

APPUNTI AGGIUNTIVI SULLE FUNZIONI IN C

UTILIZZO DI PARAMETRI PASSATI PER INDIRIZZO PER REALIZZARE FUNZIONI CHE RESTITUISCONO COME OUTPUT PIU' DI UN VALORE

Come è noto le funzioni possono restituire un solo valore di ritorno, per fare ciò utilizzano l'istruzione **return**.

Grazie al passaggio di parametri per indirizzo è possibile realizzare funzioni che **producono più di un risultato di output**.

Per ottenere più valori di output bisogna

- dichiarare le variabili che conterranno i valori di output nella funzione chiamante (solitamente nella funzione main)
- passare le due variabili di output come parametri per indirizzo alla funzione
- realizzare la funzione in modo che i risultati di output siano assegnate ai parametri di output passati per indirizzo

Esempio: funzione che riceve come input un numero intero n e restituisce due valori di output: la metà di x ed il doppio di x.

```
4 void metaDoppio(int x, int *m, int *d)
5 {
6     *m=x/2;
7     *d=x*2;
8 }
9
10 int main()
11 {
12     int n;
13     int meta, doppio;
14     printf("\ninserisci n: ");
15     scanf("%i", &n);
16     metaDoppio (n, &meta, &doppio);
17     printf("\nLa meta di %i e' %i: ", n, meta);
18     printf("\nIl doppio di %i e %i': ", n, doppio);
19     return 0;
20 }
```

1 Le variabili meta e doppio vengono dichiarate nel main

2 Le variabili meta e doppio vengono passate per indirizzo alla funzione

3 Le variabili "m" e "d" puntano alle variabili "meta" e "doppio" del main, quindi il valore finale di m e d sarà anche il valore delle variabili meta e doppio nel main

Esercizio funzione maxMin (sul passaggio di parametri per indirizzo): realizzare una funzione che, dati come input quattro numeri interi, restituisce il maggiore e il minore fra i valori inseriti .

Molto spesso l'istruzione return di una funzione viene utilizzata per restituire alla funzione invocante un codice che indica se la funzione è stata eseguita correttamente oppure se si è verificato un errore nel passaggio dei parametri o nella computazione. Solitamente 0 indica "tutto ok", funzione eseguita correttamente.

Esempio (p.140 libro di informatica) radice quadrata (sull'utilizzo dell'istruzione return per restituire un codice relativo all'esito positivo o meno dell'esecuzione di una funzione): realizziamo una funzione per il calcolo della radice quadrata.

Sappiamo che è possibile calcolare la radice quadrata solamente per i numeri maggiori o uguali a 0. La nostra funzione restituirà (con il return) un valore 0 se l'argomento della radice è maggiore o uguale a 0, restituirà -

1 se l'argomento è negativo. Il risultato della radice quadrata verrà restituito con un parametro passato per indirizzo.

```

4  int radiceQuadrata(float x, float *r)
5  {
6      if (x >= 0)
7      {
8          *r = sqrt(x);
9          return 0;
10     }
11     else
12         return -1;
13 }
14
15
16 int main()
17 {
18     float x;
19     float radice;
20     int esitoFunzione; // 0 --> tutto ok, -1 --> errore
21     printf("\nInserisci x: ");
22     scanf("%f", &x);
23     esitoFunzione = radiceQuadrata(x, &radice);
24     if (esitoFunzione == 0)
25         printf("\nLa radice quadrata di %f e' %f ", x, radice);
26     else
27         printf("\nImpossibile calcolare la radice quadrata di un numero negativo");
28     return 0;
29 }

```

Callouts:

- "r" parametro formale di output passato per indirizzo (points to `float *r` in line 4)
- "x" parametro formale di input passato per valore (points to `float x` in line 4)
- "x" parametro attuale di input passato per valore (points to `x` in line 23)
- "radice" parametro attuale di output passato per valore (points to `&radice` in line 23)

Esercizio: come sappiamo la divisione fra due numeri è possibile solamente se il divisore è diverso da 0. Realizzare una funzione "divisione" che permette di ottenere la divisione fra due float. La funzione deve utilizzare il return per restituire 0 se la divisione viene effettuata, -1 se la divisione non è possibile perché il divisore è 0. Gestire correttamente la funzione nel main, mostrando il risultato della divisione se essa viene eseguita, e mostrando il messaggio "Impossibile dividere per 0" se la funzione non viene eseguita.

Esercizio: (fare): Funzione che a partire da un prezzo e da una percentuale di sconto consente di ottenere il valore dello sconto e il prezzo scontato. La funzione restituisce 0 se tutto OK, restituisce -1 se la percentuale

PROTOTIPIZZAZIONE DELLE FUNZIONI

Una funzione (f3) può invocare un'altra funzione (f2), che a sua volta può invocare un'altra funzione (f1).

In tal caso, poiché il compilatore deve “sapere come è fatta” una funzione (tipo di dato restituito, numero e tipo parametri) prima di invocarla (un po' come quando bisogna dichiarare le variabili prima di utilizzarle), sarà necessario seguire un ordine nella dichiarazione delle funzioni. Quindi **f1 dovrà essere dichiarata per prima, f2 per seconda ed f3 per terza**. Questo perché il **compilatore deve conoscere il tipo di una funzione e dei suoi parametri prima di incontrare la sua invocazione**, altrimenti non saprebbe quanto e quale spazio riservare in memoria per la funzione ed i suoi parametri. Ricorda che il compilatore inizia la traduzione dalla prima riga di codice. Se f1 non fosse definita prima di f2, durante la traduzione, il compilatore incontrerebbe, nel corpo di f2, l'invocazione f1(), ma non sapendo cosa è segnalerebbe un errore: “cosa è f1? una variabile, un int, un double? cosa è?!!”

```
f1()
{
    ...codice
}

f2()
{
    ..codice
    f1()          //invocazione di f1
}

f3
{
    ...codice
    f2()          //invocazione di f2
}

main()
{
    ....codice
    f3()          //invocazione di f3
}
```

Quando un programma è articolato in numerose funzioni **può non essere semplice rispettare questo ordine** da parte del programmatore. Per risolvere il problema si ricorre alla tecnica della **prototipizzazione**.

Questa tecnica consiste nello scrivere un prototipo di ogni funzione prima di dichiararla. Il **prototipo** (chiamato anche **interfaccia** di una funzione) consiste semplicemente nel **header** della funzione, ossia nell'indicazione del suo tipo e dei suoi parametri formali (con rispettivo tipo). Il prototipo di una funzione deve essere terminato con un punto e virgola.

Nel prototipo non è obbligatorio indicare il nome dei parametri della funzione, e se lo si indica non è necessario che sia lo stesso utilizzato poi nella dichiarazione della funzione. E' invece obbligatorio indicare il tipo dei parametri.

Esempio 1 (su come si scrive il prototipo di una funzione): la funzione "double per2(double x)" riceve come input un valore double x e restituisce un double pari al doppio di x. Scriviamo un programma in cui il main invochi la funzione "per2" e in cui creiamo anche il prototipo della funzione "per2".

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  double per2(double x);
5
6  int main()
7  {
8      int x=10;
9      printf("%.2lf", per2(x));
10     return 0;
11 }
12
13 double per2(double x)
14 {
15     return x*2;
16 }
```

Prototipo

Questo è il prototipo della funzione "per2". Per ogni funzione è opportuno scrivere il prototipo. I prototipi delle funzioni vanno scritti tutti all'inizio del file.

Invocazione della funzione

Funzione

A COSA SERVE IL PROTOTIPO?

Mi permette di invocare la funzione in una riga di codice che precede la funzione stessa.

Cosa che non si poteva senza aver definito il prototipo

Una volta scritto il prototipo di una funzione il compilatore ne conosce le caratteristiche, non è quindi più necessario che la funzione invocata sia scritta prima di quella invocante (ciò è molto comodo per il programmatore).

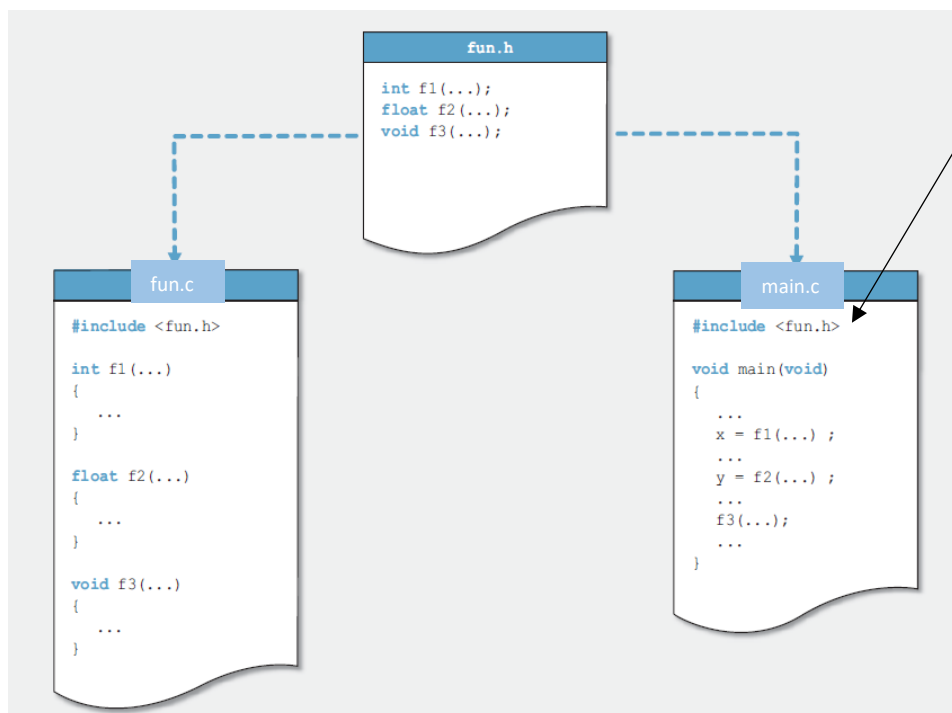
Utilizzando i prototipi è possibile scrivere il codice delle funzioni senza preoccuparsi dell'ordine in cui sono scritte. Senza i prototipi questo non è possibile (**provare** per credere!)

OSSERVAZIONE IMPORTANTE: se la funzione restituisce `int` o `void`, viene compilata anche senza prototipi, semplicemente ricevendo un warning, se invece la funzione restituisce un altro tipo di dato non viene compilata. Questo comportamento dipende dal compilatore quindi **è sempre una buona regola creare il prototipo per ogni funzione.**

Spesso nei programmi di grandi dimensioni o nelle librerie, per dare ordine al codice, si preferisce separare le **interfacce delle funzioni** (ossia i prototipi) dal loro corpo. Per fare questo si creano:

- un file (con estensione `.h` o `.H`, la `h` sta per Header) costituito **solo dai prototipi** delle funzioni chiamato "Header file".
- Un file `.c` con i corpi delle funzioni (nel quale si richiama, all'inizio, con la direttiva `#include` "nome file", il Header file contenente i prototipi).
- Un file principale (`main.c`) contenente la funzione `main()` nella quale vengono "invocate" le funzioni. Anche in questo file è necessario includere con la direttiva `"#Include"` il Header file contenente i prototipi.

La struttura dei file è rappresentata nel seguente schema:



Se dimentico questo, l'invocazione può a volte andare a buon fine (se sono di tipi `int` o `void`) e a volte no. **Per esempio una funzione `double` darà come risultato 0**

OSSERVAZIONE: La direttiva `#include` è sempre seguita dal nome del file header file da includere. Come abbiamo visto finora il nome del file era sempre compreso fra parentesi angolari (`<` e `>`), ad esempio:

```
#include <stdlib.h>
#include <math.h>
```

Le parentesi angolari indicano al preprocessore che il file indicati si trovano in una cartella di default che contiene i file della libreria standard del C.

Quando il programmatore crea dei propri header file, il loro nome va indicato fra virgolette anziché fra parentesi angolari. In tal caso i file verranno cercati dal preprocessore nella cartella di progetto del file. Naturalmente è necessario che affinché il preprocessore “trovi” gli header file, essi devono essere salvati nella cartella di progetto del file.

```
#include "fun.h"
```

E' possibile includere file che si trovano in un'altra cartella specificando l'intero pathname del file fra virgolette.

Come CREARE il file “libreria.c” e il file “libreria.h” in un progetto codeBlocks “miaLibreria”

Creazione del file libreria “libreria.c”

1. Aggiungere il file sorgente “libreria.c” al progetto nella cartella principale (Menu File/new/file/) seleziona **c/c++ source**
2. Nel wizard seleziona entrambe le opzioni (debug e release). La differenza sta nel fatto che con release il codice binario verrà ottimizzato (risparmia spazio, rende più rapide alcune operazioni), questo potrebbe rallentare il debugger. La versione per debug crea un codice binario non ottimizzato ma più veloce da eseguire in debug. Nel nostro caso non cambia nulla.
3. Nel codice del file “libreria.c” aggiungi

```
#include ""libreria.h" (con le virgolette)
```
4. Nel codice del file scrivi il codice delle funzioni da realizzare.

Creazione del file .h

1. Aggiungere il file “libreria.h” al progetto nella cartella principale (Menu File/new/file) seleziona c/c++ Header
2. Scrivere all'inizio del file i prototipi delle funzioni (fra #define e #endif)

Importazione del file .h nel file main.c

1. Scrivere all'inizio del main.c

```
#include "librerie.h"
```

Il risultato è che nella scheda Project della finestra Management si vedranno i seguenti file:



Il codice presente in ciascun file è il seguente:

File **libreria.c** (contiene il codice delle funzioni, nel nostro esempio le funzioni sono “casuale” e “scambia”)

E' necessario includere il header file libreria.h

Quando il file da includere è nella cartella di progetto, il file è compreso fra virgolette anziché fra parentesi angolari.

Le parantesi angolari si usano per includere i file delle librerie standard.

```
1  #include "libreria.h"
2
3  /*
4  funzione che genera un numero intero casuale compreso
5  fra un minimo e un massimo.
6  Prima di utilizzare questa funzione è opportuno eseguire
7  nella funzione chiamante la seguente riga di codice
8  srand(time(NULL));
9  */
10 int casuale(int min, int max)
11 {
12     int casuale;
13     if(min>max)
14         scambia(&min, &max);
15     casuale=min+rand()%(max-min+1);
16     return casuale;
17 }
18 /*
19 Funzione che scambia il valore di due variabili
20 */
21 void scambia(int *a, int *b)
22 {
23     int c;
24     c=*a;
25     *a=*b;
26     *b=c;
27 }
28
```

File libreria.h (contiene i prototipi delle funzioni)

```
1  #ifndef LIBRERIA_H_INCLUDED
2  #define LIBRERIA_H_INCLUDED
3
4  int casuale(int min, int max);
5
6  void scambia(int *a, int *b);
7
8  #endif // LIBRERIA_H_INCLUDED
9
```

Spiegazione di queste direttive.

Poiché questo header file potrebbe essere incluso in più di un file .c, queste direttive **#ifndef** generano una compilazione condizionata dei prototipi in modo che i prototipi vengano compilati una sola volta. Questo evita multiple inclusioni del file .h che genererebbero errori di compilazione

Esempio:

Se questo file libreria.h non è mai stato compilato, la costante LIBRERIA_H_INCLUDED non è mai stata definita, pertanto viene definita la costante LIBRERIA_H_INCLUDED e vengono compilati i prototipi. A questo punto se ci fosse una ulteriore inclusione del file libreria.h all'interno di un qualsiasi altro file, il compilatore rileverebbe che LIBRERIA_H_INCLUDED è già stata definita quindi eviterebbe di compilare di nuovo i prototipi, azione che, se non evitata, provocherebbe conflitto e quindi errori di compilazione.

File main.c (file in cui vengono utilizzate le funzioni. Include il file header libreria.h)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "libreria.h"
4
5  int main()
6  {
7      int a=10;
8      int b=20;
9
10     scambia(&a, &b);
11
12     printf("\na=%i", a);
13     printf("\nb=%i", b);
14
15     srand(time(NULL));
16     for(int i=0; i<100; i++)
17         printf("\ncasuale=%i", casuale(1, 10));
18     return 0;
19 }
20
```

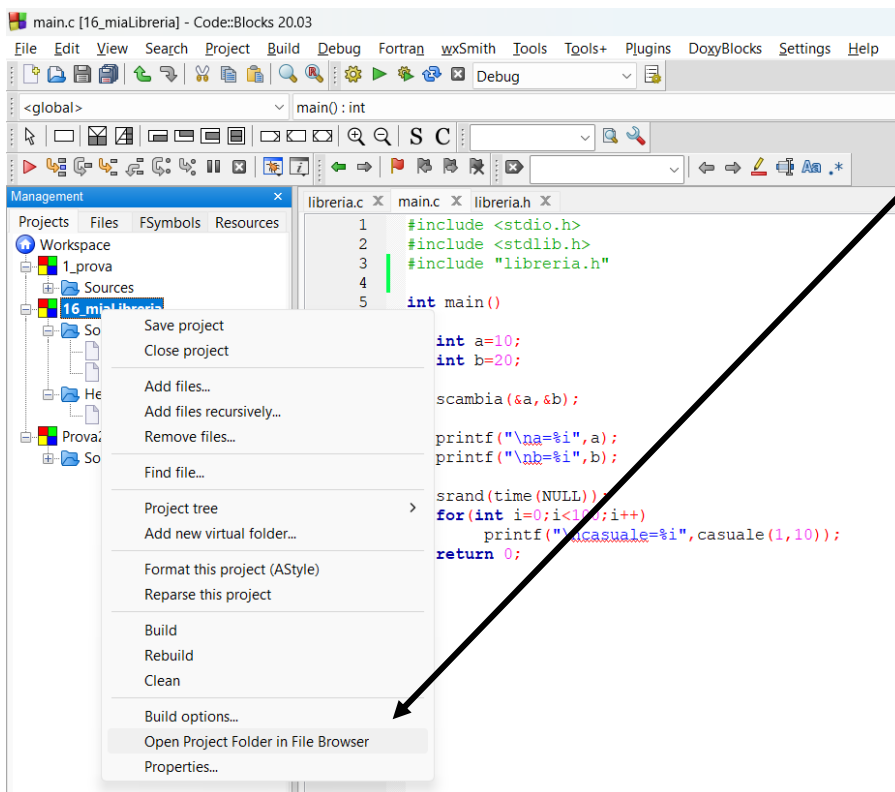
E' necessario includere il header file combinatorie.h

Come AGGIUNGERE dei file .c e file .h esistenti ad un nuovo progetto in CodeBlocks

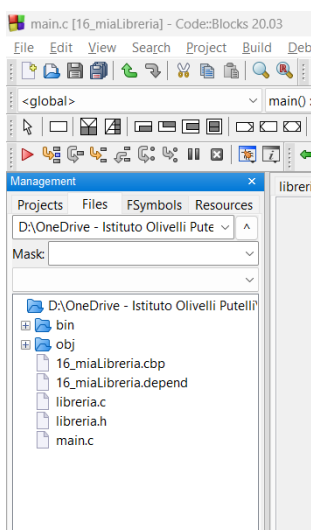
Vogliamo aggiungere i file libreria.c e libreria.h ad un nuovo progetto “prova2”.

Come prima cosa dobbiamo copiare i file libreria.c e libreria.h nella cartella del nuovo progetto “prova2”, per farlo dall’IDE si procede nel seguente modo:

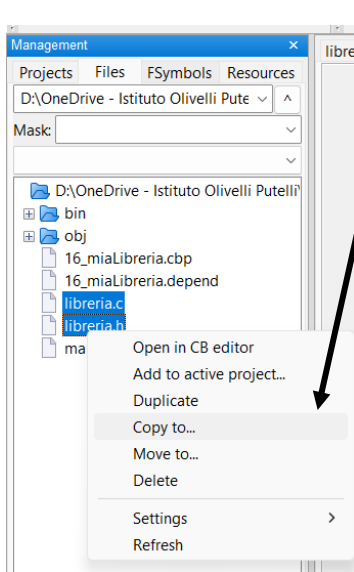
1. Apriamo il progetto “16_miaLibreria” che contiene i file della libreria
2. Tasto destro sul progetto “16_miaLibreria” e selezioniamo “Open Project Folder in File Browser”



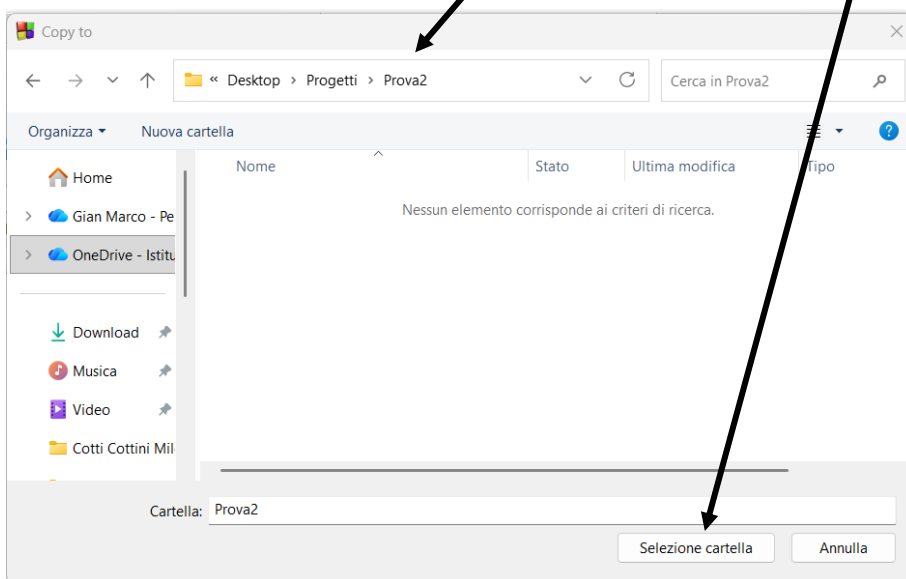
3. Si aprirà la scheda Files della finestra Manager mostrando i file del progetto “16_miaLibreria”.



4. Selezioniamo i due file da copiare “libreria.h” e “libreria.c” e clicchiamo con il tasto destro scegliendo “copy to”

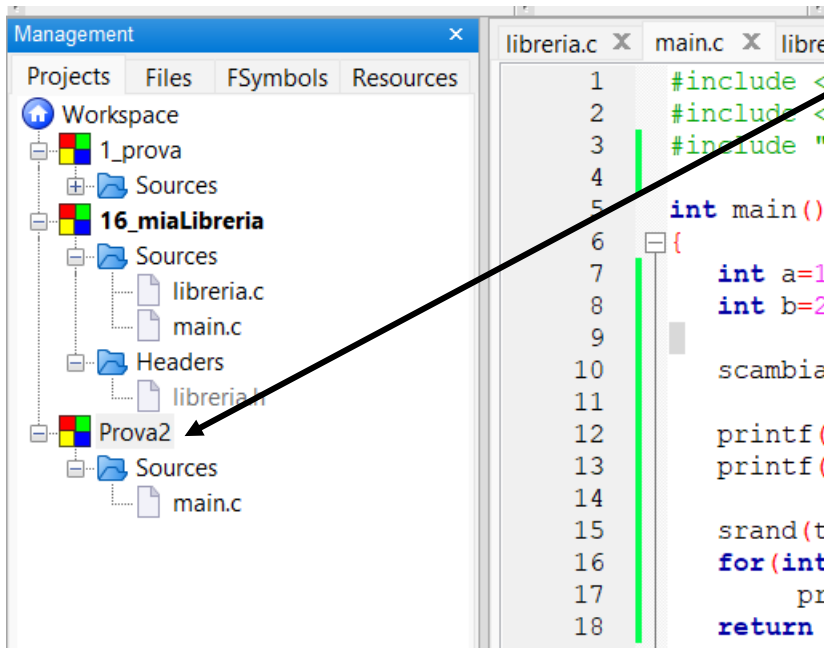


5. Si aprirà la finestra del gestore dei file di Windows. Selezioniamo la cartella di progetto in cui vogliamo copiare i file (la cartella prova2). Clicchiamo su “seleziona cartella” per copiare i file nel nuovo progetto.

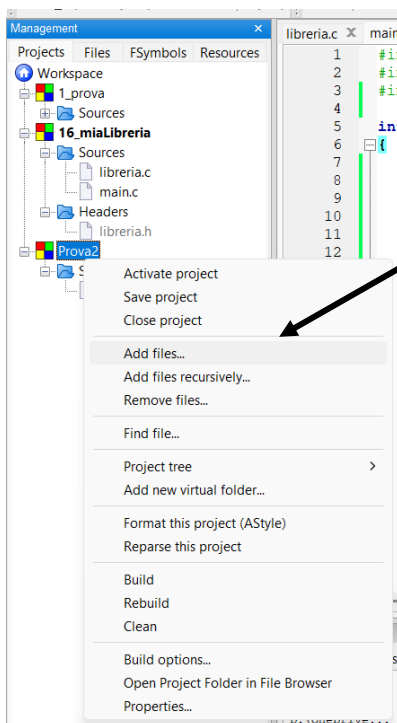


Non basta copiare i file “libreria.c” e libreria.h” nella cartella di progetto. Ora dobbiamo “aggiungere” i file al progetto “prova2”:

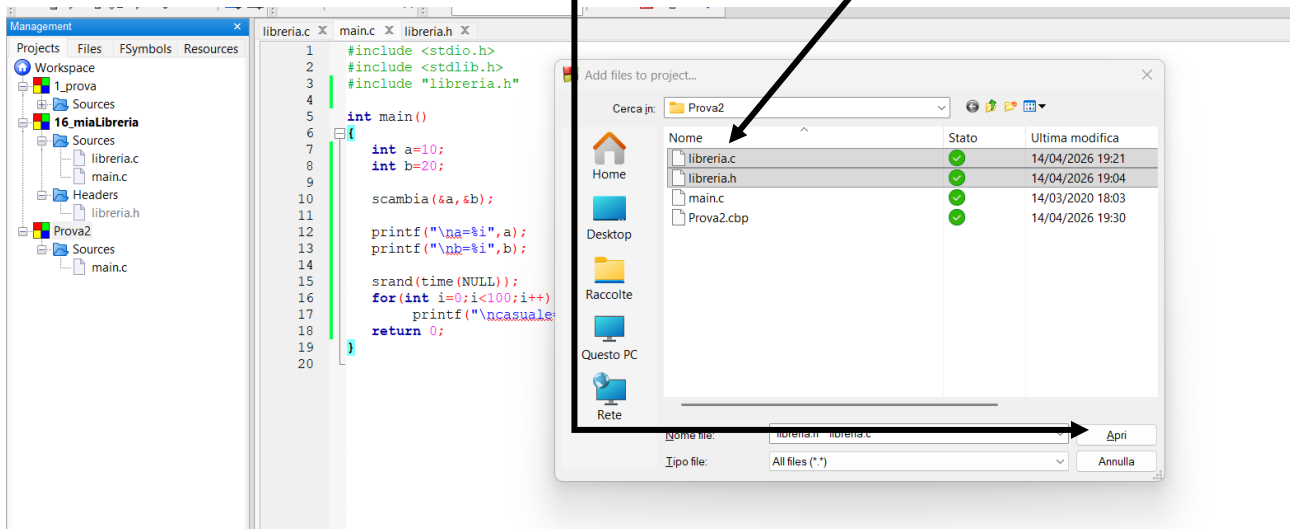
6. Clicchiamo sulla scheda Project della finestra Management per visualizzare il progetto “prova2”



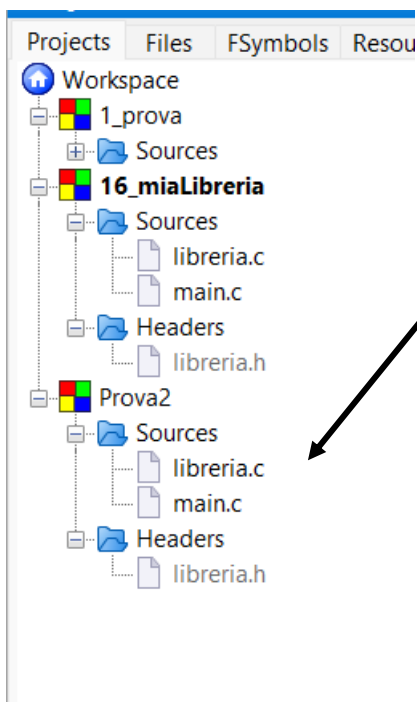
7. Clicchiamo con il tasto destro sul progetto “prova2” e selezioniamo “Add files..”



8. Si aprirà la cartella del progetto “prova2” che conterrà i due nuovi file precedentemente copiati. Selezioniamo i due file e clicchiamo su “Apri”



9. Confermiamo di voler aggiungere i file sia alla versione debug che quella release. Ora i file della libreria “16_miaLibreria” sono stati aggiunti al progetto. Salviamo le modifiche al progetto. Chiudiamo e riapriamo il progetto “prova2” per vederli apparire nella scheda Projects della finestra Management



10. Ora possiamo utilizzare le funzioni della libreria "16_miaLibreria" nel progetto "prova2". Per poterlo fare dobbiamo includere il file "libreria.h" nel file main.c del progetto "prova2":

