



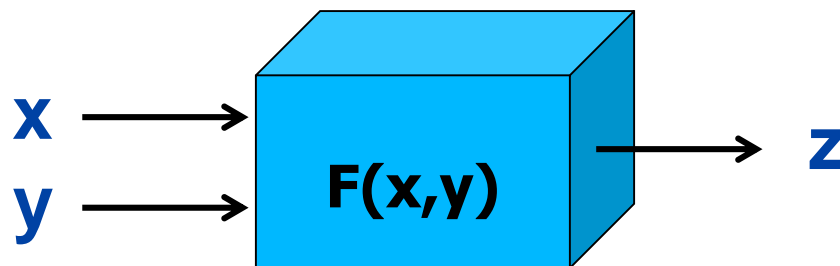
FUNZIONI

Introduzione alle funzioni in C++

Roberto Nardone, Luigi Romano

- ❑ Concetto di funzione
- ❑ Struttura di una funzione
 - Dichiarazione di una funzione
 - Definizione di una funzione
 - Passaggio di parametri ad una funzione
 - Parametri di default
- ❑ Funzioni inline
- ❑ Visibilità e classi di immagazzinamento
- ❑ Sovraccaricamento di una funzione
- ❑ Funzioni della libreria standard

- ❑ Una funzione può essere interpretata come una scatola nera che, a partire da un insieme di argomenti in input, produce un risultato.



- ❑ Una funzione “**incapsula**” un certo numero di istruzioni in un blocco di codice e per modificarne il comportamento sarà sufficiente agire sul blocco di cui è composta.
- ❑ Ad ogni funzione è assegnata una **firma** attraverso la quale il relativo blocco di codice potrà essere invocato.

❑ Vantaggi delle funzioni:

- **riutilizzo del codice**

Il set di istruzioni incapsulato nella funzione può essere invocato più volte all'interno del programma.

- **maggiore leggibilità**

La firma di una funzione fornisce un'informazione di alto livello sul sotto-problema che andrà a risolvere.

- **codice più snello**

Le funzioni evitano al programmatore di dover riscrivere più volte uno stesso set di istruzioni, riducendo la probabilità di errori e la dimensione del codice eseguibile.

- ❑ L'invocazione di una funzione è **sempre preceduta** dalla sua **dichiarazione o prototipo**.
- ❑ La dichiarazione di una funzione prevede:
 - il **tipo** del valore di ritorno dalla funzione
 - il **nome** della funzione
 - **tipo** e **nome** dei parametri/argomenti di ingresso alla funzione
 - simbolo di **terminazione** della dichiarazione (;)

float AreaTriangolo(**float base, float altezza**);

tipo del valore di ritorno

nome della funzione

tipo e nome dei parametri di ingresso

simbolo di terminazione

- ❑ La **definizione** di una funzione corrisponde all'insieme di istruzioni che verranno eseguite ogni qual volta verrà invocata.
- ❑ Il blocco di istruzioni da eseguire all'atto dell'invocazione è detto **corpo della funzione** ed è racchiuso tra i simboli “{ }”.
- ❑ Eventuali variabili definite all'interno del corpo della funzione esistono solo all'interno della funzione stessa (visibilità locale).

```
float AreaTriangolo(float base, float altezza)
{
    return (base*altezza)/2;
}
```

- ❑ La parola chiave **return** restituisce al chiamante il risultato dell'espressione che la segue.

AreaTriangolo(b, h)

AreaTriangolo.cpp

```
1  /* File: AreaTriangolo.cpp
2  */
3  #include<iostream>
4  using namespace std;
5
6  float AreaTriangolo(float, float);
7
8  int main()
9  {
10     float b, h;
11
12     cout<<"Base = ";
13     cin >> b;
14
15     cout<<"Altezza = ";
16     cin >> h;
17
18     cout << "Area = " << AreaTriangolo(b, h) << endl;
19
20     return 0;
21 }
22
23
24 float AreaTriangolo(float base, float altezza)
25 {
26     return (base*altezza)/2;
27 }
28
```

- ❑ Prototipo e definizione devono avere **tipo di ritorno, nome, tipo e numero dei parametri** corrispondenti.
- ❑ Allo stesso modo, il tipo di una variabile **x** a cui verrà assegnato il risultato di una funzione **F** dovrà essere compatibile con il tipo restituito da **F**.

double sqrt(double);

double x1=sqrt(2.1);	Ok
double x2=sqrt(2.3,1.5);	Errore!
double x3=sqrt("two");	Errore!!
char c=sqrt(2.9);	Errore!!!

- ❑ In C++ il passaggio dei parametri ad una funzione può seguire due diverse modalità:
 - **passaggio per valore**
 - **passaggio per riferimento**

- ❑ Nel passaggio per valore la funzione legge il valore dei parametri di ingresso, li **copia** nelle corrispondenti variabili interne e infine restituisce il risultato.

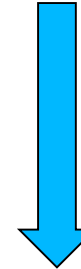
- ❑ Lavorando su di una **copia**, la funzione non potrà apportare modifiche accidentali ai parametri di ingresso. Questo si traduce in una maggiore sicurezza del codice a discapito delle performance.

- ❑ Nel **passaggio per riferimento** ciò che viene copiato nel parametro interno della funzione, non è il valore dell'argomento, ma il corrispondente indirizzo di memoria.
- ❑ All'atto della dichiarazione di una funzione, il passaggio dei parametri per riferimento sarà realizzato attraverso **l'operatore unario di referenziazione (&)**.
- ❑ In questo modo posso sfruttare anche i parametri di ingresso alla funzione per apportare modifiche allo stato del programma.
- ❑ **Effetto collaterale**: è la modifica da parte di una funzione di un valore o uno stato al di fuori del proprio scoping locale.
 - ❑ Una modifica accidentale potrebbe invalidare l'esecuzione di un algoritmo formalmente corretto.

Value.cpp



g++ Value.cpp -o val
./val



```
#include<iostream>

using namespace std;

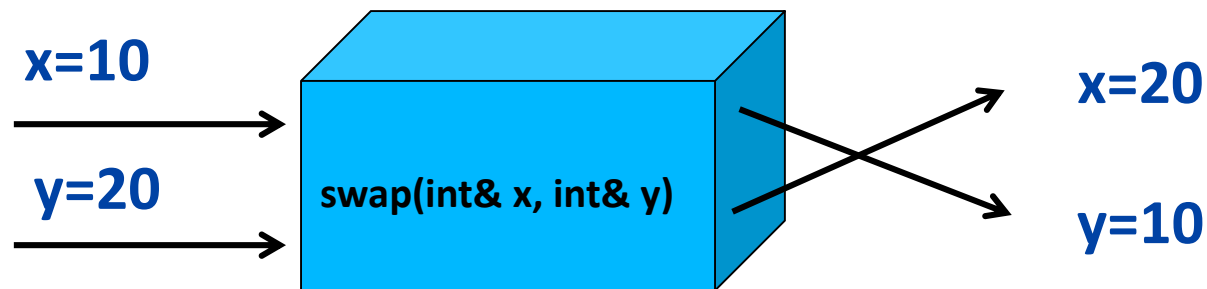
int f(int a){
    a=a+10;
    int b=a*a;
    return b;
}

int main(){
    int x,y;
    x=10;
    y=f(x);
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
}
```

```
osboxes@osboxes: ~
File Edit Tabs Help
osboxes@osboxes:~$ cd Desktop/
osboxes@osboxes:~/Desktop$ g++ Value.cpp -o val
osboxes@osboxes:~/Desktop$ ./val
x=10
y=400
```

Passaggio per valore: le modifiche effettuate dalla funzione 'f' restano **confinare nell'ambito della funzione** (la variabile x definita nel programma principale non viene modificata).

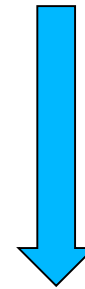
- ❑ **Obiettivo:** dimostrare che attraverso il passaggio dei parametri per riferimento è possibile restituire più di un valore di uscita.
- ❑ Definiamo una funzione **swap** il cui compito è quello di scambiare il valore dei suoi 2 parametri di ingresso passati per riferimento.



Reference.cpp



g++ Reference.cpp -o ref
./ref



```
#include<iostream>

using namespace std;

void swap(int& x, int& y){
    int temp = x;
    x=y;
    y = temp;
}

int main(){
    int x=10;
    int y=20;
    swap(x, y);
    cout<<"x="<<x<<endl;
    cout<<"y="<<y<<endl;
}
```

```
osboxes@osboxes: ~/Desktop/Funzioni
File Edit Tabs Help
osboxes@osboxes:~/Desktop/Funzioni$ g++ Reference.cpp -o ref
osboxes@osboxes:~/Desktop/Funzioni$ ./ref
x=20
y=10
```

Passaggio per riferimento: le modifiche effettuate dalla funzione 'swap' hanno **effetto sui dati del programma** (i valori delle variabili x e y definite nel programma principale vengono scambiati).



- ❑ In un'analogia implementazione della funzione di **swap**, ma con passaggio dei parametri **per valore**, che risultati avremmo ottenuto? Le variabili **x** ed **y** definite nel programma principale sarebbero state scambiate?

- Abbiamo visto in precedenza che è possibile passare i parametri alle funzioni in 2 modi diversi:
 - Per **valore** `int somma(int a, int b){...}`
 - Per **riferimento** `int somma(int &a, int &b){...}`
- C'è una terza possibilità:
 - Per **indirizzo** `int somma(int *a, int *b){...}`
- Per **valore** si passa una copia della variabile, mentre negli ultimi due casi si passa sempre un indirizzo, con la differenza che nel **passaggio per riferimento** si potrà fare una gestione semplificata degli indirizzi.

```
#include<iostream>

using namespace std;

void swap(int *a, int *b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int i=1;
    int j=2;
    swap(&i, &j);
    cout<<"i="<<i<<" j="<<j<<endl;
}
```

**Argomento
puntatore ad
intero**

Dereferenziazione


```
ubuntu@ubuntu:~/cppprogs$  
ubuntu@ubuntu:~/cppprogs$ g++ -o Pointer.exe Pointer.cpp  
ubuntu@ubuntu:~/cppprogs$ ./Pointer.exe  
x = 10  
y = 20  
x = 20  
y = 10  
ubuntu@ubuntu:~/cppprogs$
```



□ Definizione

```
void incrementa(int& i) { i++; }  
void incrementa(int* i) { (*i)++; }
```

□ Chiamata

```
incrementa(i);  
incrementa(&i);
```

Incrementa.cpp

```
1  /* File: Incrementa.cpp
2  */
3  #include<iostream>
4  using namespace std;
5
6  void incrementa(int& );
7  void incrementa(int* );
8
9  int main()
10 {
11     int n = 0;
12
13     incrementa(n);
14     cout << "n = " << n << endl;
15
16     incrementa(&n);
17     cout << "n = " << n << endl;
18
19     return 0;
20 }
21
22
23 void incrementa(int& i)
24 {
25     i++;
26 }
27
28 void incrementa(int* i)
29 {
30     (*i)++;
31 }
```

Decrementa.cpp

```
1  /* File: Decrementa.cpp
2  */
3  #include<iostream>
4  using namespace std;
5
6  void decrementa(int& );
7  void decrementa(int* );
8
9
10 int main()
11 {
12     int n = 2;
13
14     decrementa(n);
15     cout << "n = " << n << endl;
16
17     decrementa(&n);
18     cout << "n = " << n << endl;
19
20     return 0;
21 }
22
23
24 void decrementa(int& i)
25 {
26     i--;
27 }
28
29 void decrementa(int* i)
30 {
31     (*i)--;
32 }
```

- ❑ Come per le variabili, anche gli array possono essere passati come parametro ad una funzione.
- ❑ Rispetto alle variabili, il passaggio di un array come parametro avviene **solo per riferimento**.
- ❑ Il riferimento che viene passato è l'indirizzo del primo elemento dell'array, cioè il **puntatore** all'area di memoria in cui è memorizzato il primo elemento dell'array.
- ❑ Quindi, se la funzione modifica i valori dell'array, la modifica avrà effetto anche sull'array passato alla funzione.

```
#include <iostream>

using namespace std;

void azzera_vettore(int v[], int dim)
{
    int i;
    for (i=0; i<dim; i++)
    {
        v[i]=0;
    }
}

int main()
{
    int vettore[5]={1,5,12,0,-1};
    cout<<"L'array prima della chiamata è:"<<endl;
    for(int i=0; i<5;i++)
    {
        cout<<"vettore["<<i<<"]="<<vettore[i]<<endl;
    }

    cout<<"-----"<<endl;
    azzera_vettore(vettore, 5);
    cout<<"L'array all'uscita della funzione è:"<<endl;
    for(int j=0; j<5;j++)
    {
        cout<<"vettore["<<j<<"]="<<vettore[j]<<endl;
    }
    return 0;
}
```

La funzione `azzera_vettore` pone a zero i valori all'interno dell'array passato in ingresso.

Il secondo parametro `dim` serve per specificare la dimensione dell'array e quindi l'area di memoria che dovrà ospitarlo.

L'array viene «sempre» passato alla funzione per riferimento, quindi all'uscita dalla funzione `azzera_vettore` l'annullamento dei valori dell'array avrà effetto anche all'esterno della funzione.

```
#include <iostream>

using namespace std;

void azzera_vettore(
{
    int i;
    for (i=0;
    {
        v[
    }
}

int main()
{
    int vettore[5];
    cout<<"L'a
    for(int i=
    {
        co
    }

    cout<<"---
    azzera_vet
    cout<<"L'a
    for(int j=
    {
        co
    }

    return 0;
}
```

```
osboxes@osboxes: ~/Desktop
File Edit Tabs Help
osboxes@osboxes:~/Desktop$ g++ passaggio.cpp -o pas
osboxes@osboxes:~/Desktop$ ./pas
L'array prima della chiamata è:
vettore[0]=1
vettore[1]=5
vettore[2]=12
vettore[3]=0
vettore[4]=-1
-----
L'array all'uscita della funzione è:
vettore[0]=0
vettore[1]=0
vettore[2]=0
vettore[3]=0
vettore[4]=0
osboxes@osboxes:~/Desktop$
```

La funzione `azzera_vettore` pone a zero tutti gli elementi dell'array.

Il parametro `dim` serve a indicare la dimensione dell'array, per poterlo iterare.

Il parametro `re` passato per riferimento, fa sì che la funzione agisca sul vettore originale, e non su una copia. L'effetto si riflette anche all'esterno della funzione.

- ❑ Scrivere una funzione che cerca il minimo in un vettore, testarla con un main che cerca il minimo in tre diversi vettori generati casualmente
- ❑ Scrivere una funzione che dati due numeri ritorni il minimo comune multiplo calcolato con algoritmo di Euclide
- ❑ Scrivere una funzione che dato un vettore ritorni il vettore con gli elementi negative azzerati

- ❑ In C++ è possibile definire funzioni in cui alcuni argomenti (o tutti) possono assumere un valore di **default**.
- ❑ Questa funzionalità è utile nei casi in cui tali argomenti assumono dei valori tipici relativi alla funzione in oggetto.
- ❑ I parametri di default di una funzione devono essere:
 - agli ultimi posti nella lista dei parametri della funzione
 - valori costanti
- ❑ Se una funzione è dichiarata con più parametri di default, all'atto della chiamata, è possibile omettere un parametro solo se anche quelli successivi sono omessi.

- ❑ Consideriamo il seguente il prototipo di una funzione per il calcolo del volume di una scatola:

```
int calcolaVolume(int lunghezza=1, int altezza=1, int larghezza=1);
```

3 parametri di default

- ❑ Alcuni esempi di chiamate lecite sono:

`calcolaVolume();` // Utilizza **tutti** gli argomenti di default

`calcolaVolume(10);` //Annulla il **1°** parametro di default

`calcolaVolume(10,12);` //Annulla **1°** e **2°** parametro di default

`calcolaVolume(10,12,10);` //Annulla **tutti** i parametri di default

- ❑ Le seguenti chiamate, invece, produrranno un errore in fase di compilazione:

```
calcolaVolume( , 12, 15);  
calcolaVolume( , , 15);
```

- ❑ Prestata la dovuta attenzione, gli argomenti di default sono utili per semplificare la scrittura delle chiamate alle funzioni.

ESEMPIO ARGOMENTI DI DEFAULT

Open

Volume.cpp
~/Desktop/Funzioni

```

#include<iostream>

using namespace std;

int calcolaVolume(int l=1, int h=1, int w=1){

    return l*h*w;
}

int main(){
    int l=2;
    cout<<"L'area della scatola di lunghezza 2 è:"<<calcolaVolume(l)<<endl;
    cout<<"-----"<<endl;
    int h=3;
    cout<<"L'area della scatola di lunghezza 2 e altezza 3 è:"<<calcolaVolume(l,h)<<endl;
    cout<<"-----"<<endl;
    int w=2;
    cout<<"L'area della scatola di lunghezza 2 , altezza 3 e larghezza 2 è:"<<calcolaVolume(l,h,w)<<endl;
}

```

osboxes@osboxes: ~/Desktop/Funzioni

File Edit Tabs Help

```

osboxes@osboxes:~$ cd Desktop/Funzioni/
osboxes@osboxes:~/Desktop/Funzioni$ g++ Volume.cpp -o vol
osboxes@osboxes:~/Desktop/Funzioni$ ./vol
L'area della scatola di lunghezza 2 è:2
-----
L'area della scatola di lunghezza 2 e altezza 3 è:6
-----
L'area della scatola di lunghezza 2 , altezza 3 e larghezza 2 è:12
-----
osboxes@osboxes:~/Desktop/Funzioni$

```

- ❑ Il C++ prevede una speciale direttiva al compilatore che consiste nel sostituire la chiamata a funzione con il corpo della funzione stessa.
- ❑ Tale direttiva è applicata alla definizione della funzione attraverso la keyword **inline**.
- ❑ L'uso di funzioni **inline** comporta un minore tempo di esecuzione a patto che le funzioni a cui viene applicata la direttiva siano semplici ed invocate frequentemente.
- ❑ Il compilatore può autonomamente ignorare la direttiva **inline** (funzioni complesse potrebbero sovraccaricare il compilatore).

- ❑ La **definizione** di una funzione **inline** si realizza come segue:

```
inline tipo_rit nome_funzione (argomenti)
{
    corpo della funzione
}
```

- ❑ Per una funzione **inline** il compilatore ricopia realmente il codice della funzione in ogni punto in cui essa viene chiamata.
- ❑ Se in un programma la funzione **inline** viene invocata 10 volte, il compilatore inserisce 10 copie della funzione nel programma, con conseguente aumento di 10 volte della dimensione del programma.

```
#include <iostream>

using namespace std;

inline int mult(int a, int b)
{
    return a*b;
}

int main()
{
    mult(5,2);
}
```

Il compilatore riconosce la direttiva **inline** e ricopia l'istruzione contenuta nel suo blocco nel punto del codice in cui la funzione è stata invocata.

1. Scrivere una funzione che, dato un vettore di **n** interi (forniti in input da tastiera), restituisca la media dei soli interi non negativi.
2. Scrivere una funzione che dato un intero fornito in input da tastiera (Default 0), restituisca il corrispondente fattoriale. Il fattoriale di zero è 1, il fattoriale di n è $n(n-1)!$.
3. Scrivere una funzione che, dati due numeri interi **n** ed **m** forniti dall'utente ($n > m$), calcoli l'insieme dei numeri multipli di **m** nell'insieme $[1, n]$. Al termine, la funzione stamperà a video:
 - I. Il numero di elementi multipli nell'intervallo $[1, n]$
 - II. Tutti i numeri multipli di m nell'intervallo $[1, n]$.

- ❑ Stabiliscono le regole di accesso per le variabili **globali**, **locali** e i **parametri formali** in relazione alle funzioni.
- ❑ Una variabile globale è creata con una dichiarazione esterna a tutte le funzioni del programma (**main() compreso**) ed è accessibile da tutte le funzioni ed espressioni che seguono la sua dichiarazione.
- ❑ Una variabile dichiarata all'interno di una funzione è **locale** e accessibile soltanto da quella funzione.
- ❑ Quando una variabile globale ed una locale hanno lo stesso nome, quella locale occulta la variabile globale.
- ❑ I **parametri formali** si comportano come qualsiasi altra variabile locale e il loro ambito è locale alla sua funzione.

VARIABILI GLOBALI:

VANTAGGI vs SVANTAGGI

- ❑ Le variabili globali sono di aiuto quando più funzioni del programma fanno riferimento agli stessi dati, oppure quando un valore deve essere conservato per tutta la durata del programma.
- ❑ Un uso eccessivo delle variabili globali è sconsigliato perché:
 - occupano la memoria per tutto il tempo di esecuzione del programma (non soltanto quando sono necessarie);
 - rendono meno generale il codice;
 - una modifica indesiderata ad una variabile globale pregiudica il corretto funzionamento dell'intero programma.

- ❑ La **visibilità** o **scope** definisce la possibilità di richiamare un identificatore, tipicamente una **variabile**, in un determinato punto del programma.

- ❑ Esistono 4 tipi di visibilità:
 - a livello di file;
 - a livello di blocco;
 - a livello di prototipo di funzione;
 - a livello di funzione.

- ❑ Un identificatore dichiarato al di fuori di qualsiasi funzione ha visibilità **a livello di file**.
- ❑ Un identificatore di questo tipo è noto a tutte le funzioni che si trovano dopo la sua dichiarazione, fino alla fine del file.
- ❑ Le variabili globali, le definizioni di funzioni e i prototipi di funzioni che si trovano al di fuori delle funzioni hanno visibilità **a livello di file**.

- ❑ Gli identificatori dichiarati in un blocco hanno visibilità **a livello di blocco**.
- ❑ In questo caso lo scope dell'identificatore inizia alla dichiarazione dell'identificatore e termina con la chiusura del blocco.
- ❑ Le variabili locali dichiarati all'inizio di una funzione hanno visibilità **a livello di blocco**.
- ❑ Nel caso di blocchi nidificati, se un identificatore del blocco esterno ha lo stesso nome di un identificatore del blocco interno, quello del blocco esterno è occultato da quello del blocco interno.

- ❑ Gli unici identificatori che hanno visibilità **a livello di prototipo di funzione** sono gli argomenti elencati nei prototipi di funzione.
- ❑ Le **etichette** o **label**, invece, sono gli unici identificatori che hanno visibilità **a livello di funzione**.
- ❑ Le **etichette** sono utilizzate nei costrutti **switch** o nelle istruzioni **goto** e vengono seguiti dal simbolo **:** (i.e. **label:**).
- ❑ Le funzioni applicano il principio dell'occultamento dell'informazioni, nascondendo le proprie **etichette** alle altre funzioni.

```
#include<iostream>
```

```
using namespace std;
```

```
int i=5;  
void decremento(int);
```

Visibilità a livello file

```
void incremento()  
{
```

Visibilità a livello di prototipo

```
    int i=2;  
    i++;
```

Visibilità a livello blocco

Blocco
esterno

Blocco
interno

```
    {  
    }  
}
```

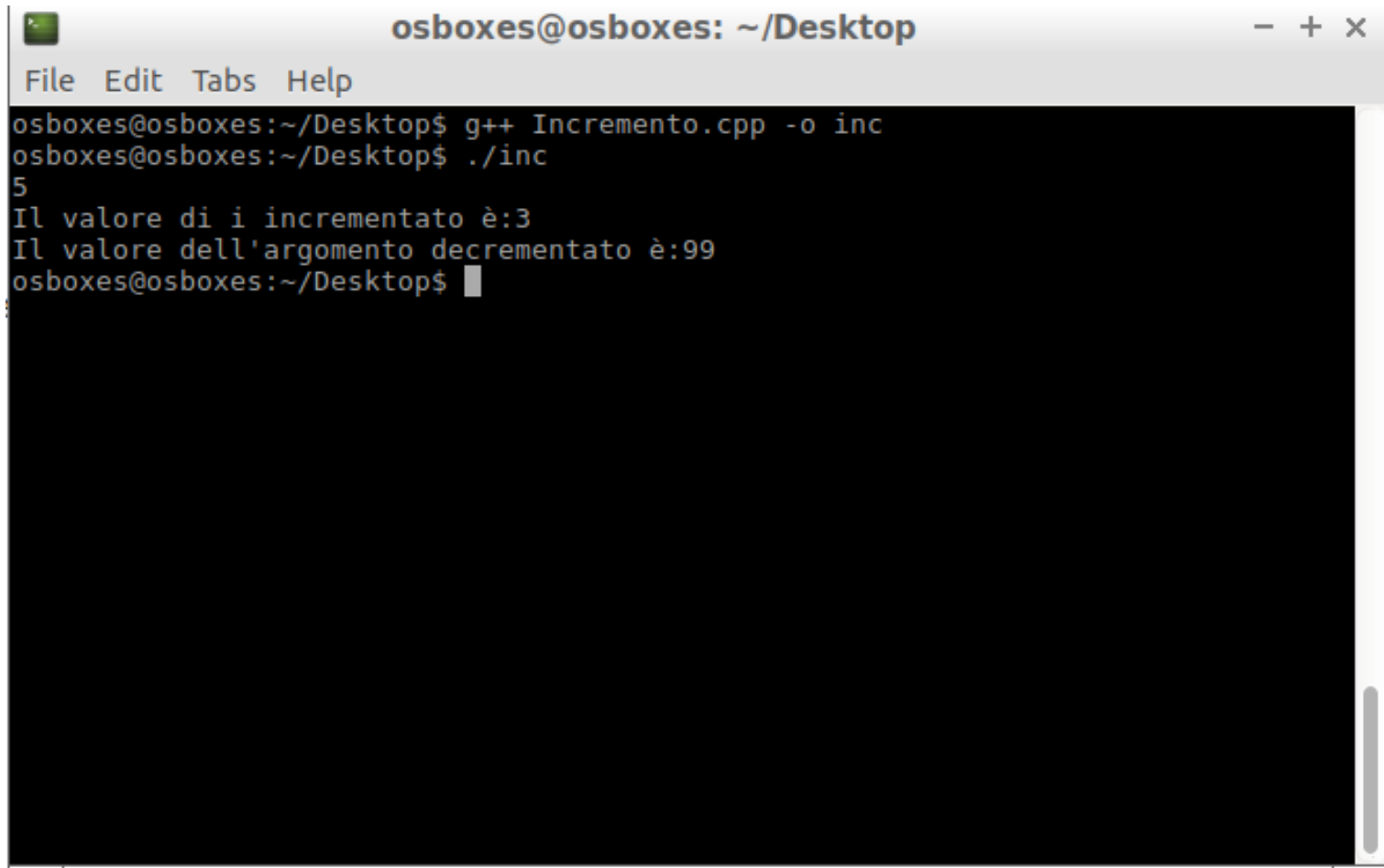
```
        int i=7;  
        i++;
```

```
    cout<<"Il valore di i incrementato è:"<<i<<endl;
```

```
void decremento(int j)  
{
```

```
    j--;  
    cout<<"Il valore dell'argomento decrementato è:"<<j<<endl;
```

```
int main(){  
    cout<<i<<endl;  
    incremento();  
    decremento(100);  
}
```



```
osboxes@osboxes: ~/Desktop
File Edit Tabs Help
osboxes@osboxes:~/Desktop$ g++ Incremento.cpp -o inc
osboxes@osboxes:~/Desktop$ ./inc
5
Il valore di i incrementato è:3
Il valore dell'argomento decrementato è:99
osboxes@osboxes:~/Desktop$
```

- ❑ Le classi di immagazzinamento indicano al compilatore le modalità di memorizzazione di una variabile.
- ❑ Il C++ definisce le seguenti 5 classi di immagazzinamento:
 - **auto;**
 - **extern;**
 - **register;**
 - **static;**
 - **mutable.**
- ❑ Lo specificatore **mutable** si applica solo agli oggetti **class** che tratteremo in seguito.

- ❑ Lo specificatore **auto** dichiara una variabile come **automatica** o **dinamica**.
- ❑ Una variabile **automatica** cessa di esistere non appena il programma esce dalla funzione in cui la variabile è stata dichiarata.
- ❑ Tutte le variabili locali a una funzione sono per default **automatiche**. All'atto della chiamata alla funzione viene automaticamente allocato spazio in memoria per ospitare le sue variabili. Tale spazio verrà automaticamente deallocato all'uscita dalla funzione.

- ❑ Le variabili di tipo **static** sono variabili permanenti all'interno della funzione o programma in cui sono state dichiarate.
- ❑ È possibile avere variabili **statiche** sia globali che locali e il modificatore **static** influisce in maniera differente.
- ❑ Una variabile **locale statica** viene allocata nella memoria permanente. Ciò consente di conservare il valore della variabile fra due chiamate alla funzione (non vero per una variabile locale).
- ❑ La differenza fra una variabile locale statica e una globale statica è che nel primo caso è nota solo al blocco in cui è dichiarata, nel secondo a tutto il programma.

- ❑ Quando il file di un programma aumenta di dimensioni, il tempo necessario alla sua compilazione può diventare molto lungo.
- ❑ In questi casi è buona norma dividere il programma in file multipli o **moduli** e ogni modulo dovrà conoscere il nome e il tipo di tutte le variabili globali utilizzate dal programma.
- ❑ C++ non consente di avere una copia di ogni variabile globale in ciascuno dei **moduli** che compongono il programma.
- ❑ La soluzione è dichiarare tutte le variabili globali in un unico file e utilizzare la dichiarazione **extern** per gli altri file.

- ❑ Il modificatore **register** comunica al compilatore di memorizzare la corrispondente variabile in modo da garantirne il più rapido tempo di accesso possibile.
- ❑ Questo si traduce nella memorizzare delle variabili nei registri della CPU o della memoria cache e non nella memoria RAM.
- ❑ Register è una richiesta che il compilatore può ignorare in quanto il numero di registri ad accesso rapido è limitato e dipende dall'ambiente utilizzato.
- ❑ In generale, quanto più frequente è l'accesso a una variabile, tanto maggiore sarà il beneficio ottimizzandola come variabile **register**.

- ☐ La memorizzazione **automatica** è un mezzo per risparmiare l'utilizzo della memoria (le variabili sono create all'inizio del blocco e distrutte all'uscita).
- ☐ Le variabili utilizzate con una certa intensità, come i contatori e i totali, sono delle ottime candidate ad essere di tipo **register**.
- ☐ Le variabili locali sono **automatiche** per default.
- ☐ Utilizzare più di uno specificatore per una stessa variabile da luogo ad un errore di sintassi.
- ☐ Le funzioni hanno visibilità **register** di default.

- ❑ Il sovraccaricamento di una funzione permette di utilizzare più funzioni con lo stesso nome , **ma con almeno un argomento di tipo diverso e/o con un diverso numero di argomenti.**
- ❑ Supponiamo di voler calcolare l'area di un quadrato. Una possibile implementazione è la seguente:

```
int areaQuadrato(int x)
{
    return x*x;
}
```

- ❑ Una implementazione del genere sarà in grado di valutare l'area del quadrato **solo** per lato di tipo intero.

- ❑ Se si vuole implementare una funzione che risolva lo stesso problema, ma per lato di tipo double, basterà definire una nuova funzione che conserva il nome della precedente con tipo di ritorno e tipo del parametro in ingresso double.

```
double areaQuadrato(double x)
{
    return x*x;
}
```

- ❑ Sarà il compilatore, in funzione del tipo di parametro passato alla funzione, a invocare la funzione corretta.
- ❑ Il sovraccaricamento delle funzioni è un tipo di **polimorfismo**.

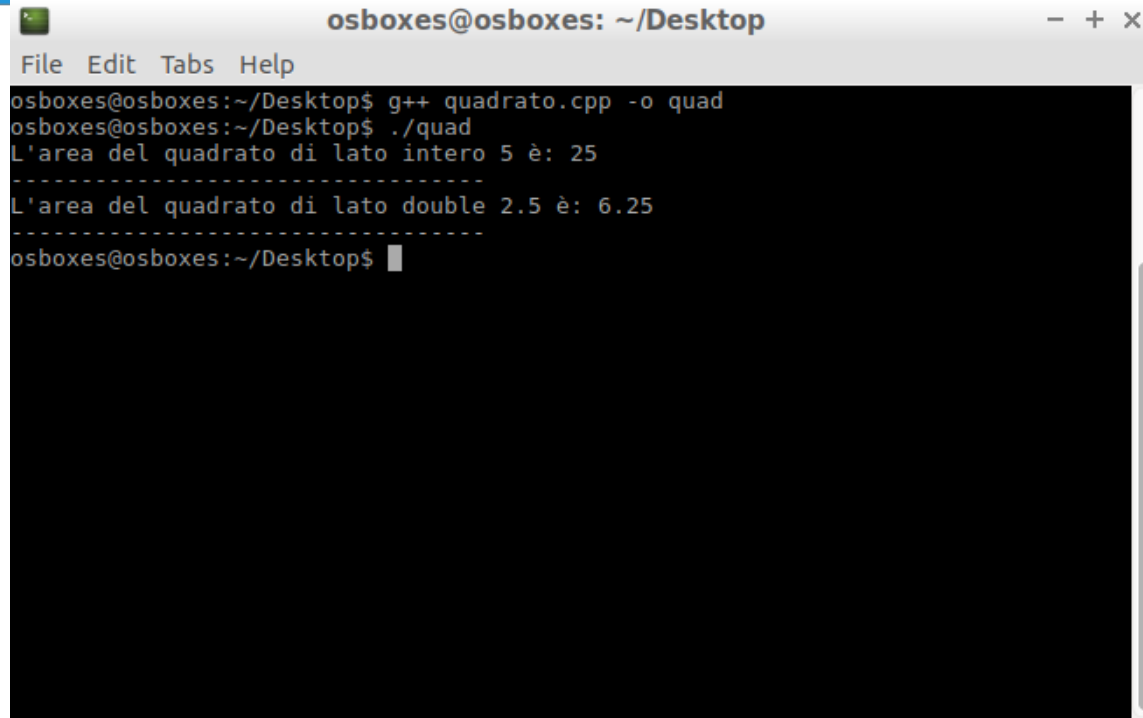
```
#include<iostream>

using namespace std;

int areaQuadrato(int x)
{
    return x*x;
}

double areaQuadrato(double x)
{
    return x*x;
}

int main()
{
    int x=5;
    double y=2.5;
    cout<<"L'area del quadrato di lato intero 5 è: "<<areaQuadrato(x)<<endl;
    cout<<"-----"<<endl;
    cout<<"L'area del quadrato di lato double 2.5 è: "<<areaQuadrato(y)<<endl;
    cout<<"-----"<<endl;
    return 0;
}
```



osboxes@osboxes: ~/Desktop

File Edit Tabs Help

```
osboxes@osboxes:~/Desktop$ g++ quadrato.cpp -o quad
osboxes@osboxes:~/Desktop$ ./quad
L'area del quadrato di lato intero 5 è: 25
-----
L'area del quadrato di lato double 2.5 è: 6.25
-----
osboxes@osboxes:~/Desktop$
```


- ❑ Nessun programma significativo è scritto utilizzando solo le istruzioni native del linguaggio di programmazione.
- ❑ I linguaggi di programmazione sono corredati da librerie che offrono funzionalità già implementate efficientemente.
- ❑ La **Standard Template Library** (STL) implementa utilissimi tipi quali string, vector, list e map ed una serie di funzionalità per usarli.
- ❑ Le funzioni di libreria sono raccolte in gruppi e definite in uno stesso **header file**.

- ❑ Per utilizzare qualsiasi funzionalità della Standard Library occorre includere l'appropriato **header file** e specificare l'utilizzo del **namespace std** all'inizio del programma.

```
#include<iostream>
#include<string>

using namespace std;

int main()
{
    string s="Hello World";
    cout<<s<<endl;
    return 0;
}
```

□ Di seguito alcune delle funzioni di libreria messe a disposizione dal linguaggio C++:

- Funzioni di Input/Output (**#include <iostream>**);
- Funzioni di manipolazione stringhe (**#include <string>**);
- Funzioni matematiche (**#include <cmath>**);
- Funzioni di manipolazione ora e data(**#include <ctime>**);
- Funzioni gestione liste (**#include <list>**);
- Funzioni gestione code(**#include <queue>**);
-



Contact Info