



UNIVERSITÀ  
degli STUDI  
di CATANIA

# Static, const, friend

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. Qualificatore `static`
2. Modificatore `const`
3. Metodi `friend`

## Qualificatore static

---

# Variabili statiche

**Variabile statica** (o “variabile di classe”): variabile comune a tutti gli oggetti della classe.

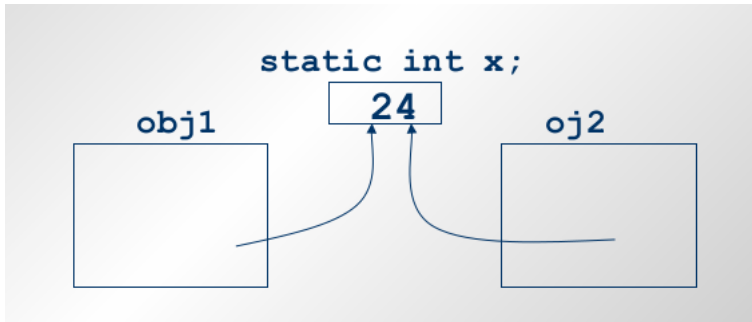
Si usa il qualificatore di persistenza **static**.

Ogni oggetto creato da quella classe avrà **accesso allo stesso blocco di memoria per una variabile static**.



Di conseguenza variabile `static` permette di **condividere informazioni** tra gli **oggetti di una classe**.

- Le modifiche al campo static saranno visibili a tutti gli altri oggetti

# Variabili statiche



# Variabili statiche

```
1  class ClasseX{  
2      static int counter;   
3      int x,y;  
4  
5      public:  
6          ClasseX(int x, int y){  
7              this->x = x;  
8              this->y = y;  
9              counter++;   
10         }  
11     }
```

# Variabili statiche

Inizializzazione “oggetti” static va fatta **all'esterno**.

```
1  class ClasseX{
2      static int counter;
3      public:
4          ClasseX(int x, int y){
5              //..
6              counter++;
7          }
8  }
9
10 int ClasseX::counter = 0;
11
12 int main(){
13     //...
14 }
```

# Variabili statiche

```
1  class ClasseX{  
2      static int counter = 0; ←  
3      int x,y;  
4  
5      //...
```

Istruzione alla linea 1 è ERRORE. Il campo static va inizializzato esternamente. Il **compilatore** darà il seg messaggio:

error: ISO C++ forbids in-class initialization of non-const static member



# Variabili statiche

```
1  //int ClasseX::counter = 0;
2  class ClasseX{
3      //...
4  }
5  int main(){
6      //...
7  }
```

Se istruzione alla linea 1 **viene omessa**, allora **linker** non potrà generare un programma funzionante.

undefined reference to "ClasseX::counter"

### Attributo di istanza:

- per ogni istanza di oggetto, una copia separata in memoria dello attributo di istanza
- ognuno degli oggetti gestirà le **modifiche della propria copia in memoria**
- allocazione in memoria al momento della **creazione dell'oggetto**

### Attributo di classe (o statico):

- tutte le istanze della classe (oggetti) condividono una unica copia in memoria
- eventuale modifica si riflette su tutti gli altri oggetti
- va inizializzata fuori dallo scope della classe

Esempi svolti

25\_01.cpp

## Variabile locale static

Variabile locale al metodo può essere dichiarata `static`.

```
1 void f(){  
2     ↪ static int a = 0;  
3     if(a++>10)  
4         cout << "No more activity.." << endl;  
5     else{  
6         //do something..  
7     }  
8 }
```

## Variabile locale `static`

**Creazione ed inizializzazione** contestuali, una sola volta, alla prima esecuzione del metodo.

Variabile sarà **distrutta alla fine del programma**.

Variabile **conservata in zona di memoria per allocazione statica**.

```
1 void f(){  
2     static int a = 0;  
3     // ...  
4 }  
5 }
```

Esempi svolti

25\_02.cpp

Una funzione membro static è rappresenta un **comportamento indipendente dallo stato delle istanze** di quella classe.

Una funzione membro static **può manipolare solo membri statici**:

- *invocare altre funzioni membro static.*
- *manipolare attributi static.*



Sintassi **invocazione** fa uso di operatore risoluzione di scope, come per variabili static.

```
<nome_classe>::<nome_metodo>(...);
```

Ma anche (non consigliata perchè poco espressiva..)

```
<nome_variabile_istanza>.<nome_metodo>(...);
```

# Metodi static

```
1  class ClasseX{  
2      static int counter;  
3      //...  
4      static int getCounter(){  
5          return counter;  
6      }  
7  }
```

```
8  
9  int main(){  
10     ClasseX istanza(1,2);  
11  
12     cout << ClasseX::getCounter();  
13     cout << istanza.getCounter();  
14 }
```

esempi svolti

25\_03.cpp


## Modificatore const

---

# Modificatore “const”

## Funzione membro const.

```
1  class ClasseX {  
2      int x;  
3  
4      int getX() const {  
5          return this->x;  
6      }  
7  }
```



Non può modificare lo stato dell'oggetto:

- non può modificare variabili membro
- non può invocare metodi non const

esempio svolto

25\_04.cpp

**Metodi** friend

---

Per una **funzione membro** valgono le segg. regole:

- P1** la funzione **ha accesso alle proprietà dell'oggetto**, quindi della sua parte “privata”;
- P2** la funzione “risiede” **nello scope della classe**;
- P3** la funzione va **invocata mediante una istanza dell'oggetto**.



Per una funzione **membro static** valgono solo le **P1** e **P2**:

**P1** la funzione **ha accesso alle proprietà dell'oggetto**, quindi della sua parte “privata”;

**P2** la funzione “risiede” **nello scope della classe**;

~~**P3** la funzione va invocata mediante una istanza dell'oggetto.~~

Per una funzione dichiarata `friend`, vale solo la **P1**:

**P1** la funzione **ha accesso alle proprietà dell'oggetto**, quindi della sua parte “privata”;

**P2** ~~la funzione “risiede” nello scope della classe;~~

**P3** ~~la funzione va invocata mediante una istanza dell'oggetto.~~

Dunque, una funzione dichiarata friend **non risiede nello scope di una specifica classe.**

**Ad essa è garantito lo accesso a tutti i membri delle classi** per le quali è stata dichiarata friend.

Si consideri una classe `Matrix` ed una classe `Vector`, ed una operazione **moltiplicazione** tra matrice e vettore.

Considerazioni:

- sarebbe opportuno **garantire accesso alla rappresentazione dei dati** di `Vector` e `Matrix` al metodo che implementa operazione di moltiplicazione;
- MA un metodo **non può essere membro di entrambe** le classi contemporaneamente;
- d'altro canto si vuole mantenere **information hiding**, ovvero **nascondere** la rappresentazione dei dati agli utenti;

## Metodi friend

Un metodo che non risiede nello scope di una classe, può essere dichiarato friend di una o più classi:

```
1  class Matrix{  
2      //...  
3  friend Vector &mult(const Matrix &m, const Vector &v);  
4  }  
5  
6  class Vector{  
7      //...  
8  friend Vector &mult(const Matrix &m, const Vector &v);  
9  }
```

Metodo mult definito come metodo/funzione **globale**, quindi non risiede nello scope di alcuna delle classi Vector e Matrix:

```
1  Vector &mult(const Matrix &m, const Vector &v){  
2      //... moltiplicazione matrice vettore..  
3      //accesso ad elementi private di m e v..  
4  }
```

Corpo del metodo mult(): accesso ad elementi private di m e v.

**Due parametri reference:** Matrix e Vector.

MA funzione friend per una classe potrebbe anche essere un membro di un'altra classe:

```
1  class Matrix{
2      //...
3      friend Vector &Vector::mult(const Matrix &m);
4  }
5
6  class Vector{
7      //...
8      Vector &mult(const Matrix &m);
9  }
```

NB alla linea 9: un solo parametro reference di tipo Matrix.

### Esempio svolto

A25\_05.cpp - funzione globale friend per una classe

A25\_06.cpp - funzione globale friend per due classi

A25\_07.cpp - funzione membro di una classe friend di un'altra classe



### Remark

Una funzione dichiarata friend all'interno di una classe X:

- **non è membro** della classe X
- MA **può accedere** a TUTTI i membri (pubblici e non pubblici) della classe X;

Lo scope di una funzione friend:

- è **globale** se la funzione non appartiene allo scope di alcuna classe (**non è funzione membro** di alcuna classe)
- se la funzione friend è membro di un'altra classe Y, (il suo scope) sarà quello **della sezione in cui è stata dichiarata e definita nella classe di Y**

Con la seg. dichiarazione:

```
1  class X{  
2      friend class Y;  
3  }
```

**Ogni funzione membro della classe Y avrà accesso a TUTTI i membri della classe X.** Esempio:

```
1  class Matrix{  
2      friend class Vector;  
3  }
```

### Funzione membro friend

```
1  class Matrix{  
2      friend Vector& Vector::mult(const Matrix &m);  
3  }
```

### VS Classe friend

```
1  class Matrix{  
2      friend class Vector;  
3  }
```

Osservazione: nel secondo caso **non si conosce a priori la lista (a la natura) dei metodi che hanno accesso “full” ai membri della classe Matrix (!!)**.

### Esempio svolto

A25\_16.cpp - classe friend

A25\_17.cpp - classi reciprocamente friend

Uso della keyword `friend` “viola” le regole dello incapsulamento.

Da usare con parsimonia:

- **overloading operatori binari** ( caso più comune, si vedrà in seguito..)
- considerazioni di efficienza (limitare uso di funzioni “getter”)
- relazioni speciali tra le classi..

FINE