



UNIVERSITÀ  
degli STUDI  
di CATANIA

# Funzioni in C++

Corso di programmazione I AA 2019/20

Corso di Laurea Triennale in Informatica

---

Prof. Giovanni Maria Farinella

Web: <http://www.dmi.unict.it/farinella>

Email: [gfarinella@dm.unict.it](mailto:gfarinella@dm.unict.it)

Dipartimento di Matematica e Informatica

1. Introduzione alle funzioni in C++
2. Invocazione di funzioni

# Introduzione alle funzioni in C++

---

# Definizione e struttura di una funzione in C++

Le **funzioni** costituiscono la base della programmazione strutturata/procedurale.

Una funzione rappresenta un **blocco di codice** identificato da un **nome**.

Le funzioni andrebbero concepite per eseguire una o più attività **strettamente correlate tra loro**.

L'uso delle funzioni favorisce la **modularità** nei linguaggi come il C.

Nel C++ esistono le classi, alle quali vengono associate delle funzioni membro (incapsulamento) o “metodi”.

# Definizione e struttura di una funzione in C++

In C/C++ una funzione è costituita da:

- nome della funzione;
- tipo di ritorno;
- lista di argomenti o parametri formali;
- corpo della funzione (istruzioni) tra parentesi graffe;

```
1 | char func(string s, int i) {  
2 |     return s[i];  
3 | }
```

func(string, 2)

# Definizione e struttura di una funzione in C++

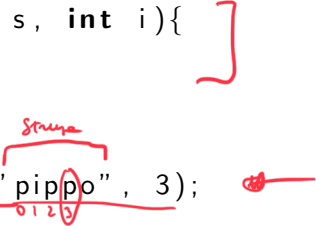
```
1 char func(string s, int i){  
2     return s[i];  
3 }  
4 ...  
5 char ret = func("pippo", 3);  
6 cout << ret;
```

Per la semplice funzione denominata func

- char è il tipo di ritorno;
- s e i sono i parametri formali di tipo rispettivamente string e int;

# Definizione e struttura di una funzione in C++

```
1  char func(string s, int i){  
2      return s[i];  
3  }  
4  
5  char ret = func("pippo", 3);  
6  cout << ret;
```



L'istruzione alla linea 2 **costituisce il corpo della funzione**. La parola chiave `return` fa sì che il flusso di esecuzione prosegua con la istruzione successiva alla chiamata a funzione;

La linea 5 costituisce la **invocazione** (o chiamata) della funzione.

## Chiamata a funzione

```
1  char func(string s, int i){  
2      return s[i];  
3  }  
4  
5  char ret = func("pippo", 3);  
6  cout << ret;
```

Il valore di ritorno della funzione viene **copiato** nella variabile `ret`.


L'istruzione eseguita successivamente alla istruzione della linea 2 è quella della linea 6.



# Definizione vs prototipo di funzione

## Prototipo della funzione

```
1 //dichiarazione del prototipo
2 double sum(double , double );
```



## Definizione della funzione.

```
1 //definizione
2 double sum(double p, double q){
3     double result = p + q;
4     return result;
5 }
```

# Definizione vs prototipo di funzione

Prototipo:

```
double sum(double , double );
```

*// oppure*

```
double sum(double p, double q);
```

- **tipo di ritorno;** ✓
- **segnatura:**
  - nome funzione;
  - lista parametri formali definiti da tipo e nome oppure semplicemente la lista dei tipi (i nomi dei parametri formali si possono omettere).

# Definizione vs prototipo di funzione

È buona pratica:

- raccogliere le **dichiarazioni dei prototipi** delle funzioni in appositi file *header* (ES: `modulo1.h`). Un header contiene in genere:
  - direttive `#define` e altre direttive `#include`
  - dichiarazione di costanti globali
  - prototipi di funzioni
  - dichiarazione di classi (si vedrà dopo)
- raccogliere la **definizione delle funzioni** (e metodi) in un modulo sorgente, ES: `modulo1.cpp`
- includere la direttiva `#include "modulo1.h"` in ogni file sorgente in cui si fa uso di tali funzioni.

*modulo1.cpp*

## Definizione vs prototipo di funzione

I moduli contenenti una o più funzioni si possono compilare separatamente in uno o più file *oggetto* da assemblare successivamente.





```
$ g++ -c modulo1.cpp  
$ g++ -c modulo2.cpp  
...  
$ g++ -c modulok.cpp  
$ g++ -c main.cpp
```

oppure

```
$ g++ -c main.cpp modulo1.cpp modulo2.cpp \  
[...] modulok.cpp
```

## Definizione vs prototipo di funzione

Il risultato sarà un set di file *oggetto*:

- modulo1.o 
- modulo2.o 
- ...
- modulok.o 
- main.o 

Infine si possono assemblare (*fase di linking* – produce eseguibile):

```
$ g++ main.o modulo1.o modulo2.o ... modulok.o
```

## Definizione vs prototipo di funzione

```
18_00_main.cpp  
18_00_func.cpp  
18_00.h
```

## Invocazione di funzioni

---

# Parametri formali vs parametri attuali

La lista di argomenti presenti nella segnatura di una funzione o metodo è detta lista di **parametri formali**.

I valori passati nella invocazione della funzione vengono detti **parametri attuali**.

```
1 void foo(int x){ //x parametro formale
2   //...
3 }
4 int main(){
5   // ...
6   int a;
7   foo(a); // a parametro attuale
8 }
```



## Parametri formali vs parametri attuali

Nella programmazione strutturata/procedurale, il flusso è rappresentato da **una sequenza di invocazioni di funzioni**.

I parametri permettono alle funzioni di **scambiare dati**.

In C/C++, per ottenere un programma eseguibile, è “obbligatorio” fornire al compilatore una (e una sola) funzione denominata `main`.

La prima istruzione della esecuzione della applicazione è rappresentata dalla invocazione della funzione `main()`.

La **porzione di memoria riservata** allo stoccaggio dei dati utili alla esecuzione delle **istruzioni contenute nel corpo delle funzioni** è denominata **stack** (pila).

Lo stack è una struttura in cui i dati vengono inseriti e prelevati in base al meccanismo LIFO (Last In - First Out).

Il singolo dato viene depositato (push) sempre sul top dello stack.

Si può prelevare un dato alla volta (pop), solo dal top dello stack.

# Area stack e record di attivazione

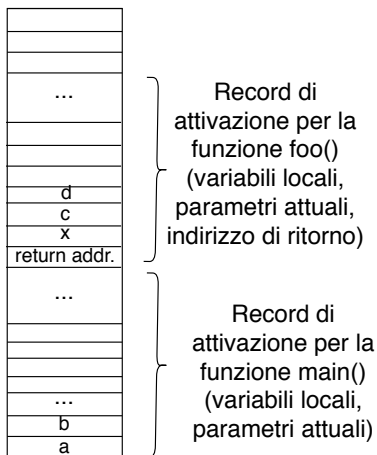
## Segmento STACK



```
1  void foo(int x){  
2      double c, d;  
3      // ...  
4  }  
5  int main(){  
6      int a, b;  
7      // ...  
8      foo(a);  
9      // ..  
10 }
```

# Area stack e record di attivazione

## Segmento STACK



```
1  void foo(int x){  
2      double c, d;  
3      //...  
4  }  
5  int main(){  
6      int a, b;  
7      //...  
8      foo(a);  
9      //..  
10 }
```

1-Un record per la funzione **foo()** viene allocato sul segmento stack.

# Area stack e record di attivazione

## Segmento STACK



```
1 void foo(int x){  
2     double c, d;  
3     // ...  
4 }  
5 int main(){  
6     int a, b;  
7     // ...  
8     foo(a);  
9     // ..  
10 }
```

2-1 parametri attuali vengono depositati nello stack: in questo caso il valore del parametro attuale b, denominato x nella lista di parametri formali;

# Area stack e record di attivazione

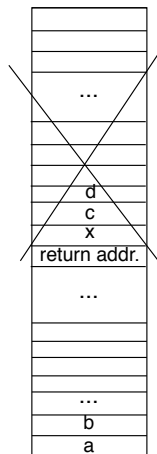


```
1 void foo(int x){  
2     double c,d;;  
3     // ...  
4 }  
5 int main(){  
6     int a,b;  
7     // ...  
8     foo(a);  
9     // ..  
10 }
```

3-Vengono poi allocate le variabili locali definite all'interno della funzione (c e d);

# Area stack e record di attivazione

## Segmento STACK



Record di  
attivazione per la  
funzione main()  
(variabili locali,  
parametri attuali)

```
1 void foo(int x){  
2     double c,d;  
3     //...  
4 }  
5 int main(){  
6     int a,b;  
7     //...  
8     foo(x);  
9     //..  
10 }
```

4-A seguito di una istruzione `return` o al arraggiungimento della fine della funzione `foo()`, il flusso prosegue con la istruzione successiva alla chiamata a `foo()`.

## Passaggio mediante indirizzo e per valore

Passaggio “per valore” di un dato ad una funzione.

Il valore attuale del dato viene copiato sul record di attivazione dello stack (ES: a). .

```
1 void foo(int x){ //x parametro formale
2   //...
3 }
4 int main(){
5   // ...
6   int a = 10;
7   foo(a);
8   cout << a;
9 }
```



## Passaggio mediante indirizzo e per valore

**Conseguenza del passaggio per valore:** dato che la funzione opera su una copia di *a*, non può modificarne il valore.

```
1  void foo(int x){ //x parametro formale
2      //...
3      x = 90;
4  }
5  int main(){
6      // ...
7      int a = 10;
8      foo(a); // parametro attuale e' VALORE in a
9      cout << a; //stampa 10!
10 }
```

La istruzione alla linea 3 non ha alcun effetto sul valore di *a*!

## Passaggio mediante indirizzo e per valore

**Passaggio mediante indirizzo.** La funzione riceve l'indirizzo del dato (il puntatore), quindi può operare modifiche al dato della “funzione chiamante” mediante **l'operatore di dereferenziazione o indirezione**.

```
1  void spam(int *x){
2      //...
3      *x = 90;
4  }
5  int main(){
6      // ...
7      int a = 10;
8      spam(&a); // parametro attuale e' INDIRIZZO di a
9      cout << a; //stampa 90!
10 }
```

# Premessa: categorie di allocazione o memorizzazione delle variabili

## Allocazione automatica:

- la memoria viene allocata mediante una dichiarazione di una **variabile locale** ad una funzione.
- Scope/visibilità **limitato al blocco di codice** in cui è stata dichiarata.
- Ciclo di vita del blocco allocato termina con la fine dell'esecuzione del blocco in cui viene
- Area di memoria usata è denominata **STACK**.

```
void foo(){  
    int a = 0; // a visibile solo in foo()  
}
```

# Premessa: categorie di allocazione o memorizzazione delle variabili

## Allocazione dinamica

- Effettuata mediante operatore `new` in qualsiasi punto del programma.
- Area di memoria usata è denominata `HEAP`.
- Ciclo di vita del blocco di memoria termina con invocazione di operatore `delete` sul puntatore.

```
//qualsunque punto del programma  
int *p = new int(2); // cella int, valore iniziale 2  
//...  
delete p; // deallocazione della cella puntata da p
```

# Premessa: categorie di allocazione o memorizzazione delle variabili

## Allocazione statica.

- Dichiarazione di variabili al di fuori da qualunque blocco.
- Segmento di memoria ospitante è detto segmento DATA.
- Ciclo di vita / scope: inizia e termina con il programma stesso.

```
#include <iostream>
//...
double data = 0.5;
//...
int main(){
    // ...
}
```

## Esempi

`18_mem.cpp`

## Passaggio di parametri array

**Il passaggio di un array come parametro di una funzione avviene sempre per indirizzo.** (il nome di uno array è un puntatore costante al primo elemento dello array..).

```
1 void init(int *v, int n){
2     //...
3     for(int j=0; j<n; j++){
4         v[j] = 0;
5     }
6 }
7 int main(){
8     // ...
9     int x[10];
10    init(x, 10);
11 }
```

## Passaggio di parametri array

Forma equivalente..

```
1  void init(int v[], int n){ //equivalente a int *v, ...
2      //...
3      for(int j=0; j<n; j++){
4          v[j] = 0;
5      }
6  }
7  int main(){
8      // ...
9      int x[10];
10     init(x, 10);
11 }
```



## Passaggio di parametri array

Per il passaggio di **array multidimensionali allocati sul segmento DATA o sullo STACK**, nel prototipo della funzione che riceve il dato, vanno specificate tutte le dimensioni, dalla seconda in poi.

```
1  #define N 5
2  #define M 10
3  void init(int v[][M]); // OK
4  void init(int v[N][M]); // OK
5  void init(int v[][], int n); // Err. di compilazione!
6  void foo(){
7      int w[N][M];
8      init(w, N);
9  }
```

Si può eventualmente specificare anche la prima, che comunque non verrà usata.

# Passaggio di parametri array

Attenzione!

```
1 void init(int *v[M], int n);  
2 //...  
3 int x[10][10]; //allocazione statica o automatica  
4 init(x, 10); // NO: Errore di compilazione!
```

Alla linea 1 il compilatore interpreta come **un vettore di M puntatori a int**(a causa della maggiore precedenza dello operatore [] rispetto a \*).

La dichiarazione alla linea 1, **sebbene sintatticamente corretta**, provoca un **errore di compilazione in corrispondenza della invocazione alla linea 4**.

## Passaggio di parametri array

La seg. invece viene interpretata come **un puntatore a vettore di M interi**.

```
1 void init(int (*v)[M], int n); // OK
2 //...
3 int x[10][10]; //allocazione statica o automatica
4 init(x, 10); // OK
```

In questo caso la chiamata `init(x)` è lecita.

# Passaggio di parametri array

Array a tre dimensioni..

```
1  #define N 10
2  #define M 10
3  #define L 5
4  void init(int w[][M][L], int n); //OK
5  void init(int w[N][M][L], int n); //OK
6  void init(int w[][][L], int n); //Err. di compilazione!
```

Le linee 4 e 5 contengono dichiarazioni **valide**, ci sono tutte le dimensioni che andavano specificate, dall'ultima alla seconda.

Alla linea 6 manca la seconda dimensione, quindi è una dichiarazione **non valida**.

# Passaggio di parametri array

Attenzione!

```
1 void init(int *v[M][L], int n);  
2 int x[10][M][L];  
3 init(x, 10); // Errore di compilazione!
```

Mentre la seg. è corretta...

```
1 void init(int (*v)[M][L], int n); // OK  
2 int x[10][M][L];  
3 init(x, 10); // OK
```

La dichiarazione alla linea 1 viene interpretata come **un puntatore ad un array bidimensionale di interi dimensioni M x L**, quindi invocazione della linea 3 è valida.

## Allocazione automatica o statica di array.

```
1  #define ROWS 3
2  #define COLS 4
3  int main(){
4      //...
5      int v[ROWS][COLS]; //segmento STACK
6      //...
7  }
```

La istruzione alla linea 5 implica la **allocazione** di un blocco di  $\text{ROWS} \times \text{COLS}$  celle di memoria **contigue** di tipo `int` sullo *stack* (allocazione automatica).

# Allocazione automatica o statica di array.

Allocazione **statica**.

```
#define ROWS 3
#define COLS 4

int v[ROWS][COLS]; //segmento DATA
int main(){
    //...
}
```

**Allocazione** di un blocco ROWS×COLS celle **contigue** di tipo int nel segmento DATA.

# Allocazione automatica o statica di array.

```
#define ROWS 3
#define COLS 4
int main(){
    //...
    int v[ROWS][COLS]; //sullo stack
    //...
}
```

int v[4][3];



→  
Celle di memoria contigue

-----Stack----->



## Allocazione automatica o statica di array.

```
int v[ROWS][COLS];
```

Per v allocato come sopra, la seg. espressione (1)

```
(1) &v[i][j]; // indici
```

è equivalente alle seg. espressioni (2) e (3):

```
(2) (*(v+i) + j); //aritmetica dei punt.
```

```
(3) (v[i] + j); //aritmetica dei punt. e indici
```

Entrambe le espressioni rappresentano **l'indirizzo della cella agli indici (i,j).**

## Allocazione automatica o statica di array.

La seg. espressione (4)

```
(4)  v[i][j]
```

è equivalente alle segg. espressioni (5) e (6) :

```
(5)  (*(v+i) + j); //aritmetica dei punt.
```

```
(6)  *(v[i] + j); //arit. punt. + operatore [] .
```

Entrambe le espressioni rappresentano il **valore contenuto nella cella agli indici (i,j)**.

## Allocazione automatica o statica di array.

Le segg. espressioni rappresentano indirizzi di memoria e danno identico risultato

$(v+i)$  oppure  $v[i]$   
equivale a ..  
 $\&v[0][0] + i \times COLS.$

Di conseguenza, applicando l'operatore di indirezione  $*$  è possibile ottenere l'indirizzo dello elemento  $v[i][j]$ , ovvero  $\&v[i][j]$ .

$*(v+i) + j$  oppure  $*v[i] + j$   
equivale a ...  
 $\&v[0][0] + i \times COLS + j.$

A18\_01.cpp

A18\_02.cpp

PTR\_01.cpp

...

PTR\_13.cpp

**FINE**