



UNIVERSITÀ
degli STUDI
di CATANIA

DIPARTIMENTO DI
MATEMATICA E INFORMATICA

OOP in C++

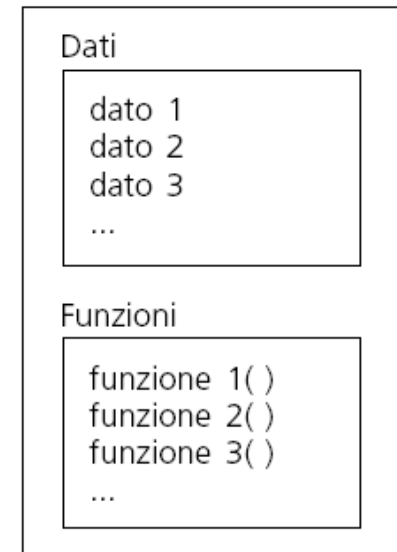
Alessandro Ortis

alessandro.ortis@unict.it

www.dmi.unict.it/ortis/

Le classi

- Classe: insieme di oggetti che condividono una struttura ed un comportamento
- Contiene la specifica dei dati che descrivono l'oggetto che ne fa parte, insieme alla descrizione delle azioni che l'oggetto stesso è capace di eseguire
- in C++ questi dati si denominano *attributi* o *variabili*, mentre le azioni si dicono *funzioni membro* o *metodi*
- Le classi definiscono ***tipi di dato personalizzati*** in funzione dei problemi da risolvere,
 - ciò facilita scrittura e comprensione delle applicazioni;
- Possono ***separare l'interfaccia dall'implementazione***;
 - solo il programmatore della classe conoscerà i dettagli implementativi,
 - l'utilizzatore deve soltanto conoscere l'interfaccia



Definizione di una classe

- Due parti:
 - *dichiarazione*: descrive i dati e l'interfaccia (cioè le "funzioni membro", anche dette "metodi")
 - *definizioni dei metodi*: descrive l'implementazione delle funzioni membro

```
class NomeClasse          // Identificatore valido
{
    dichiarazioni dei dati      // attributi
    definizione delle funzioni  // metodi
};
```

- Attributi: variabili semplici (interi, strutture, arrays, float, ecc.)
- Metodi: funzioni semplici che operano sugli attributi (*dati*)

Specificatori di accesso

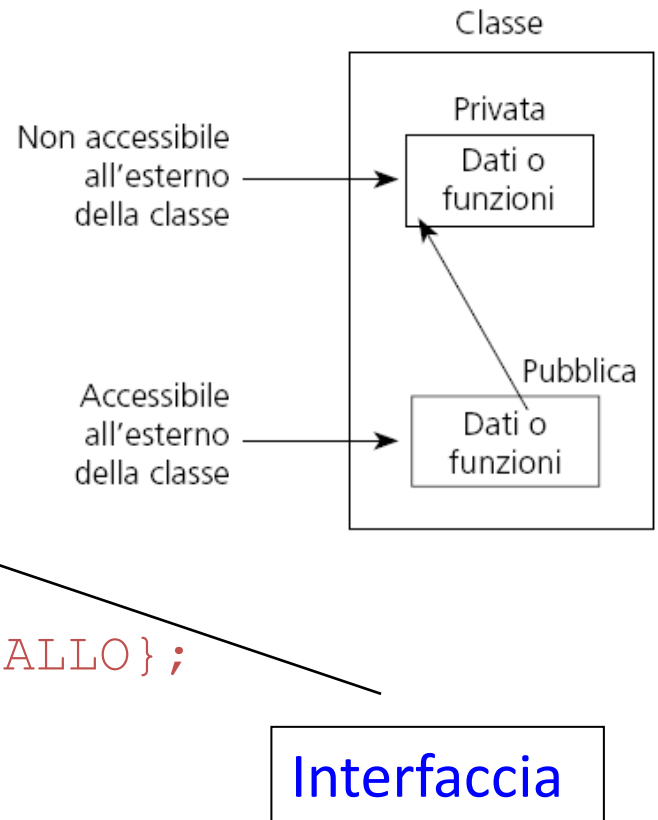
- Per default, i membri di una classe sono nascosti all'esterno, cioè, i suoi dati ed i suoi metodi sono *privati*
- E' possibile controllare la *visibilità* esterna mediante ***specificatori d'accesso***:
 - la sezione `public` contiene membri a cui si può accedere dall'esterno della classe
 - la sezione `private` contiene membri ai quali si può accedere solo dall'interno
 - ai membri che seguono lo specificatore `protected` si può accedere anche da metodi di classi *derivate* della stessa

```
class NomeClasse
{
    public:
        Sezione pubblica    // dichiarazione di membri pubblici
    protected:
        Sezione protetta    // dichiarazione di membri protetti
    private:
        Sezione privata      // dichiarazione di membri privati
};
```

Information hiding

- Questa caratteristica della classe si chiama *occultamento di dati (information hiding)* ed è una proprietà dell'OOP
- limita molto gli errori rispetto alla programmazione strutturata

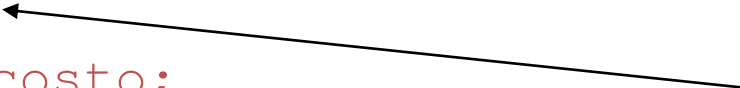
```
class Semaforo
{
    public:
        void cambiareColore();
        //...
    private:
        enum Colore {VERDE, ROSSO, GIALLO};
        Colore c;
};
```



Regole pratiche

- le dichiarazioni dei metodi (i.e., intestazioni delle funzioni), normalmente, si collocano nella sezione pubblica
- le dichiarazioni dei dati (attributi), normalmente, si mettono nella sezione privata
- E' indifferente collocare prima la sezione pubblica o quella privata;
 - Meglio collocare la sezione pubblica prima per mettere in evidenza le operazioni che fanno parte dell'interfaccia utente pubblica
- `public` e `private` seguite da due punti, segnalano l'inizio delle rispettive sezioni pubbliche e private;
 - una classe può avere varie sezioni pubbliche e private
- L'interfaccia deve essere pubblica
 - Altrimenti non può essere invocata dal programma!

```
class Prova
{
    private:
        float costo;
        char nome[20];
    public:
        void calcolare(int);
};
```



`private` non è necessario
ma è utile per evidenziare
l'occultamento

Per usare una classe bisogna sapere

.Nome

- Tipicamente definito in un header file (con lo stesso nome della classe)

.Dove è definita

.Che operazioni supporta

- Nelle definizioni delle classi si collocano (tipicamente) solo le intestazioni dei metodi
- Le definizioni dei metodi stanno in un file di *implementazione* (estensione .cpp)

Oggetti

- definita una classe, possono essere generate *istanze* della classe, cioè *oggetti*

```
nome_classe    identificatore ;
```

```
Rettangolo r;  
Semaforo s;
```

- un oggetto sta alla sua classe come una variabile al suo tipo
- quello che nelle struct era l'operatore di accesso al campo (.), qui diventa l'operatore *di accesso* al membro

```
Punto p;  
p.setX(100);  
cout << " coordinata x è " << p.getX();
```


Oggetti

- Gli oggetti (come le strutture) possono essere copiati
- C++ fa una copia bit a bit di tutti i membri
- Tutti i membri presenti nell'area dati dell'oggetto originale vengono copiati nell'oggetto destinatario

```
Rettangolo r1;
```

```
...
```

```
Rettangolo r2;
```

```
r2=r1;
```

Dati membro

- possono essere di qualunque tipo valido, tranne il tipo della classe che si sta definendo.
- Un dato membro (attributo) ha un nome (o identificatore) e un tipo; mantiene un valore di uno specifico tipo (tipo base o altra classe).
- Convenzione sui nomi: si usa la notazione a cammello. I nomi delle variabili sono parole singole (sostantivi), oppure parole composte unendo più parole tra loro, ma lasciando le iniziali maiuscole.

```
double coeffAngolare;  
int anniPersona;  
...
```

Funzioni membro

- possono essere sia dichiarate che definite all'interno delle classi; la definizione consiste di quattro parti:
 - il tipo restituito dalla funzione
 - il nome della funzione
 - la lista dei parametri formali (eventualmente vuota) separati da virgole
 - il corpo della funzione racchiuso tra parentesi graffe
- le tre prime parti formano il **prototipo** (o signature) della funzione che *deve essere definito* dentro la classe,
 - il corpo della funzione può essere definito altrove

```
class Quadrato {  
public:  
    double calcola_area(); // dichiarazione prototipo (definito altrove)  
    double calcola_perimetro () // definizione  
        {return lato*4; } // funzione  
private: // membri privati  
    double lato;  
};  
  
double Quadrato::calcola_area(){ return lato*lato;}
```

Funzioni membro

- La definizione di funzioni dichiarate in una classe deve contenere il riferimento alla classe

```
tipo_restituito Nome_Classe :: Nome funzione  
(lista parametri)  
{  
    corpo della funzione  
}
```



```
double Quadrato::calcola_area() {  
    return lato*lato;  
}
```

Chiamate a funzioni membro

- i metodi di una classe s'invocano così come si accede ai dati di un oggetto, tramite l'operatore punto (.) con la seguente sintassi:

nomeOggetto.nomeFunzione (valori dei parametri)

```
class Demo
{
private:
    // ...
public:
    void funz1 (int P1)
        {...}
    void funz2 (int P2)
        {...}
};
Demo d1, d2;           // definizione degli oggetti d1 e d2
...
d1.funz1(2005);
d2.funz1(2010);
```

**Alcuni linguaggi chiamano messaggi
le invocazioni a funzioni membro**

Tipi di funzioni membro

- Costruttori e distruttori
 - Invocati automaticamente alla creazione e alla distruzione di oggetti
- Selettori
 - Restituiscono valori di membri dato
- Modificatori
 - Modificano i valori di membri dato
- Operatori
 - definiscono operatori standard in C++
- Iteratori
 - elaborano collezioni di oggetti (es. array)

Funzioni *inline* e *offline*

- i metodi definiti nella classe sono funzioni in linea; per funzioni grandi è preferibile codificare nella classe solo il prototipo della funzione
- nella definizione *fuori linea* della funzione bisogna premettere il nome della classe e l'*operatore di risoluzione di visibilità* ::;

```
class Punto {  
public:  
    void fissareX(int valx);  
private:  
    int x;  
    int y;  
};  
  
void Punto::fissareX(int valx)  
{  
    x = valx;  
}
```

Nella dichiarazione il nome dei parametri può essere omesso.

Header files ed intestazioni di classi

- il codice sorgente di una classe si colloca normalmente in un file indipendente con lo stesso nome della classe ed estensione .cpp
- le dichiarazioni si collocano normalmente in header files indipendenti da quelli che contengono le implementazioni dei metodi

Costruttori

- Può essere conveniente che un oggetto si possa auto-inizializzare all'atto della sua creazione, senza dover effettuare una successiva chiamata ad una sua qualche funzione membro
- un *costruttore* è un metodo di una classe che viene automaticamente eseguito all'atto della creazione di un oggetto di quella classe
- ha lo stesso nome della propria classe e può avere qualunque numero di parametri ma non restituisce alcun valore

```
class Rettangolo
{
    private:
        int base;
        int altezza;

    public:
        Rettangolo(int base, int altezza); // Costruttore
        // definizioni di altre funzioni membro
};
```

Definizione oggetto con costruttore

- quando si definisce un oggetto, si passano i valori dei parametri al costruttore utilizzando la sintassi di una normale chiamata di funzione:

```
Rettangolo rect(25, 75); // rect è ISTANZA di Rettangolo
```

```
Rettangolo* nr = new Rettangolo(25, 75); // nr punta  
// una nuova ISTANZA di Rettangolo
```

- un costruttore senza parametri si chiama *costruttore di default*;
 - inizializza i membri dato assegnandogli valori di default
- C++ crea automaticamente un costruttore di default quando non vi sono altri costruttori, esso non inizializza i membri dato della classe a valori predefiniti
- un *costruttore di copia* è creato automaticamente dal compilatore quando:
 - si passa un oggetto per valore ad una funzione (si costruisce una copia locale dell'oggetto)
 - si definisce un oggetto inizializzandolo ad un oggetto dello stesso tipo

Definizione oggetto con costruttore

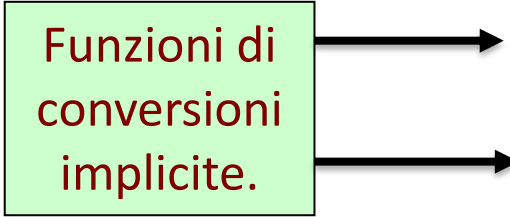
Esistono diversi modi per inizializzare gli attributi di un oggetto.

```
class MyClass{
    public:
        // Costruttore di default
        MyClass(){}
        // lista di inizializzazione
        MyClass(int a, char b): x(a), c(b) {}
        MyClass(int a): y(a*2) {}
        // costruttore con delega
        MyClass(char b): MyClass(10, b){}
        void printAll();
    private:
        int x = 18; // inizializzazione
        char c {'M'}; // iniz. Uniforme
        const int y = 0;
};
```

Definizione oggetto con costruttore

Esistono diversi modi per inizializzare gli attributi di un oggetto.

```
class MyClass{
    public:
        // Costruttore di default
        MyClass(){}
        // lista di inizializzazione
        MyClass(int a, char b): x(a), c(b) {}
        MyClass(int a): y(a*2) {}
        // costruttore con delega
        MyClass(char b): MyClass(10, b){}
        void printAll();
    private:
        int x = 18; // inizializzazione
        char c {'M'}; // iniz. Uniforme
        const int y = 0;
};
```



Funzioni di
conversioni
implicite.

Qualificatore `const`

Il qualificatore `const` indica che un tipo è costante ed è utilizzabile in diversi ambiti, allo scopo di evitare la modifica di informazioni marcate come costanti.

```
const double pi = 3.14;
```

Puntatori e qualificatore `const`

```
int a = 22;  
  
int* p1 = &a; // niente e' costante  
const int* p2 = &a; // dato costante  
int* const p3 = &a; // puntatore costante  
const int* const p4 = &a; // tutto costante
```

Parametri di funzioni `const`

```
void funzione(const int& a) {  
    // qualcosa che non modifica a  
}
```

Qualificatore `const`

Le funzioni membro `const` non possono modificare gli attributi dell'oggetto di appartenenza.

```
class MyClass{
    int x;
    ...
    int myFunction() const;
};

int MyClass::myFunction() const {
    // non posso modificare x
}
```

Valori di ritorno `const`

```
class MyClass{
    int x= 100;
    public:
        int const & goodGetX(){ return x;}
};
```

Distruttore

- si può definire anche una funzione membro speciale nota come *distruttore*, che viene chiamata automaticamente quando si distrugge un oggetto
- il distruttore ha lo stesso nome della sua classe preceduto dal carattere ~
- neanche il distruttore ha tipo di ritorno ma, al contrario del costruttore, non accetta parametri e *non* ve ne può essere più d'uno

```
class Demo
{
private:
    int dati;
public:
    Demo()    {dati = 0;}           // costruttore
    ~Demo()   {}                   // distruttore
};
```

- serve normalmente per liberare la memoria assegnata dal costruttore
- se non si dichiara esplicitamente un distruttore, C++ ne crea automaticamente uno vuoto

Membri statici

Un membro statico è associato alla classe anziché con un oggetto (istanza di una classe). Questo significa che ne esiste una copia unica per tutte le istanze di quella classe nel caso di un attributo statico.

Nel caso di un metodo statico, significa che è possibile invocare quel metodo senza aver istanziato alcun oggetto.

```
class Point {  
public:  
    static int n;  
  
    Point(): x(0), y(0) {n++;}           // costruttore  
    ~Point() { n--; }                   // distruttore  
  
    static float distance(Point a, Point b) { //... }  
  
private:  
    int x,y;  
};
```


Friend e incapsulamento

A volte può essere utile consentire l'accesso a membri privati anche a funzioni o metodi di altre classi. Questo può essere utile nell'overloading di operatori di input/output.

```
class Point {
public:
    // dichiarazione di amicizia
    friend bool operator==(Point a, Point b);
private:
    int x,y;
};

bool operator==(Point a, Point b){
    if ( (a.x != b.x) || (a.y) != (b.y) ) return false;
    else return true;
}

int main(){ Point p,q;
if (p == q) cout << 'p e q sono uguali' << endl; }
```

Friend e incapsulamento

```
bool operator==(Point a, Point b){  
    if ( (a.x != b.x) || (a.y) != (b.y) ) return false;  
    else return true;  
}
```

Alternativa senza friend

```
class Point {  
public:  
    // dichiarazione come funzione membro  
    bool operator==(Point b);  
private:  
    int x,y;  
};  
  
bool Point::operator==(Point b){  
    if ( (this->x != b.x) || (this->y) != (b.y) ) return  
false;  
    else return true;  
}
```

Friend e incapsulamento

La funzione `operator==` non è un membro della classe `Point`, eppure può accedere ai suoi membri privati. L'uso del qualificatore `friend` consente di uniformare l'interfaccia di I/O della nostra classe allo standard senza violare il principio di incapsulamento. Infatti, anche se le funzioni `friend` non sono membri di classe, la loro dichiarazione deve apparire nella definizione della classe. Questo ci consente di rimanere in totale controllo dell'interfaccia, rendendo impossibili alterazioni dall'esterno mediante la definizione di funzioni `friend`.

```
class Point {  
    ... // dichiarazione di amicizia  
        friend bool operator==(Point a, Point b);  
    ...  
};
```

```
bool operator==(Point a, Point b){  
    if ( (a.x != b.x) || (a.y) != (b.y) ) return false;  
    else return true;    }
```

Sovraccaricamento di metodi e operatori

- anche le funzioni membro possono essere sovraccaricate, ma soltanto nella loro propria classe
- Seguono le stesse regole utilizzate per sovraccaricare funzioni ordinarie:
 - due funzioni membro sovraccaricate non possono avere lo stesso numero e tipo di parametri
 - l'*overloading* permette di utilizzare uno stesso nome per più metodi che si distingueranno solo per i parametri passati all'atto della chiamata

```
class Prodotto
{
public:
    int prodotto (int m, int n);           // metodo 1
    int prodotto (int m, int p, int q);    // metodo 2
    int prodotto (float m, float n);       // metodo 3
    int prodotto (float m, float n, float p); // metodo 4
}
```

Sovraccaricamento di metodi e operatori

```
class Array{
public:
    Array(int size=10); // costruttore
    // l'operatore == non puo' modificare nulla
    bool operator ==(const Array& right) const;
    int& operator[] (int index);
    ...
    int size;          int* data;
}

bool Array::operator==(const Array& right) const{
    if(size!= right.size) return false;
    for(int i=0;i<size; i++)
        if(data[i] != right.data[i])
            return false;
    return true;
}

int& Array::operator[] (int index){
    return data[index];
}
```

Sovraccaricamento di metodi e operatori

```
class Frazione{

friend Frazione operator+(const Frazione& f1, const
Frazione& f2);
friend ostream& operator<<(ostream& s, const Frazione& f);
...
}

Frazione operator+(const Frazione& f1, const Frazione& f2){
    Frazione r;
    r.num = (f1.num*f2.den) + (f2.num*f1.den);
    r.den = f1.den * f2.den;
    return r;
}

ostream& operator<< (ostream& s, const Frazione& f){
    s << f.num << '/' << f.den;
    return s;
}
```

Gestione delle eccezioni

Per implementare la gestione delle eccezioni in C++, si usano le keyword **try** **throw** e **catch**.

throw segnala una condizione eccezionale in un blocco **try**.

È possibile utilizzare un oggetto di qualsiasi tipo come operando di un'espressione **throw**. Questo oggetto viene utilizzato per comunicare informazioni sull'errore.

Nella maggior parte dei casi, è consigliabile usare la `std::exception` classe o una delle classi derivate definite nella libreria standard. In alternativa è possibile derivare la propria classe di eccezione da `std::exception`.

Gestione delle eccezioni

```
try {  
    throw 'a';  
}  
catch (int x) {  
    cout << " catturo " << x;  
}  
catch (...) {  
    cout << "Default Exception\n";  
}
```

Output:

>> *Default Exception*

Gestione delle eccezioni

Le eccezioni vengono intercettate per riferimento const nell'istruzione catch. Questo non è obbligatorio ma consigliabile.

```
try{
    // codice che potrebbe generare l'errore
}
catch (const exception& ex){
    // gestione dell'errore della classe exception
}
catch (const string& ex){
    // gestione dell'errore che passa una stringa
}
catch (...){
    // gestione di qualsiasi tipo di errore
}
```

Gestione delle eccezioni

In molti casi possiamo prevenire gli errori a runtime dei nostri programmi quando si verificano situazioni anomale relativamente alla logica del problema.

```
class Tempo {
private:
    int ore;        // 0 - 23
    int minuti;     // 0 - 59
    int secondi;    // 0 - 59
    ...
void setOre(int h) {
    if (h >= 0 && h <= 23)
        ore = h;
    else {
        cout << "Errore: valori di ore validi 0-23." << endl;
        exit(1); // Termina il programma
    }
};
```

Gestione delle eccezioni

Il metodo `setOre()` assegna `h` all'ora se `h` è un'ora valida. Altrimenti, usiamo la funzione di gestione delle eccezioni C++ per lanciare una eccezione del tipo `invalid_argument`.

```
#include <stdexcept> // Necessario per gestire le eccezioni

class Tempo {
private:
    int ore;        // 0 - 23
    int minuti;     // 0 - 59
    int secondi;    // 0 - 59
    ...
    void setOre(int h) {
        if (h >= 0 && h <= 23)
            hour = h;
        else
            throw invalid_argument("Errore: valori di ore validi
0-23.");
    }
}
```

Gestione delle eccezioni

Ciò consente al chiamante di catturare l'eccezione e di elaborare correttamente la condizione anomala.

```
int main() {  
  
    Tempo t;  
  
    try {  
        t.setOre(100);  
    }  
    catch (invalid_argument& ex) {  
        cout << "Eccezione: " << ex.what() << endl;  
    }  
  
    return 0;  
} // fine del main
```

Esercizio

Definire una classe Rettangolo con i seguenti requisiti funzionali:

- Attributi base e altezza privati, ma accessibili tramite getter/setter.
- Il costruttore prende la base e l'altezza come parametri, in alternativa imposta due valori di default.
- Implementare delle funzioni membro pubbliche per il calcolo di area, perimetro e diagonale
- Implementare una funzione membro che verifica se si tratta di un quadrato

Definire un metodo main() dove vengono istanziati alcuni oggetti Rettangolo per testare le funzionalità della classe.

Hint: utilizzare in maniera opportuna i diversi modi per definire un costruttore ed il qualificatore **const**.

Esercizio

Definire una classe `Punto2D` e utilizzarla per definire una classe `Rettangolo` simile a quella precedente. In particolare la nuova classe `Rettangolo` dovrà:

- avere due attributi `top_left` e `bottom_right` di tipo `Punto2D`
- prevedere i seguenti metodi:
 - `contiene(Punto2D p)` restituisce vero se `p` si trova dentro l'area del rettangolo chiamante
 - `contiene(Rettangolo r)` restituisce vero se `r` è contenuto nel rettangolo chiamante

Inoltre, definire le classi `Punto2D` e `Rettangolo` in modo tale che sia possibile istanziare due oggetti, uno di tipo `Punto2D` e l'altro di tipo `rettangolo` nel seguente modo:

```
Punto2D p = {10,20}  
Rettangolo B({10,20},{50,10});
```

dove:

- `{10,20}` rappresenta un oggetto `Punto2D` di coordinate `x=10 y=20`
- `{50,10}` rappresenta un oggetto `Punto2D` di coordinate `x=50 y=10`

Esercizio

Implementare una classe `Person` e una classe `Pet`, ciascuna persona possiede zero o più animali. Definire in maniera arbitrariamente estesa le due classi, in particolare ridefinire il metodo `operator<<` di entrambe facendo in modo che la classe `Person` sfrutti il metodo `operator<<` della classe `Pet` dei suoi animali.

Esercizio

Implementare un programma che permette a degli utenti di giocare a due tipologie di gioco:

- Carta forbice sasso: massimo 2 giocatori, si vince dopo 5 tentativi.
- Massimo nel lancio di 2 dadi: vince chi effettua il punteggio più alto.

Dopo aver selezionato il gioco, il programma registra i giocatori chiedendo il loro nome. Nel caso di carta forbice sasso possono giocare solo due persone, mentre al secondo gioco possono registrarsi due o più giocatori. Successivamente il programma simula delle partite tra giocatori. Gli esiti devono essere simulati simulando le probabilità di vincita dei giochi. Ad esempio, lanciando un singolo dado ogni esito (numero da 1 a 6) ha probabilità $1/6$ di uscire. Gestire il programma mediante una opportuna classe SalaGiochi.

Esercizio

Implementare una classe Counter. I costruttori devono permettere di definire un contatore che:

- inizia da un valore dato come parametro, oppure da zero
- effettua incrementi/decrementi di una quantità data, oppure di 1

il costruttore di default crea un contatore che inizia da 0 e incrementa/decrementa di 1 ogni volta.

Definire inoltre (almeno) i seguenti membri della classe Counter:

- `public void Increase()` //incrementa il contatore
- `public void Decrease()` //decrementa il contatore
- `public void Increase(int increaseBy)` // accetta solo valori non negativi
- `public void Decrease(int decreaseBy)` // accetta solo valori non negativi