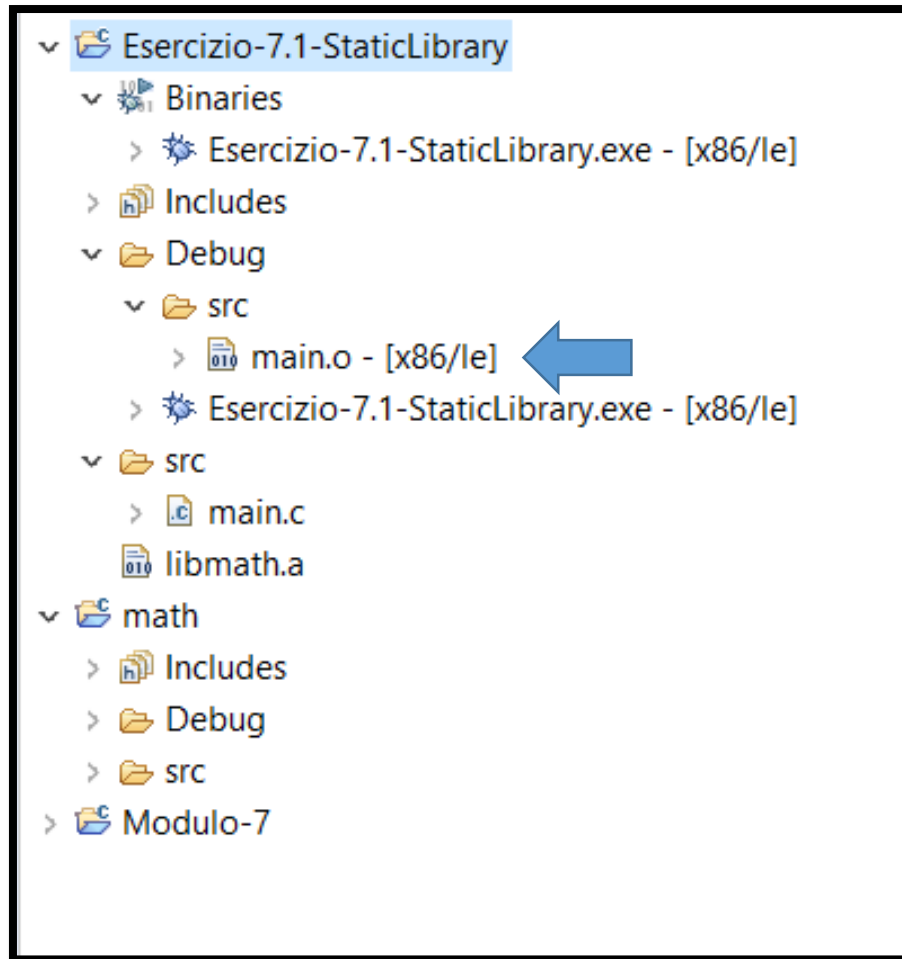
2) Si aggiungono i riferimenti al file .h

2.2 Bisogna far sapere al compilatore che funzioni implementa!

Tasto Dx sul progetto → Properties → C/C++
Build → Settings → GCC C Compiler → Libraries

In **Includes** si aggiunge (cliccando sul +) il **path del file .h**, mentre in **Include files** si clicca su **Workspace** e si seleziona proprio il file .h che serve al compilatore per sapere **che funzioni sono disponibili nella libreria statica**

Progettazione Modulare – Librerie Statiche (in Eclipse)



Terminato il processo di linking manuale della libreria, è possibile eseguire normalmente il progetto.

Importante: in questo caso non abbiamo due file .o , ma un unico file (il main). Questo avviene perché gli altri file sono inclusi nella libreria che abbiamo precedentemente linkato.



Esercizio 7.2

- **Migliorare l'implementazione dell'Esercitazione 2**

- 1) Modularizzare il codice

- Definire **funzioni e procedure** per le varie funzionalità
- Valutare le diverse **implementazioni** possibili
 - Un'unica funzione per controllare nome, cognome e indirizzo? Una funzione separata per ogni controllo?
- Valutare **le diverse intestazioni** delle funzioni
 - Nella funzione che restituisce il grado di sicurezza della funzione, i valori di sicurezza (≥ 2 , ≥ 4 , etc.) sono dei parametri o sono già presenti nel codice?
- **Motivare e Giustificare** le scelte (maggiore leggibilità del codice, maggiore riuso ma minore modularità, etc.)