

# FONDAMENTI DI PROGRAMMAZIONE

## C / C++

### Docente: Armando Valentino

In collaborazione con:



per una crescita intelligente,  
sostenibile ed inclusiva

[www.regione.piemonte.it/europa2020](http://www.regione.piemonte.it/europa2020)

INIZIATIVA CO-FINANZIATA CON FSE



# LINGUAGGIO C/C++

## PROCEDURE E FUNZIONI

```
double rand_CLT (double xMin, double xMax,  
                 int tries)  
{  
    double x = 0. ;  
  
    for (int i = 0 ; i < tries ; ++i)  
    {  
        x += rand_range (xMin, xMax) ;  
    }  
    return x / tries ;  
}
```

*Prof. Armando Valentino*

# Sottoprogrammi

- Altra giustificazione all'esistenza dei sottoprogrammi deriva dal fatto che un programma può consistere di decine di migliaia di istruzioni.
- Sebbene fattibile, una soluzione "monolitica" del problema è inefficiente in quanto complicata da "*debuggare*" e difficile da leggere.
- Per questo si tende a organizzare il programma complessivo in "*moduli*" ognuno dei quali tratta e risolve solo una parte del problema.
- In altre parole si adotta, per la soluzione del problema, il cosiddetto "approccio *top-down*"

# Approccio top-down

- Il problema è decomposto in una sequenza di sottoproblemi più semplici.
- Quest'azione è ripetibile su più livelli, ovvero ogni sottoproblema può a sua volta essere decomposto in una sequenza di sottoproblemi.
- La decomposizione prosegue sino a che si ritiene che la sequenza è composta ormai solamente da “*sottoproblemi terminali*”, ovvero da sottoproblemi risolvibili in modo “semplice”.

Sviluppare un programma in modalità top down significa suddividere il problema in sottoproblemi di complessità inferiore

L'uso di funzioni permette di strutturare il programma in modo modulare, sfruttando tutte le potenzialità fornite dalla programmazione strutturata fornite dal C++.

Una *funzione* è un blocco di istruzioni con un nome che viene eseguito in ogni punto del programma in cui viene richiamata la funzione usando il nome.

Si dichiara nel modo seguente....

```
Tiporestituito NomeFunzione ( Argomento1, Argomento2 , ...) {  
    istruzioni;  
    return valoreRestituito;  
}
```

dove:

- **Tipo restituito** è il tipo del valore ritornato dalla funzione che può essere uno dei tipi di dati.
- **NomeFunzione** è il nome con cui possiamo richiamare la funzione.
- **Argomento** Un **argomento** è costituito da un **nome di tipo** seguito da un identificatore (ad esempio **int x** ), esattamente come in una dichiarazione di variabile; Gli argomenti permettono di passare dei parametri quando la funzione viene richiamata. I parametri sono separati da virgole.
- **Istruzioni** è il corpo della funzione: un blocco di istruzioni racchiuse tra parentesi graffe {}

In C++ una procedura è un caso particolare di funzione, una funzione che non restituisce alcun valore.

Ha il seguente formato:

```
void nomeprocedura( void) {  
    istruzioni;  
}
```

**void** davanti al nome della procedura indica che non viene restituito nessun valore, **void** come parametro indica che non viene passato nessun valore alla procedura

# Esempio di funzione

```
// esempio dell' uso di una funzione
#include <iostream>
int z;                                     //variabile globale
int somma (int a, int b){                // parametri formali
    int r;                               //variabile locale
    r=a+b;
    return r;
}

int main () {
    /*chiamata alla funzione con il passaggio
    di parametri attuali */
    z = somma (5,3);
    cout << "Il risultato e' " << z;
    return 0;
}
```

Il risultato e' 8

# Parametri delle Funzioni

La lista dei parametri formali di una funzione e quella dei parametri attuali nella chiamata alla funzione devono rispettare tre regole:

- **Numero dei parametri** nella definizione e poi nella chiamata alla funzione deve essere uguale;
- **Il tipo** dei parametri formali deve essere uguale a quello degli attuali;
- **L'ordine dei parametri** deve essere lo stesso.



# Parametri delle Funzioni

un programma C++ inizia ad essere eseguito sempre dalla funzione **main**.

Nel main vi è una chiamata alla funzione **somma**.

Osserviamo la somiglianza della chiamata alla funzione con l'intestazione della dichiarazione della funzione:

```
int somma (int a, int b)
          ↑      ↑
z = somma ( 5 , 3 );
```

Vi è una chiara corrispondenza: nella funzione **main** abbiamo richiamato la funzione **somma** passando, come parametri, i due valori **5** e **3** che corrispondono agli argomenti **int a** ed **int b** nella dichiarazione della funzione **somma**.

**Questo passaggio di parametri si dice per valore**

Quando la funzione **somma** viene richiamata dal **main**, il controllo passa dalla funzione **main** alla funzione **somma**.

I valori **5** e **3** passati come **parametri attuali** vengono copiati nelle due variabili **int a** ed **int b** (detti **parametri formali** alla funzione **somma**) .

**L'istruzione:** *return r;*

L'istruzione **return** ha come argomento la variabile **r** ( **return r;** ), che al momento dell'esecuzione della **return** ha valore **8**; di conseguenza **8** è il valore ritornato dalla funzione nel main.

Tale valore (ossia **8**) che viene assegnato alla variabile **z**.

# Regole di Visibilità (SCOPE) delle variabili

Una **variabile** si dice **Globale** quando si può utilizzare in qualunque parte del programma; Si dice **locale** se può essere utilizzata solo all'interno di un modulo e dei suoi sottomoduli

Le **variabili locali** sono definite (ed il loro uso dichiarato) nella funzione (o nel blocco) che le usa; nascono quando la funzione entra in esecuzione e muoiono al termine dell'esecuzione della funzione.

Le **variabili locali** sono memorizzate nello **stack**, quelle **globali** nell'area dati del programma

Normalmente si usano i parametri per scambiare dati tra funzioni e main.

Una buona norma di programmazione richiede, quando è possibile, **di evitare l'uso delle variabili globali**

Questo permette una **più facile lettura e manutenzione** del programma, **ridurre l'occupazione di memoria** del programma, evitare side effect indesiderati (la modifica di una variabile globale in una funzione provoca un effetto collaterale o side effect)

# Regole di Visibilità (SCOPE) delle variabili

- **Globali dichiarate** in testa al programma, dopo la direttiva **#include**
- **Locali dichiarate** all'interno di funzioni sono visibili ed utilizzabili solo nell'ambiente in cui sono definite
- **Parametri formali** sono quelli che vengono descritti in fase di **definizione della funzione**
- **Parametri attuali** sono le variabili passate alle funzioni nel **momento della loro chiamata**

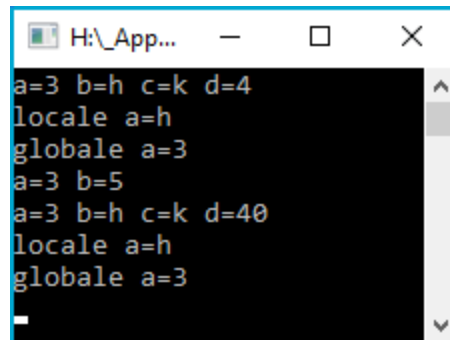
In una funzione si possono **dichiarare variabili locali con lo stesso nome di variabili globali**, in questo caso la **variabile locale nasconde quella globale (shadowing)**.

- Una **variabile** si dice **statica** se il suo periodo termina con l'esecuzione del programma, se termina prima è detta **dinamica**
- Le **variabili globali** (e le costanti globali ) sono sempre statiche.
- Le **variabili statiche** possono essere **dichiarate anche all'interno di funzioni** ( es: *static int nomevar*) e mantengono ad ogni accesso alla funzione il valore che avevano nel precedente accesso
- Per accedere ad una variabile globale oscurata da una locale si può utilizzare l'operatore **:: operatore di riferimento globale**

# Esempio variabili globali e locali

```
#include <iostream>
using namespace std;
int a, b;           // variabili globali
void varLocali (void) {
    static int d=4;  // variabile locale statica
    char b,c;        // variabili locali dinamiche
    b='h';
    c='k';
    cout << "a=" << a << " b=" << b << " c=" << c << " d=" << d << endl;
    d*=10;
    if(b < c) {
        char a;      // variabile locale dinamica del blocco if
        a=b;
        b=a;
        c=a;
        cout << "locale a=" << a << endl;
        cout << "globale a=" << ::a << endl; // variabile globale
    }
}
```

```
int main () {
    a=3;
    b=5;
    varLocali();
    cout << "a=" << a << " b=" << b << endl;
    varLocali();
    getchar();
    return 0;
}
```



```
H:\_App...
a=3 b=h c=k d=4
locale a=h
globale a=3
a=3 b=5
a=3 b=h c=k d=40
locale a=h
globale a=3
```

variabili a e b sono globali.  
b e c sono locali alla funzione  
a è locale al blocco if

Al secondo richiamo della funzione il valore di ingresso della variabile statica **d** è quello che aveva all'uscita della funzione nella precedente chiamata, perché la dichiarazione **static int d=4** non ha effetti perché la variabile già esiste, quindi la stampa è **d=40**.

L'operatore **::** richiama il valore della variabile globale all'interno della funzione

# Esempio di uso di funzioni

```
// esempio di funzioni  
#include <iostream>
```

```
int sottrazione (int a, int b)  
{  
    int r;  
    r=a-b;  
    return r;  
}
```

```
int main ()  
{  
    int x=5, y=3, z;  
    z = sottrazione (7,2); <--  
    cout << "\n Il primo risultato e' " << z;  
    cout << "\n Il secondo risultato e' " << sottrazione (7,2);  
    cout << "\n Il terzo risultato e' " << sottrazione (x,y);  
    z = 4 + sottrazione (x+2,y); <--  
    cout << "Il quarto risultato e' " << z << '\n';  
    return 0;  
}
```

Il primo risultato e' 5  
Il secondo risultato e' 5  
Il terzo risultato e' 2  
Il quarto risultato e' 8

diversi modi per  
chiamare la funzione  
per illustrare cosa  
succede quando una  
funzione viene  
chiamata

# Funzioni senza risultato. Uso di void

E se non vogliamo ritornare alcun valore?

Supponiamo di **voler scrivere una funzione che deve soltanto scrivere qualcosa sullo schermo.**

**Non ci serve che essa ritorni un valore** e neppure abbiamo bisogno di passargli dei **parametri**.

Allo scopo il C fornisce un particolare tipo **void**.

```
// esempio di funzione void
#include <iostream>

void stampa (void) {
    cout << "Sono una funzione!";
}

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

# Parametri per valore

## Parametri passati *per valore* e *per riferimento*.

Negli esempi di funzioni visti finora i parametri venivano passati *per valore*. Questo significa che quando viene chiamata una funzione quello che viene passato alla funzione è il valore dei parametri (siano essi delle costanti o delle variabili o delle espressioni). In particolare, se il parametro è una variabile viene passato alla funzione il valore della variabile ma non la variabile stessa. Supponiamo, ad esempio, di richiamare la funzione **somma** nel modo seguente:

```
int x=5, y=3, z;
```

```
z = somma ( x , y );
```

In questo caso viene richiamata la funzione **somma** passandogli i valori di **x** ed **y** , ossia **5** e **3** , ma non le variabili stesse:

```
int somma (int a, int b)
           ↑5   ↑3
z = somma ( x , y );
```

In questo modo, quando la funzione **somma** è chiamata, i valori delle sue variabili **a** e **b** sono **5** e **3** rispettivamente. Una modifica di **a** o **b** all'interno della funzione **somma** non cambia i valori delle variabili **x** ed **y** esterne ad essa. Questo perché non sono state passate le variabili **x** ed **y** alla funzione **somma** ma soltanto il loro valore .



# Parametri per Riferimento

```
// passaggio di parametri per riferimento
#include <iostream>
using namespace std;
void raddoppia (int &a, int &b, int &c)
{
    a*=2;
    b*=2;
    c*=2;
}

int main ()
{
    int x=1, y=3, z=7;
    raddoppia (x, y, z);
    cout << "x=" << x << ", y=" << y << ", z=" << z;
    return 0;
}
```

Ci sono però casi in cui **vogliamo modificare dall'interno di una funzione il valore di variabili** definite esternamente alla funzione stessa.

A questo scopo possiamo usare dei parametri **passati per riferimento**, come nella funzione **raddoppia** dell'esempio seguente:

**risultato**  
**x=2, y=6, z=14**

# Passaggio di parametri per riferimento

La prima cosa da notare è che nella dichiarazione di raddoppia il tipo di ciascun parametro è seguito dal carattere e commerciale (&); **&** indica un passaggio di parametro per riferimento

Quando passiamo una variabile per riferimento è la variabile stessa che noi passiamo alla funzione e non soltanto il suo valore. Di conseguenza una **modifica del valore del parametro all'interno della funzione modifica il valore della variabile** passata come parametro.

```
void raddoppia (int& a, int& b, int& c)
                ↑x      ↑y      ↑z
raddoppia (    x    ,    y    ,    z    );
```

In altre parole noi abbiamo associato le variabili locali **a**, **b** e **c** (i *parametri formali* della funzione) alle variabili **x**, **y** e **z** (i *parametri attuali* passati nella chiamata alla funzione) in modo tale che **a** diventa *sinonimo* di **x**, **b** sinonimo di **y** e **c** sinonimo di **z**.

Ricordando che una variabile è il nome di una zona di memoria in cui può essere memorizzato un valore (il valore della variabile appunto), dire che **a** e **x** sono sinonimi significa che essi sono **nomi diversi per la stessa zona di memoria**. Se **a** ed **x** sono sinonimi, una modifica del valore di **a** ha come conseguenza la modifica del valore registrato nella zona di memoria comune ad **a** e **x** e dunque anche il valore di **x** cambia.

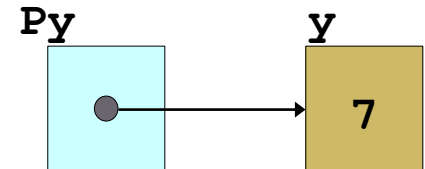
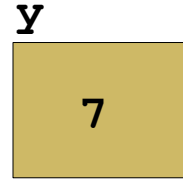
## Regole di progettazione di Procedure con parametri:

- Le procedure **devono comunicare all'esterno solo tramite parametri**
- Il **numero dei parametri non deve essere eccessivo** (altrimenti forse è sbagliata la progettazione)
- **Le procedure non devono usare variabili globali** e soprattutto non devono modificarle (evitare side-effect)
- **Le procedure** che risolvono un problema non devono usare istruzioni di input/output, ma **comunicare i dati solo tramite parametri**;
- La gestione dell'input/output deve avvenire in **apposite procedure**

# Variabili puntatore

Le variabili di tipo puntatore:

- Contengono come valore un indirizzo di memoria
- Le **variabili** (**y**) normali contengono uno specifico valore (indirizzamento diretto)
- La variabili **puntatore** (**Py**) contengono l'**indirizzo di una variabile** che a sua volta contiene un valore specifico (indirizzamento indiretto)



Un **puntatore** è una variabile che **memorizza** l'*indirizzo* di una locazione di memoria, cioè **l'indirizzo di una variabile**.

Un puntatore deve essere dichiarato come qualsiasi altra variabile, in quanto anch'esso **è una variabile**. Per esempio:

**int \*p;**

indica la dichiarazione di una variabile di tipo: *puntatore ad un intero*.

L'introduzione del carattere **\*** davanti al nome della variabile indica che si tratta di un puntatore del tipo dichiarato.

- Dichiarazione di puntatori

- Per i puntatori si usa il simbolo \*

```
int *p;
```

- Questo dichiara la variabile y come puntatore ad **int** (puntatore di tipo **int** \*)

- Se ci sono più dichiarazioni di puntatori, si usano più \*

```
int *p1, *p2;
```

- Si possono dichiarare puntatori a qualsiasi tipo
- I puntatori si inizializzano a **0**, **NULL**, o ad un indirizzo
  - **0** o **NULL** – puntano a nessun indirizzo (**NULL** è preferibile)

- **& (operatore indirizzo)**

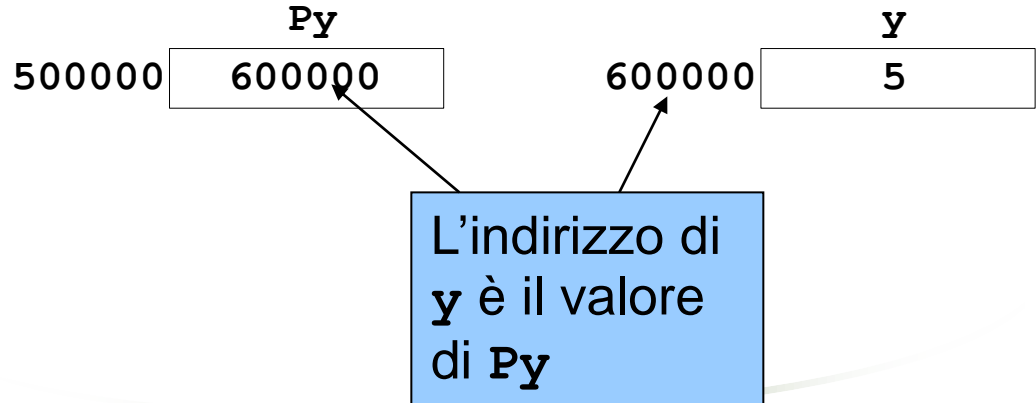
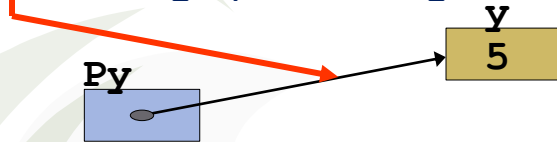
- Restituisce l'indirizzo dell'operando

```
int y = 5;           //dichiarazione di variabile intera y
```

```
int *Py;             //dichiarazione di variabile puntatore ad intero Py
```

```
Py = &y;             //assegna a Py l'indirizzo della variabile y
```

- Py "punta a" y



# Operatori & e \*

**L'operatore & restituisce l'indirizzo di una variabile.** Si consideri lo stralcio di codice seguente:

```
int *p , q, n;
```

```
p = &q;
```

```
n = q;
```

L'effetto dell'istruzione 2 è memorizzare l'indirizzo della variabile q nella variabile p. Dopo questa operazione, p **punta** a q.

L'operatore \* ha il seguente effetto: **se applicato ad una variabile puntatore restituisce il valore memorizzato nella variabile a cui punta:**

**p** memorizza l'*indirizzo*, o *punta*, ad una variabile

**\*p** restituisce il *valore* memorizzato nella variabile a cui punta p.

L'operatore \* viene chiamato operatore di **dereferenziazione**.



## \* (operatore di dereferenziazione)

– Restituisce un **sinonimo/alias** di quello a cui *punta* il suo operando

\* **p** restituisce **y** (perchè **p** punta a **y**)

- \* Può essere usato nelle assegnazioni
- Restituisce l'alias di un oggetto

```
int y=5;  
int *p = NULL;  
p = &y;  
*p = 7; // assegna 7 al valore della variabile puntata da p
```

p



y=7

# Puntatori e indirizzi

- Per **dichiarare un puntatore**, mettere **\*** davanti al nome.
- Per **ottenere l'indirizzo di una variabile**, utilizzare **&** davanti al nome.
- Per **ottenere il valore di una variabile**, utilizzare **\*** davanti al nome del puntatore.

Si consideri:

```
int *a , b , c;
```

```
b = 10;
```

```
a = &b;
```

```
c = *a;
```

*/\* a punta a b \*/*

*/\* c contiene il valore della variabile puntata da a, cioè 10 \*/*

Delle tre variabili dichiarate, **a** è un **puntatore ad intero**, mentre **b** e **c** sono **interi**.

L'ultima istruzione memorizza il valore della variabile puntata da a (ossia b) in c, quindi viene memorizzato in c il valore di b (10).

- Si noti che **b** è un int e **a** è un puntatore ad un int, quindi  
**b = a;**  
è un errore perchè cerca di memorizzare un *indirizzo* in un int.
- In modo analogo:  
**b = &a;**  
cerca di memorizzare l'*indirizzo* di un puntatore in un int ed è altrettanto errato.
- L'unico assegnamento sensato tra un int ed un puntatore ad un int è:  
**b = \*a;**  
Cioè a **b** si assegna il valore della variabile puntata da **a**
- **\*** e **&** sono **uno l'inverso dell'altro**, quindi si possono “eliminare” a vicenda:
  - **\*&p** -> **\* (&p)** -> **\* (indirizzo di p)**-> restituisce un alias di ciò a cui *punta* l'operando -> p
  - **&\*p** -> **&(\*p)** -> **&(y)** -> restituisce l'indirizzo di y, che è p -> p

# Parametri con i puntatori

```
// passaggio di parametri per riferimento  
#include <iostream>
```

```
void raddoppia (int *a, int *b, int *c)  
{  
    *a*=2;  
    *b*=2;  
    *c*=2;  
}
```

```
int main ()  
{  
    int x=1, y=3, z=7;  
    raddoppia (&x, &y, &z);  
    cout << "x=" << x << ", y=" << y << ", z=" << z;  
    return 0;  
}
```

La procedura raddoppia, può essere riscritta nel seguente modo se si usano **i puntatori**:

Nella **dichiarazione della funzione** i parametri sono **puntatori** e nella chiamata della funzione si passano gli indirizzi

L'utilizzo dei puntatori rende **più facile sapere se il passaggio è per indirizzo**

**risultato**  
**x=2, y=6, z=14**

# Licenza



Quest'opera è distribuita con Licenza [Creative Commons Attribuzione - Non commerciale - Non opere derivate 4.0 Internazionale](https://creativecommons.org/licenses/by-nc-nd/4.0/).