Laboratorio di Programmazione Gr. 3 (N-Z)

Corso di Laurea in Informatica

Università degli Studi di Napoli Federico II

A.A. 2022/23

A. Apicella

Le Funzioni

Si desidera implementare in C un programma *CalcolatriceCarina* che, dati gli operandi, visualizzi il risultato delle 4 operazioni base (+,-,/,x) circondato da una cornice di asterischi.

```
dammi in primo operando
4
dammi il secondo operando
2
la somma di 4.00 e 2.00 e'
**********************
**         6.00    **
**********************
la differenza tra 4.00 e 2.00 e'
**********************
**         2.00    **
**********************
il prodotto tra 4.00 e 2.00 e'
**********************
**          8.00   **
**********************
il quoziente tra 4.00 e 2.00 e'
**********************
**          2.00   **
**********************
```

In [44]:
```c
#include <stdio.h>
int main()
{
    float op1;
    float op2;
    float ris;
    printf("dammi in primo operando\n");
    scanf("%f", &op1);
    printf("dammi il secondo operando\n");
    scanf("%f", &op2);
    printf("la somma di %.2f e %.2f e'\n", op1, op2);
    ris = op1 + op2;
    printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",  10,ris , 10,"**");
    printf("**********************\n");
    printf("la differenza tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 - op2;
    printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",  10,ris , 10,"**");
    printf("**********************\n");
    printf("il prodotto tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 * op2;
    printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",  10,ris , 10,"**");
    printf("**********************\n");
    printf("il quoziente tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 / op2;
    printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",  10,ris , 10,"**");
    printf("**********************\n");
    return 0;
}
```



```
dammi in primo operando

dammi il secondo operando

la somma di 2.00 e 2.00 e'
**********************
**      4.00        **
**********************
la differenza tra 2.00 e 2.00 e'
**********************
**      0.00        **
**********************
il prodotto tra 2.00 e 2.00 e'
**********************
**      4.00        **
**********************
il quoziente tra 2.00 e 2.00 e'
**********************
**      1.00        **
**********************
```

Lo stesso codice ripetuto più volte!

```c
#include <stdio.h>
int main()
{
    float op1;
    float op2;
    float ris;
    printf("dammi in primo operando\n");
    scanf("%f", &op1);
    printf("dammi il secondo operando\n");
    scanf("%f", &op2);
    printf("la somma di %.2f e %.2f e'\n", op1, op2);
    ris = op1 + op2;
```

```c
printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",  10,ris , 10,"**");
    printf("**********************\n");
```

```c
printf("la differenza tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 - op2;
```

```c
printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",  10,ris , 10,"**");
    printf("**********************\n");
```

```
printf("il prodotto tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 * op2;
```

```
printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",  10,ris , 10,"**");
    printf("**********************\n");
```

```
printf("il quoziente tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 / op2;
```

```
printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",  10,ris , 10,"**");
    printf("**********************\n");
```

```
return 0;
}
```

- Definisco una funzione col codice ripetuto

```
#include <stdio.h>

void incornicia(float numero)
{
    printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",  10,numero , 10,"**");
    printf("**********************\n");

}
```

- Invoco la funzione

```
int main()
{
    float op1;
    float op2;
    float ris;
    printf("dammi in primo operando\n");
    scanf("%f", &op1);
    printf("dammi il secondo operando\n");
    scanf("%f", &op2);
    printf("la somma di %.2f e %.2f e'\n", op1, op2);
    ris = op1 + op2;
    incornicia(ris);
    printf("la differenza tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 - op2;
    incornicia(ris);
    printf("il prodotto tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 * op2;
    incornicia(ris);
    printf("il quoziente tra %.2f e %.2f e'\n", op1, op2);
    ris = op1 / op2;
    incornicia(ris);
  }
```

L'output effettivo è identico al precedente.

I parametri: perchè?

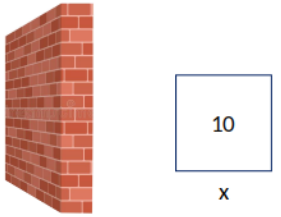
Le variabili dichiarate all'interno di un blocco **non sono visibili** (ossia non sono accessibili) all'esterno del blocco in cui sono dichiarate

Definizione

```
void incornicia( )
{
    printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",  10, x , 10,"**");
    printf("**********************\n");
}
```

Invocazione

```
int main()
{
  float x = 10;
  Incornicia( );
}
```



Errore di compilazione! La variabile **x** non è definita nello scope della funzione *incornicia*

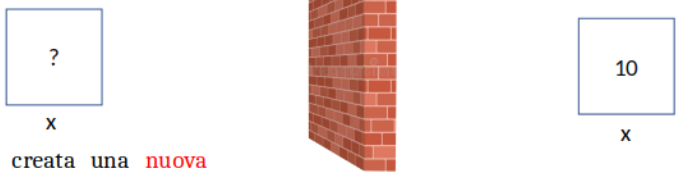
Potrei pensare di dichiarare una nuova variabile in incornicia, MA...

Definizione

```
void incornicia( )
{
    float x;
    printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",10,x, 10,"**");
    printf("**********************\n");
}
```

Invocazione

```
int main()
{
  float x = 10;
  Incornicia( );
}
```



Viene creata una nuova variabile, con lo stesso nome ma valore diverso

Il programma verrebbe eseguito, ma l'output non sarebbe quello desiderato

NECESSARIO UN SISTEMA DI COMUNICAZIONE TRA BLOCCHI ⇒ utilizzo dei parametri!

I parametri: come?

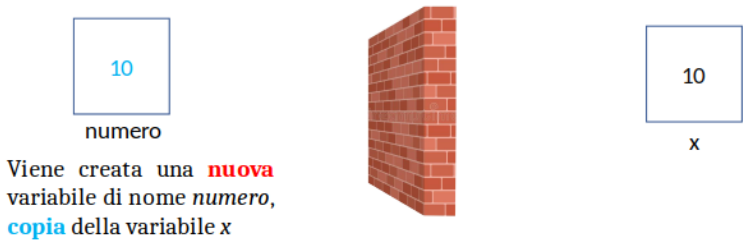
Definizione

```
void incornicia(float numero)
{
    printf("**********************\n");
    printf("%s%*.2f%*s\n", "**",10,numero, 10,"**");
    printf("**********************\n");
```

Invocazione

```
int main()
{
  float x = 10;
  Incornicia( x );
}
```

```
}
```



> **parametri formali**: parametri dichiarati come parte *dichiarazione/definizione* della funzione

> **parametri attuali**: parametri passati durante l'*invocazione* della funzione

Nello standard C è più corretto parlare di *parametri* e *argomenti*, rispettivamente al posto di *parametri formali* e *parametri attuali*.

Ritorno al chiamante

Il meccanismo del passaggio di parametri permette di copiare valori dall' *invocante* all'*invocato*, ma non viceversa.

Problema: implementare un programma che, data base ed altezza, stampi a video l'area ed il perimetro di un rettangolo

```c
#include <stdio.h>
int main()
{
    float b, h;
    float area, perimetro;
    printf("inserire il valore della base\n");
    scanf("%f", &b);
    printf("inserire il valore dell'altezza\n");
    scanf("%f", &h);
    area = b * h;
    perimetro = 2*b + 2*h;
    printf("l'area è %.2f \n", area);
    printf("il perimetro è %.2f \n", perimetro);
}
```

Una prima soluzione **sbagliata**

Definizione

```c
void calcola_area(float b, float h, float area)
{
   area = b * h;
}

void calcola_perimetro(float b, float h, float perimetro)
{
   perimetro = 2*b + 2*h;
}
```

Invocazione

```c
int main()
{
    float b, h;
    float area, perimetro;
    printf("inserire il valore della base\n");
    scanf("%f", &b);
    printf("inserire il valore dell'altezza\n");
    scanf("%f", &h);
    calcola_area(b, h, area);
    calcola_perimetro(b, h, perimetro);
    printf("l'area è %.2f \n", perimetro);
    printf("il perimetro è %.2f \n", perimetro);
}
```


La tecnica del passaggio di parametri, in C, è definita **solo per copia**. Tale copia viene eseguita soltanto all'atto dell'invocazione della funzione dal chiamante al chiamato, e non alla fine dal chiamato al chiamante!

Definizione

```c
float calcola_area(float b, float h)
{
   float a;
   a = b * h;
   return a;
}

float calcola_perimetro(float b, float h)
{
    float p;
    p = 2*b + 2*h;
    return p;
}
```

Invocazione

```c
int main()
{
    float b, h;
    float area, perimetro;
    printf("inserire il valore della base\n");
    scanf("%f", &b);
    printf("inserire il valore dell'altezza\n");
    scanf("%f", &h);
    area = calcola_area(b, h);
    perimetro = calcola_perimetro(b, h);
    printf("l'area è %.2f \n", area);
    printf("il perimetro è %.2f \n", perimetro);
}
```

Il valore che segue la parola chiave `return` viene copiato all'interno della variabile a sinistra dell'assegnazione nell'invocante (o in un'eventuale espressione). Se una funzione restituisce un valore attraverso la parola chiave `return`, il tipo di ritorno deve essere dello stesso tipo del valore restituito (e non più `void`)

In generale:

Una funzione in *C* viene **dichiarata/definita** nel seguente modo:

```c
tipo_ritorno nome_fun(tipo_param1 nome_param1,
                      tipo_param2 nome_param2, ...)
    {
        ...corpo...
        return valore_ritorno;
    }
```

- tutto ciò che viene prima della `{` è detto **header**
- la lista dei parametri (formali) è composta da **dichiarazioni di variabili** (quindi compresa di tipo) separati da `,`
- il tipo di ritorno deve corrispondere al tipo del valore di ritorno. In caso contrario, il valore di ritorno sarà convertito nel tipo di ritorno.
- nel caso in cui non sia necessario che la funzione restituisca un valore, *tipo_ritorno* può essere `void` e la parola chiave `return` può essere omessa (o lasciata senza valore, ossia `return;`)
- I valori che assumono i parametri in fase di invocazione sono detti **parametri attuali**

- una funzione può anche solo essere *__dichiarata__* senza essere (ancora) definita esplicitando l'header seguito da un `;`. Esempio:

```
tipo_ritorno nome_fun(tipo_param1 nome_param1,
                      tipo_param2 nome_param2, ...);
```

La *dichiarazione* senza definizione di una funzione serve a segnalare che esiste(rà) una funzione di nome `nome_fun` avente parametri di tipo `tipo_param1, tipo_param2,...`, la cui *__definizione__* è specificata altrove.

In [4]:
```
#include <stdio.h>
int f()
{
    double a = 3.5;
    return a;
}

int main()
{
    double r = f(); // invocazione
    printf("%f\n",r);

    return 0;
}
```

```
3.000000
```

In [7]:
```
#include <stdio.h>
void stampa_doppio(int n)
{
    printf("%d",n*2);
}

int main()
{
    stampa_doppio(4); //invocazione

    return 0;
}
```

```
8
```

L'invocazione di una funzione che restituisce un valore presuppone che tale invocazione sia fatta all'interno di una espressione o un'assegnazione.

Esempio:

```
double quadrato(double l)
    {
        return l*l;
    }

    int main()
    {
        int q = quadrato(10);
        printf("il quadrato di 10 è %d\n",q);

        printf("il quadrato di 3 è %d\n", quadrato(3));

        int l = q + quadrato(3);

        printf("la loro somma è %d\n", l);

        return 0;
    }
```


da notare:

- non salvare `quadrato(3)` in una variabile non è convenuto, in quanto l'ho dovuto ricalcolare con una seconda invocazione
- l'object `int l` dichiarato nel corpo del `main()` non ha nulla a che vedere con il `double l` dichiarato come parametro della funzione `quadrato(double l)`. Gli object dichiarati nel corpo di una funzione sono visibili solo all'interno di quest'ultima → Gli *scope* sono separati (anche per questo sono necessari i parametri)
- `return` esce dalla funzione in esecuzione (anche se quest'ultima non è terminata, quindi attenzione a dove viene messa!)

In [10]:
```
# include <stdio.h>
//Esempio:
double calcola_media(double a, double b)
{
    double media = a + b;
    return media;
    media = media / 2.0;
}

int main()
{
    printf("la media tra 2 e 3 è %f\n", calcola_media(2,3));
    return 0;
}
```

```
la media tra 2 e 3 è 5.000000
```

In [11]:
```
# include <stdio.h>
double calcola_valore_assoluto(double n)
{
    if( n >= 0 )
    {
        return n;
    }
    else
    {
        return -n;
    }
}

int main()
{
    printf("il valore assoluto di -4 è %f\n",
                calcola_valore_assoluto(-4));
    return 0;
}
```

```
il valore assoluto di -4 è 4.000000
```

La **funzione** main()

il cosiddetto `main()` è una funzione a tutti gli effetti.

`main()` è la funzione che viene invocata all'atto dell'esecuzione del programma (*entry point*). Sarà quindi sua cura invocare al suo interno le funzioni che devono essere a loro volta utilizzate nel programma (se necessario). Niente vieta di dichiarare/definire funzioni senza mai invocarle.

In quanto funzione:

- viene dichiarata con tipo di ritorno (`int` , in versioni di C precedenti era accettato anche `void`)
- restituisce un valore (per questo `return 0;`) al termine della sua esecuzione, che corrisponde al termine del programma.
- può anche avere parametri, ma lo vedremo più in là

Perchè proprio 0?

Exit code

Solitamente il valore di ritorno della funzione `main()` (o anche *exit code*) è un intero tra 0 e 255 che può servire al sistema operativo ad innescare determinate operazioni una volta terminato il programms. Se il programma termina correttamente ci si aspetta che tale valore sia 0, se invece ci sono stati errori il valore di ritorno potrebbe essere $\neq 0$.

Se si desidera visualizzare l'exit code dell'ultimo programma mandato in esecuzione, dare il comando linux `echo $?`

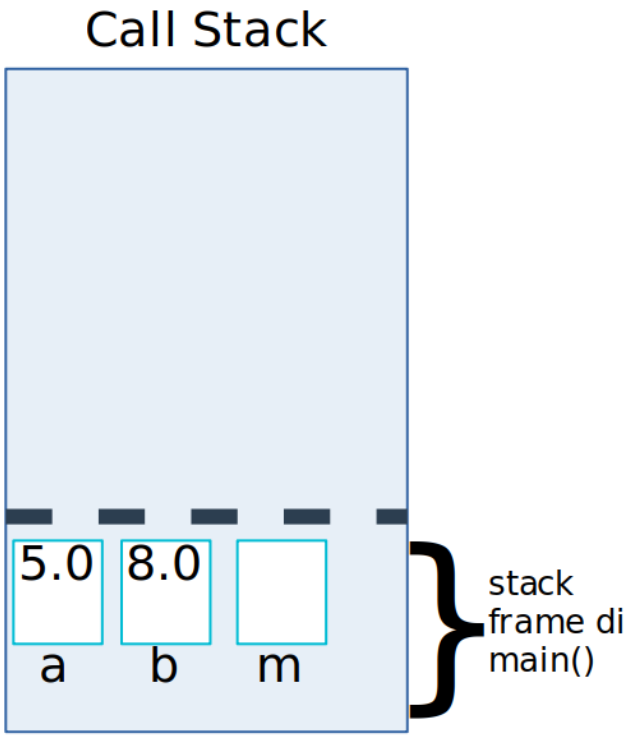
In [13]:
```
# include <stdio.h>
int main()
{
    return 34;
}
```

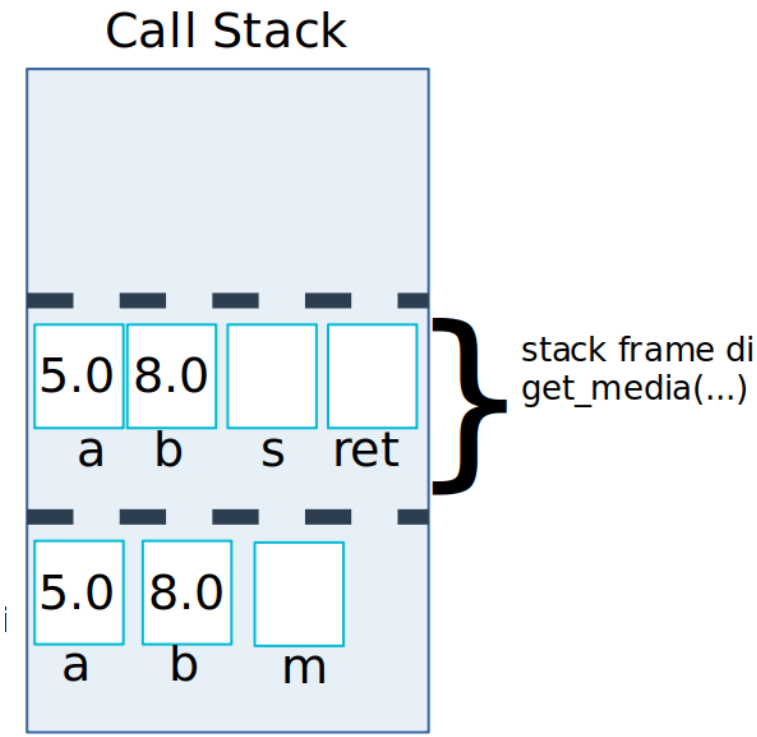
```
[C kernel] Executable exited with code 34
```

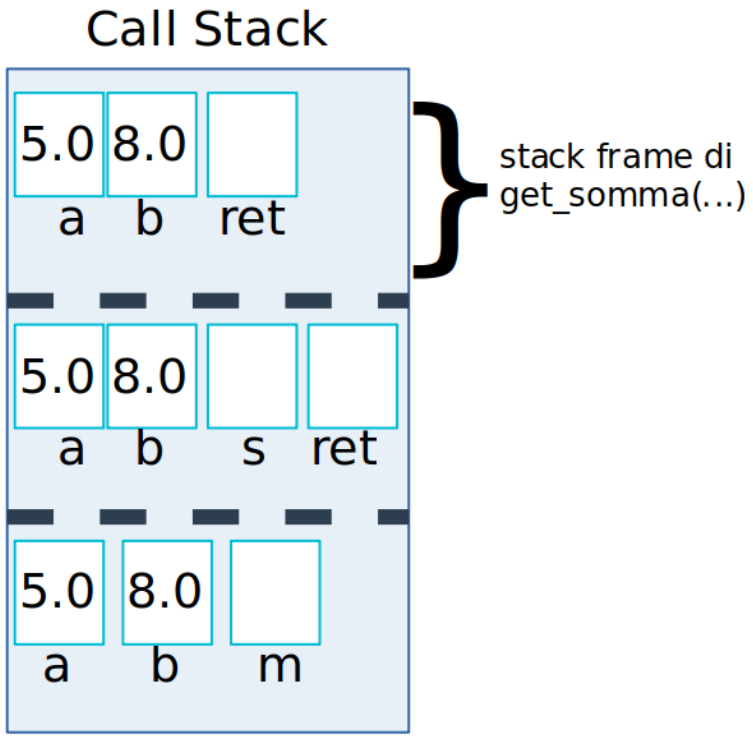
Il call stack

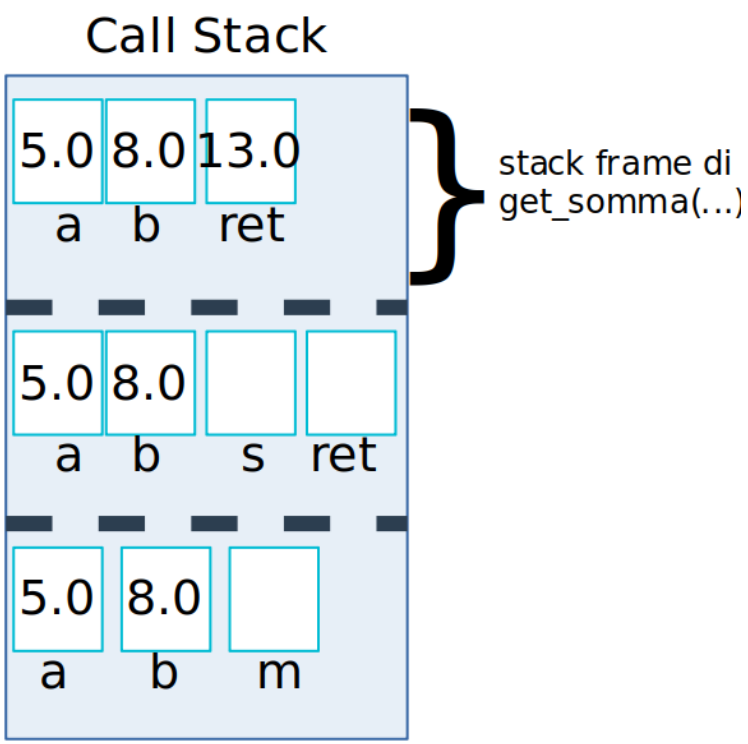
In [4]:
```
# include <stdio.h>

double get_somma(double a, double b)
{
    double ret = a + b;
    return ret;
}

double get_media(double a, double b)
{
    double s;
    s          = get_somma(a, b);
    double ret;
    ret        = s/2.0;
    return ret;
}

int main()
{
    double a = 5;
    double b = 8;
    double m;
    m        = get_media(a,b);
    printf("la media tra %.2lf e %.2lf è %.2lf\n",a,b,m);

    return 0;
}
```

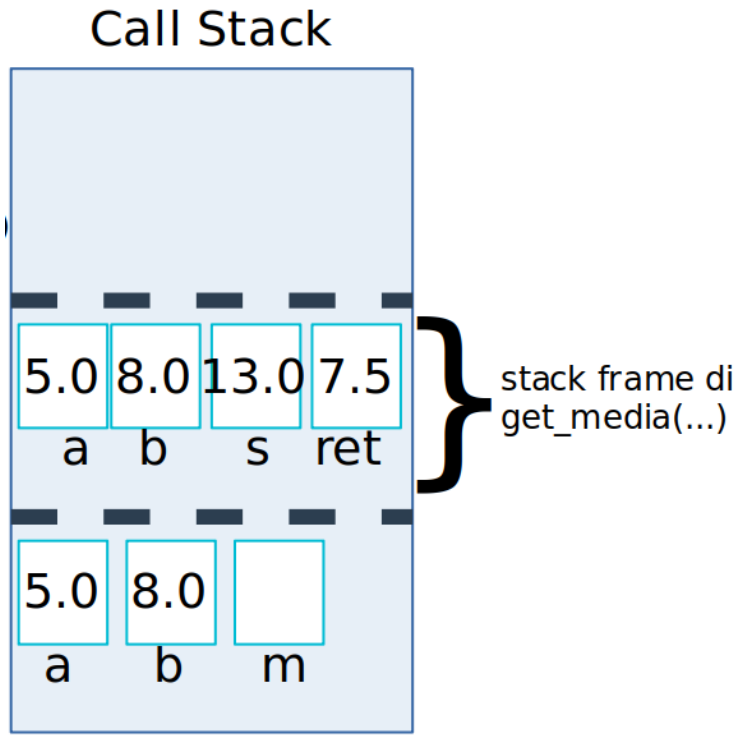
```
la media tra 5.00 e 8.00 è 6.50
```

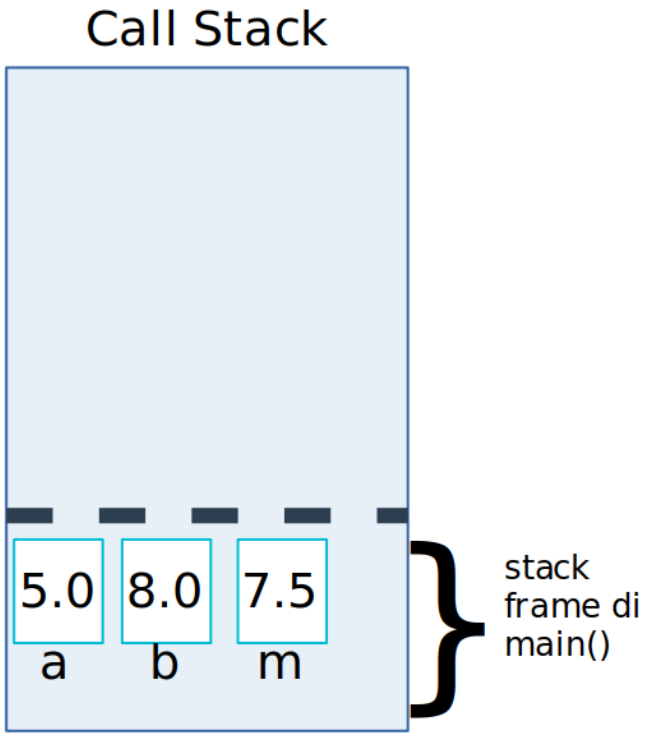




Call Stack
5.0
8.0
a
b
s
ret
stack frame di
get_media(...)
5.0
8.0
a
b
m


Call Stack
5.0
8.0
a
b
ret
stack frame di
get_somma(...)
5.0
8.0
a
b
s
ret
5.0
8.0
a
b
m


Call Stack
5.0
8.0
13.0
a
b
ret
stack frame di
get_somma(...)
5.0
8.0
a
b
s
ret
5.0
8.0
a
b
m


Call Stack
5.0
8.0
13.0
7.5
a
b
s
ret
stack frame di
get_media(...)
5.0
8.0
a
b
m



Ogni volta che viene invocata una funzione, viene impegnato un frammento (Stack Frame) del *call stack*. Tale frammento conterrà:

- le variabili locali della funzione
- eventuali parametri attuali
- dati (e.g.,l'indirizzo di ritorno) per ripristinare lo stato al programma al termine della funzione

Ogni stack frame verrà eliminato dal call stack al termine della relativa funzione.

Parametri di uscita: come?

Il problema dello scambio

> Definire una funzione che, date due variabili, ne scambi il contenuto. Stampare quindi, nel chiamante, le variabili scambiate

Versione senza funzioni:

```c
#include <stdio.h>
int main()
{
    int a;
    int b;
    int t;  // necessaria per lo scambio
    printf("dammi il valore della variabile a\n");
    scanf("%d", &a);
    printf("dammi il valore della variabile b\n");
    scanf("%d", &b);
    t = a;
    a = b;
    b = t;
    printf("dopo lo scambio a vale %d e b vale %d\n", a, b);
}
```


Una soluzione **sbagliata**

Definizione

```c
void scambia(int a, int b)
{
    int a;
    int b;
    int t;  // necessaria per lo scambio
    t = a;
    a = b;
    b = t;
}
```

Invocazione

```c
int main()
{
    int a;
    int b;
    printf("dammi il valore della variabile a\n");
    scanf("%d", &a);
    printf("dammi il valore della variabile b\n");
    scanf("%d", &b);
    scambia(a,b);
    printf("dopo lo scambio a vale %d e b vale %d\n", a, b);
}
```

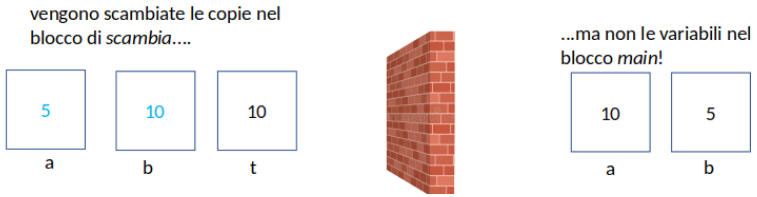
Il programma, compilato ed eseguito, **non** scambierà il contenuto delle variabili.

Definizione

```c
void scambia(int a, int b)
{
    int a;
    int b;
    int t;  // necessaria per lo scambio
    t = a;
    a = b;
    b = t;
}
```

Invocazione

```c
int main()
{
    int a;
    int b;
    printf("dammi il valore della variabile a\n");
    scanf("%d", &a);
    printf("dammi il valore della variabile b\n");
    scanf("%d", &b);
    scambia(a,b);
    printf("dopo lo scambio a vale %d e b vale %d\n", a, b);
}
```



Le variabili `a,b,t` sono variabili ***locali*** del metodo `scambia(...)`, mentre le variabili `a,b` sono variabili ***locali*** del metodo `main()`

Definizione

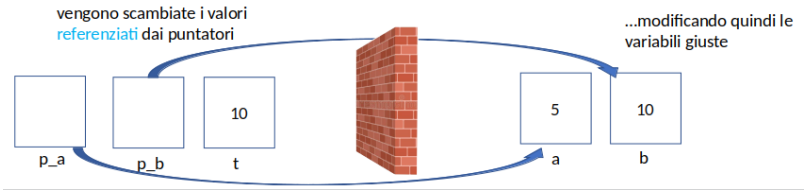
```c
void scambia(int* p_a, int* p_b)
{
    int t;  // necessaria per lo scambio
    t = *p_a;
```

Invocazione

```c
#include <stdio.h>
int main()
{
    int a;
    int b;
```

```
    *p_a = *p_b;
    *p_b = t;
}
```

```
    printf("dammi il valore della variabile a\n");
    scanf("%d", &a);
    printf("dammi il valore della variabile b\n");
    scanf("%d", &b);
    scambia(&a, &b);
    printf("dopo lo scambio a vale %d e b vale %d\n", a, b);
}
```



> **Domanda:** potevo risolvere lo stesso problema utilizzando la parola chiave `return`?

> **Risposta: no**, in quanto con la parola chiave return, in *C*, posso restituire **al più un valore**. In alternativa, avrei potuto restituire il valore di una variabile con `return` (es. `return a;`), e modificare l'altra (es. `b`) con un puntatore

> **Domanda:** quindi posso utilizzare la tecnica del passaggio di parametri per puntatori e del return nella stessa funzione?

> **Risposta: sì**, sono due cose totalmente diverse, quindi possono essere utilizzate tranquillamente assieme.

Passaggio di array allocati tramite operatore `[]` ad una funzione

Il passaggio di parametri di array è da considerarsi sempre *per riferimento*

```
void f(int V[10], int n)
{
    for(int i=0; i < n; i++)
    {
        scanf("%d", &V[i]);
    }
}

int main()
{
    int n;
    int vett[10];
    printf("quanti elementi vuoi inserire?");
    scanf("%d", &n);

    f(vett,n);

    printf("i valori inseriti sono:\n");
    for(int i=0; i < n; i++)
    {
        printf("%d ", vett[i]);
    }
    printf("\n");
```



```
    return 0;
}
```

Questo perchè il nome simbolico del vettore `vett` si riferisce sempre all' *indirizzo* del vettore, quindi "passare" un vettore come parametro significa, di fatto, passare il suo indirizzo.

Analogamente, la dichiarazione del parametro `int V[10]` è simile alla dichiarazione di un puntatore. Volendo, la stessa funzione poteva essere definita come:

```
void f(int V[], int n)
{
    for(int i=0; i < n; i++)
    {
        scanf("%d", &V[i]);
    }
}
```

La dimensione, nel caso di array monodimensionali come parametro, non è necessaria. Dato che, all'atto pratico, ciò che viene passato è l'indirizzo ad un `int` (nello specifico, l'indirizzo del primo elemento dell'array), tale funzione poteva anche essere scritta come:

```
void f(int* V, int n)
{
    for(int i=0; i < n; i++)
    {
        scanf("%d", &V[i]);
    }
}
```

Discorso simile, **ma non identico**, con array multidimensionali:

```
void f(int M[10][3], int n_r, int n_c)
{
    for(int i=0; i < n_r; i++)
    {
        for(int j=0; j < n_c; j++)

        {scanf("%d", &M[i][j]);}
    }
}

int main()
{
    int n_r, n_c;
    int mat[10][3];
    printf("quante righe vuoi inserire?");
    scanf("%d", &n_r);
    printf("quante colonne vuoi inserire?");
    scanf("%d", &n_c);
    f(vett,n);

    printf("i valori inseriti sono:\n");
    for(int i=0; i < n_r; i++)
    {
        for(int j=0; j < n_c; j++)
            printf("%d ", mat[i][j]);
    }
    printf("\n");
    return 0;
}
```

MA... in questo caso la dichiarazione del parametro `int M[10][3]` come `int M[][]` non va bene!

Perchè?

semplifichiamo il problema analizzando la seguente funzione (sbagliata):

```
void f(int M[][])
{
    M[1][2] = 42; // dove finisce la riga 0
                  // e dove inizia la colonna 1??
}
```

ricordiamo che una matrice in memoria è rappresentata in maniera sequenziale in modalità row-major (in C). Senza conoscere il numero di colonne, come può la funzione determinare dove finisce la riga con indice `0` ed inizia la riga con indice `1` ? Fornire alla funzione almeno il numero di colonne è indispensabile. Quindi una versione corretta della funzione è la seguente:

```
void f(int M[][3])
{
    M[1][2] = 42;
}
```

Questa funzione è quindi valida per qualsiasi matrice avente 3 colonne, indipendentemente dal numero di righe che tale matrice abbia.

Come specificato quando si è parlato di puntatori, però, i nomi delle matrici (o più in generale gli array multidimensionali) allocate tramite l'operatore `[]` **non** possono essere visti come *pointers to pointers* (i.e. `**`), quindi una dichiarazione di parametro *formale* fatta come nella seguente funzione:

```
void f(int** M)
{
   ...
}
```

non sarebbe compatibile con un parametro *attuale* di tipo matrice allocata tramite operatore `[]`.

Passare un array ad una funzione per copia

Se proprio indispensabile, c'è un trick per passare un vettore ad una funzione per copia. La tecnica consiste nell'incapsularlo in una struttura:

```
struct Involucro
{
    int vett[10];
}

void f(struct Involucro inv, int n)
{
    for(int i=0; i < n; i++)
    {
        scanf("%d", &(inv.vett[i]));
    }
}

int main()
{
    int n;
    struct Involucro t;
    printf("quanti elementi vuoi inserire?");
    scanf("%d", &n);

    f(t,n);
```



```
    printf("i valori inseriti sono:\n");
    for(int i=0; i < n; i++)
    {
        printf("%d ", t.vett[i]);
    }
    printf("\n");
    return 0;
}
```

Una struttura, in C, di default viene passata per copia, e quindi con essa viene **copiato** tutto il suo contenuto, compresi eventuali array definiti attraverso `[]`.

Questa funzione, quindi, farà ciò che ci aspettiamo che faccia??

Ovviamente la risposta è **no**, in quanto la struct `t` viene passata *per copia*.

In [ ]:

In [ ]: