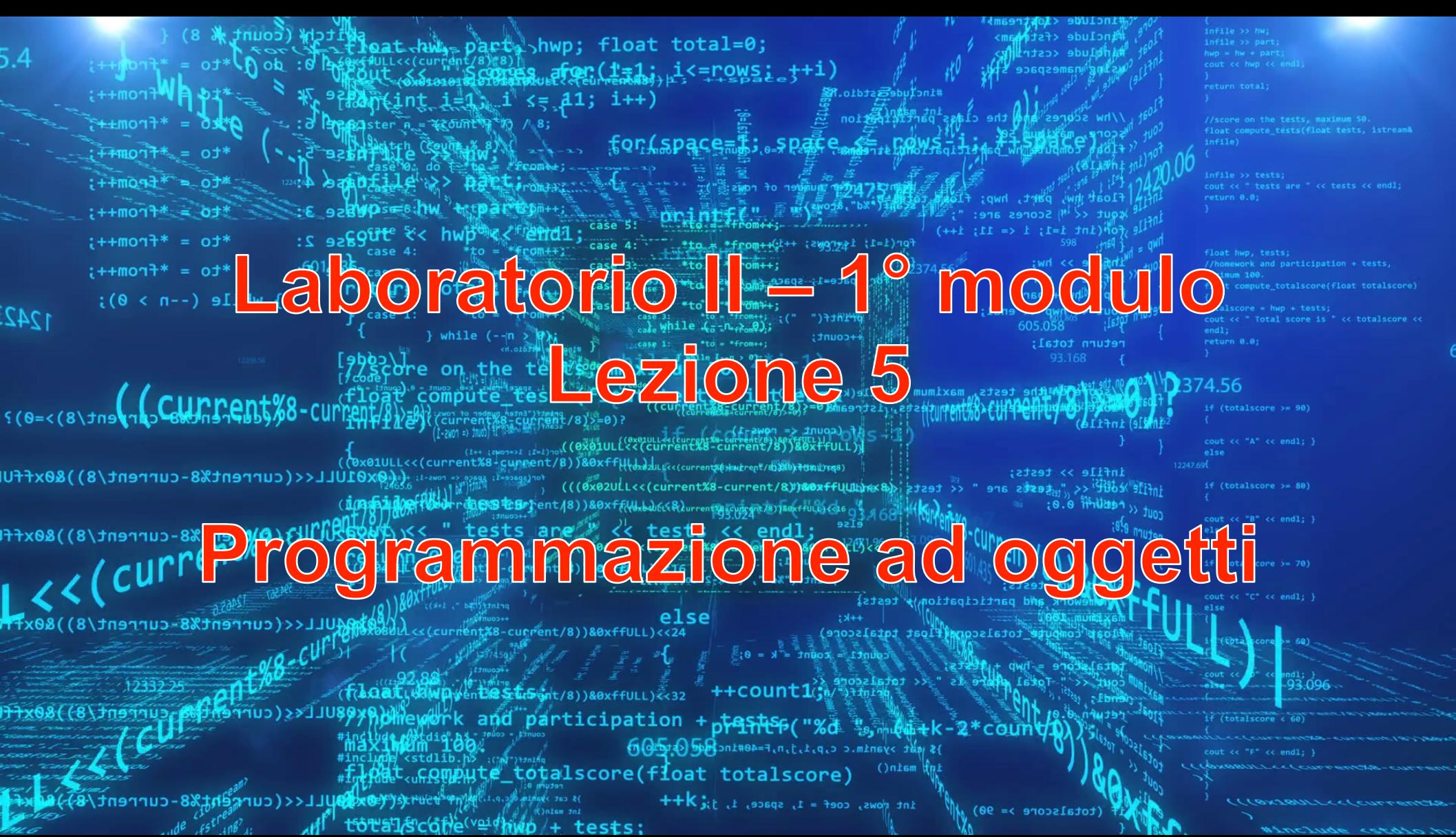


Laboratorio II – 1° modulo

Lezione 5

Programmazione ad oggetti



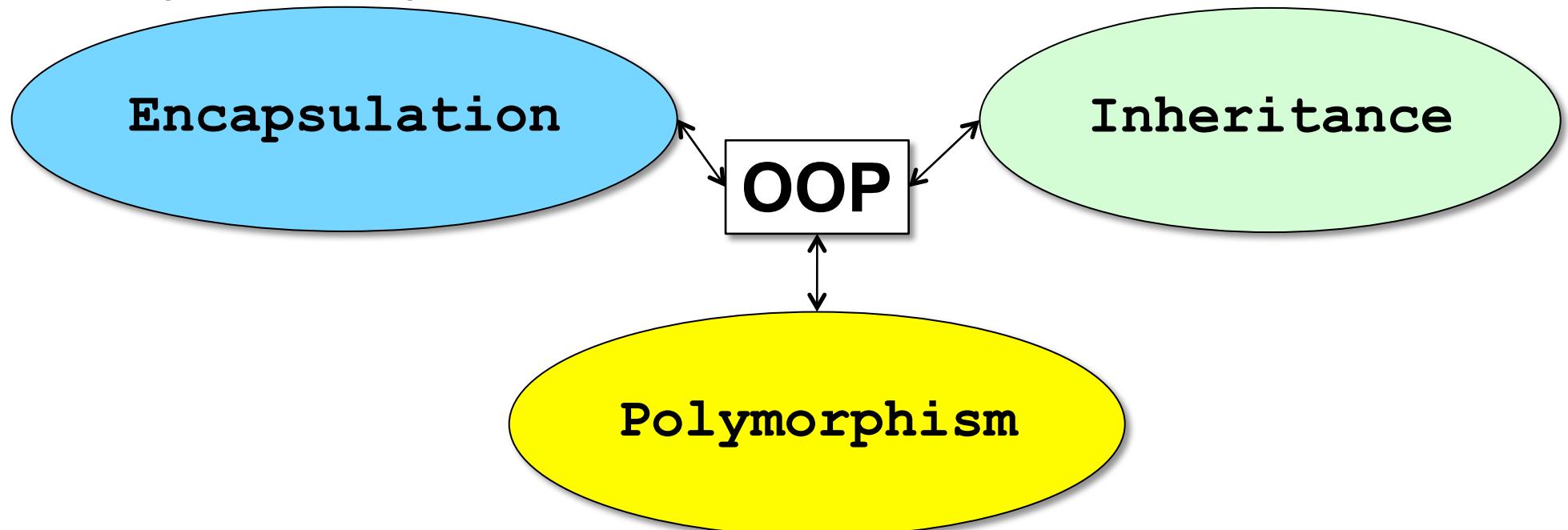
Indice

- Cosa è la programmazione ad oggetti:
 - Incapsulamento
 - Ereditarietà
 - Polimorfismo
- Costruiamo insieme l'oggetto istogramma
- Esercizi

- Approfondimenti
 - struct
 - typedef

Programmazione ad oggetti

- La **programmazione ad oggetti**, **Object Oriented Programming - OOP**, è un paradigma di programmazione, cioè è un modo particolare di programmare
- Programmare ad oggetti vuol dire scrivere un programma che implementa queste tre caratteristiche fondamentali



- Vediamo insieme nel dettaglio cosa vogliono dire

Incapsulamento

- Fino ad ora abbiamo operato sui dati, per esempio sequenze di numeri, con delle funzioni indipendenti dai dati (la sequenza di numeri veniva passata alle funzioni tramite un puntatore o una referenza)
- In alcuni casi può essere utile avere a disposizione una struttura che contenga al suo interno sia i dati che le operazioni che devono essere svolte sui quei dati (detti **metodi**, cioè le funzioni)
 - Pensiamo per esempio ai numeri complessi. Non sarebbe bello se il linguaggio di programmazione ci mettesse a disposizione un modo con cui costruire il tipo “numero complesso”: inizializzandolo con due numeri, parte reale e parte immaginaria, ed operando su di esso con dei metodi?
- Il C++ ci permette di realizzare questo mix di dati e metodi

Incapsulamento: esempio

```
#include <iostream>
#include <cmath>
```

```
class ComplexNumber
{
public:
    ComplexNumber(double r, double i)
    {
        real = r;
        imag = i;
    }

    ~ComplexNumber()
    {
        std::cout << "I'm the "
        << "ComplexNumber::destructor"
        << std::endl;
    }

    double modulus()
    {
        return sqrt(real*real + imag*imag);
    }

private:
    double real;
    double imag;
};

int main ()
{
    ComplexNumber c(1,2);
    std::cout << "Modulo: " << c.modulus() << std::endl;
    return 0;
}
```

Definizione del nuovo tipo “numero complesso”

Metodi con cui operare sui dati (sono in genere pubblici, cioè accessibili dall'esterno)

Dati (sono in genere privati, cioè non accessibili dall'esterno)

Istanzia una variabile di tipo **ComplexNumber** ed inizializzo la sua parte reale a 1 e la sua parte immaginaria a 2. Uso poi il suo metodo pubblico **modulus** per calcolare il modulo

Da notare ...

```
int var (10);
```

Tipo generico, con i
suoi metodi: +, *, /,
-, modulus

Nome della variabile
istanziata nel codice

Definisco un oggetto di tipo
`int` contenente il valore 10

Valore di inizializzazione
della variabile

ComplexNumber

```
c (1,2);
```

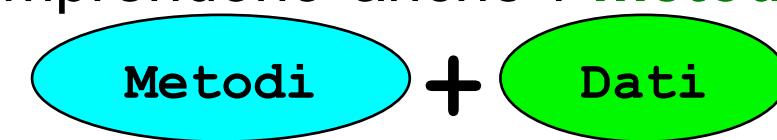
Definisco un oggetto di tipo `ComplexNumber`
contenente parte reale 1 e parte immaginaria 2

Le classi

- La **classe** è la struttura astratta in cui si vuole memorizzare ed organizzare un insieme di dati atti a descrivere un determinato tipo di **oggetto**

Esempio:

- la classe “automobile” sarà strutturata in modo da poter contenere tutti i parametri che definiscono un’automobile: marca, modello, numero di targa, cilindrata, cavalli, combustibile, optional, ...
- una volta definita la struttura astratta della classe automobile, sarà poi possibile **istanziare** diversi **oggetti** appartenenti a questa classe, ciascuno dei quali rappresenta una specifica automobile
- Nel C++ le classi non sono solo strutture contenenti dati numerici, ma comprendono anche i **metodi**, cioè le istruzioni per gestire i dati



Ereditarietà

Un ulteriore elemento caratteristico della programmazione ad oggetti è l'**ereditarietà**. Cioè la facoltà di estendere le proprietà di una classe facendola ereditare da una classe madre

```
#include <iostream>
#include <cmath>

class Shape
{
public:
    Shape(double a)
    {
        area = a;
    }

    ~Shape()
    {
        std::cout << "I'm the "
        << "Shape::destructor"
        << std::endl;
    }

    double getArea()
    {
        return area;
    }

private:
    double area;
};

...
```

“Superclass”

```
...
class Circle: public Shape
{
public:
    Circle(double a, int c) : Shape(a)
    {
        color = c;
    }

    ~Circle()
    {
        std::cout << "I'm the "
        << "Circle::destructor"
        << std::endl;
    }

    double getRadius()
    {
        return sqrt(getArea() / M_PI);
    }

    int getColor()
    {
        return color;
    }

private:
    int color;
};

...
```

“Subclass”

```
...
int main()
{
    Shape sh(1);
    Circle cir(4,0);

    std::cout << "Area: "
    << sh.getArea() << std::endl;

    std::cout << "Area: "
    << cir.getArea() << " radius: "
    << cir.getRadius() << std::endl;

    return 0;
}
```

Polimorfismo

Il terzo, forse più importante, elemento caratteristico della programmazione ad oggetti è il **polimorfismo**. Cioè la facoltà di specializzare un oggetto a **compile time** o addirittura a **run time**

- Per esempio ho una classe “base o superclass” che si chiama Shape che possiede solo l’attributo “area”. Di se però non sa dire il perimetro perché dipende dal tipo di shape (cerchio, quadrato, rettangolo, ecc...)
- Ho una classe derivata, cioè che eredita da Shape, che si chiama Circle e che possiede anche il metodo getPerimeter()
- Ho una classe derivata, cioè che eredita da Shape, che si chiama Square e che possiede anche il metodo getPerimeter()
- Istanziò una variabile di tipo Shape e decido a *run time* se questa è un Circle oppure uno Square e quando chiamo il metodo getPerimeter() mi fornisce la risposta appropriata:

```
...
Shape* sh;
if ("condition")
    sh = new Circle(...);
else
    sh = new Square(...);
sh->getPerimeter();
```

Per maggiori dettagli vedere
esempio nelle ultime slide

Riassumendo ...

- Un programma che segue il paradigma della programmazione ad oggetti deve essere scritto in maniera tale che implementi: **incapsulamento, ereditarietà, polimorfismo**
- Il C++ mette a disposizione tutti gli strumenti per programmare OOP, ma il fatto solo di usare un compilatore C++ non significa necessariamente che uno stia programmando OOP ... giusto?
- La programmazione OOP va usata con parsimonia perché non sempre è necessaria o utile, infatti se usata male può risultare in un codice poco comprensibile e complicato oltre misura
- Per appropriarvi a pieno di questo nuovo paradigma di programmazione dovete programmare, programmare, programmare, ...

Riassumendo ...

Abbiamo solo raschiato la superficie. La programmazione ad oggetti è addirittura ancora più versatile (e complicata) di come è stata descritta. **Per chi volesse approfondire:**

- A proposito dei metodi vedere significato ed utilizzo della variabile nascosta `this` e dell'attributo `virtual`, vedere il concetto di `static member function` (i.e. `static method`)
- A proposito delle classi vedere il significato degli attributi `public`, `private`, `protected`
- A proposito dell'ereditarietà vedere il concetto di “multiple inheritance” e di classe `friend`
- A proposito del polimorfismo vedere i concetti di “static e dynamic dispatch” (detti anche “early e late binding”)
- Vedere il concetto di “association”, “aggregation” e “composition”

Vediamo ora in dettaglio un **eSEMPIO DI INCAPSULAMENTO**:
costruiamo una classe istogramma ...

L'istogramma “artigianale”

Nelle scorse lezioni abbiamo sviluppato un insieme di strumenti adatti a “gestire” una densità di probabilità discreta:

Calcolo di statistiche

$$\sigma^2 = E[x^2] - \mu^2, \quad E[x^2] = \frac{\sum_{i=1}^N}{N}$$

Riempimento

```
for (int it = 0; it < max_num; it++)
{
    double numero_casuale = rand_CLT(xMin, xMax, nTries);
    int index = floor((numero_casuale - xMin)/step);
    histo[index]++;
}
```

Inizializzazione

```
int nBins = 30;
int* histo = new int[nBins];
for (int i = 0; i < nBins; i++)
    histo[i] = 0;
```

Si può mettere tutto insieme?

Calcolo del massimo

```
int getHistoMax (int* histo, int nBins)
{
    int max = 0;
    for (int i = 0; i < nBins; i++)
        if (histo[i] > max) max = histo[i];
    return max;
}
```

Visualizzazione

La classe istogramma

```
class istogramma
```

```
int numero_bin;  
double minX;  
double maxX;  
int* hist;
```

```
double getMax();  
void Fill(double input);
```

In una classe sono presenti:

- **Data member:** variabili interne alla classe

e.g. il numero totale di bin dell'istogramma, il range della variabile x , il vettore dei conteggi di ciascun bin, ...

- **Metodi o member functions:** funzioni interne alla classe

e.g. per restituire il numero del bin con il numero maggiore di conteggi, per incrementare i conteggi di un certo bin, ...

La classe istogramma

```
class istogramma
{
    private
        int numero_bin;
        double minX;
        double maxX;
        int* hist;

    public
        double getMax();
        void Fill(double input);
}
```

Di solito:

- I **data member** sono collocati nella sezione “**privata**”
- I **metodi** sono collocati nella sezione “**pubblica**”

Per avere maggior controllo, e per proteggere i data member da utilizzi inappropriati da parte di chi usa la classe, l’interazione con i data member avviene tramite opportuni metodi detti “**Getter**” e “**Setter**”

Tipi di data, o function, member:

public: l’accesso è consentito anche al di fuori della classe

private: nessuno, se non la classe medesima, può averne accesso

Scrivere e usare una classe

- La struttura astratta della classe è generalmente definita in un **header file** (il file “.h”)
- I metodi della classe vengono implementati in un file “.cc”
- Il main del nostro programma, dove ad esempio possiamo istanziare (cioè utilizzare) gli oggetti della nostra classe, è definito all'interno di un file “.cpp”
- Per compilare un programma che usa una classe si usa il solito comando:

```
c++ -o exeFile classFile.cc programFile.cpp
```

Digressione sull'attributo const

const si applica al primo attributo alla sua sinistra, se non c'è nulla si applica al primo attributo alla sua destra

```
const int C1 = 10;
```

C1: un intero il cui valore è costante

```
int const C1 = 10;
```

```
const int* C2;
```

C2: un puntatore ad un “const int”,
cioè un puntatore ad un intero costante

```
int const* C2;
```

```
int* const C3;
```

C3: un puntatore costante ad un intero
variabile

```
int const* const C4;
```

C4: un puntatore costante ad un intero
costante

Digressione sull'attributo const

- **Funzioni:**

```
void funzione(int const& C1)
```

- Passa alla funzione la referenza, ma **non** permette di modificare C1

- **Metodi e classi:**

```
classe::metodo () const;
```

- Il metodo **non** può modificare nessun data member della classe
- Se una classe è stata istanziata come **const** si possono usare solo i suoi metodi **const**
- I metodi “Getter” possono essere definiti **const** poiché hanno come solo scopo quello di restituire qualcosa

Definizione della classe

```

class istogramma <--  
{  

  public: <--  

    // Constructor  

    istogramma (const int& nBin, const double& min, const double& max);  

    // Destructor  

    ~istogramma ();  

    // Member functions  

    int Fill (const double& value);  

    void Print () const;  

    double GetMean () const;  

    double GetRMS () const;  

  private: <--  

    // Data members  

    int nBin_p;  

    double min_p;  

    double max_p;  

    double step_p;  

    int* binContent_p;  

    int entries_p;  

    int overflow_p;  

    int underflow_p;  

    double sum_p;  

    double sum2_p;  

}; <--

```

Definizione: file **istogramma.h**

class nomeClasse

public: da quel punto inizia la zona “pubblica”

private: da quel punto inizia la zona “privata”

Ricordarsi il punto e virgola alla fine

Definizione della classe

```

class istogramma
{
public:

    // Constructor
    istogramma (const int& nBin, const double& min, const double& max);

    // Destructor
    ~istogramma ();

    // Member functions
    int      Fill   (const double& value);
    void     Print   () const;
    double   GetMean () const;
    double   GetRMS  () const;

private:
    // Data members
    int      nBin_p;
    double   min_p;
    double   max_p;
    double   step_p;
    int*    binContent_p;
    int      entries_p;
    int      overflow_p;
    int      underflow_p;
    double   sum_p;
    double   sum2_p;
};

```

Definizione: file *istogramma.h*

Costruttore: stesso nome della classe

Distruttore: `~`stesso nome della classe

Metodi: definizione delle “funzioni” interne

Data member: definizione delle variabili della classe

N.B.: sono accessibili da tutti i metodi, i.e. non è quindi necessario “passarli” ai metodi

Implementazione della classe

Implementazione: file istogramma.cc

```
istogramma::istogramma (const int& nBin, const double& min, const double& max):
    nBin_p      (nBin),
    min_p       (min),
    max_p       (max),
    step_p      ((max_p - min_p) / nBin_p),
    binContent_p ( new int[nBin_p] ),
    entries_p   (0),
    overflow_p  (0),
    underflow_p (0),
    sum_p        (0.),
    sum2_p       (0.)
{
    // Azzero gli elementi dell'istogramma
    for (int i = 0; i < nBin_p; ++i)
        binContent_p[i] = 0;
}

istogramma::~istogramma()
{
    delete[] binContent_p;
}
```

Costruttore: dove si fa tutto quello che va necessariamente fatto per creare un oggetto

N.B.: nonostante i parametri siano definiti come referenze (mediante &) è possibile invocare il costruttore passandogli dei numeri, per esempio, `istogramma histo(10,0,1);` solo perché i parametri sono stati dichiarati `const`, altrimenti il compilatore darebbe errore!

Distruttore: “ripulisce” la memoria prima di eliminare l’oggetto per sempre

Implementazione della classe

Implementazione: file istogramma.cc

```
istogramma::istogramma (const int& nBin, const double& min, const double& max):
    nBin_p      (nBin),
    min_p       (min),
    max_p       (max),
    step_p      ((max_p - min_p) / nBin_p),
    binContent_p ( new int[nBin_p] ),
    entries_p   (0),
    overflow_p  (0),
    underflow_p (0),
    sum_p        (0.),
    sum2_p       (0.)
{
    // Azzero gli elementi dell'istogramma
    for (int i = 0; i < nBin_p; ++i)
        binContent_p[i] = 0;
}
```

• nomeclasse::

Per definire costruttori, distruttore e metodi

Si possono **inizializzare** i data member in questo modo (chiamato ***inline initialization***):

Costruttore (val1_init,
val2_init):

var1_p(val1_init),
var2_p(val2_init)

oppure all'interno del costruttore:

var1_p = valore1_init;
var2_p = valore2_init;

Utilizzo della classe (esercizio01.cpp)

```

int main()
{
    srand (1);

    double a;
    double b;
    std::cout << "Inserisci gli estremi dell'intervallo [a,b) in cui generare i numeri: ";
    std::cin >> a >> b;

    int N;
    std::cout << "Inserisci quanti numeri casuali vuoi generare: ";
    std::cin >> N;

    // Istanzia l'istogramma
    double min;
    double max;
    std::cout << "Inserisci gli estremi dell'istogramma [min,max): ";
    std::cin >> min >> max;

    int nBin;
    std::cout << "Inserisci il numero di bin dell'istogramma: ";
    std::cin >> nBin;

    // Costruttore
    istogramma histo(nBin, min, max); ←

    // Riempie l'istogramma
    int estrazioni = 10;
    double random;
    for (int i = 0; i < N; i++)
    {
        random = rand_CLT (min, max, estrazioni);
        histo.Fill(random); ←
    }

    // Print
    std::cout << "Mean = " << std::setprecision(5) << histo.GetMean() << std::endl;
    std::cout << "RMS = " << std::setprecision(5) << histo.GetRMS() << std::endl;
    histo.Print(); ←

    return 0;
}

```

Ricordarsi di includere il .h dell'istogramma prima di

`int main()`

Istanzia l'istogramma come un tipo generico (qui viene chiamato il costruttore)

Riempie l'istogramma

Stampa l'istogramma

Il metodo Fill (histogramma.cc)

```
int histogramma::Fill(const double& value)
{
    if (value < min_p)
    {
        ++underflow_p;
        return -1;
    }

    if (value >= max_p)
    {
        ++overflow_p;
        return -1;
    }

    ++entries_p;

    int bin = int((value-min_p) / step_p);
    ++binContent_p[bin];

    sum_p += value;
    sum2_p += value*value;

    return bin;
}
```

Metodo per il riempimento dell'istogramma

Condizioni per l'inserimento del valore

Il valore del bin corrispondente viene incrementato

Contatori per il calcolo di media e varianza

Il metodo Print (histogramma.cc)

```

void histogramma::Print() const
{
    // Normalizza l'istogramma al valore maggiore
    int max = 0;
    for (int i = 0; i < nBin_p; ++i)
    {
        if (binContent_p[i] > max) max = binContent_p[i];
    }

    // Fattore di dilatazione per la rappresentazione dell'istogramma
    int scale = 50;

    // Disegna l'asse y
    std::cout << "-----> Entries" << std::endl;

    // Disegna il contenuto dei bin
    for (int i = 0; i < nBin_p; ++i)
    {
        std::cout << std::fixed << std::setw(8) << std::setprecision(2) << min_p + i
* step_p << "|";
        int freq = int(scale * binContent_p[i] / max);
        for (int j = 0; j < freq; ++j)
            std::cout << "#";

        std::cout << binContent_p[i] << std::endl;
    }

    std::cout << " | X axis\n" << std::endl;
}

```

Metodo per la stampa su terminale dell'istogramma

Ricerca del massimo

Funzioni per la corretta visualizzazione (libreria <iomanip>)

Esempio di esecuzione

```
Inserisci gli estremi dell'istogramma [min,max): 0 10
Inserisci il numero di bin dell'istogramma: 20
Mean = 5.0028
RMS  = 0.92094
Overflow = 0
Underflow = 0
Max counts = 2017
Integral [0, 10] = 5000
+-----> Entries
0.00|0
0.50|0
1.00|0
1.50|2
2.00|24
2.50|##105
3.00|#####399
3.50|#####856
4.00|#####1587
4.50|#####2017
5.00|#####2012
5.50|#####1557
6.00|#####929
6.50|#####379
7.00|##110
7.50|18
8.00|5
8.50|0
9.00|0
9.50|0
| X axis
```

Come usare classi e puntatori

- Ovviamente è possibile creare dei puntatori anche a degli oggetti. Riprendiamo l'esempio dei numeri complessi:

```
ComplexNumber* c = new ComplexNumber(1, 2)
```

Per chiamare il metodo `modulus()` si dovrà ora scrivere:

```
c->modulus(); // Oppure equivalentemente  
(*c).modulus();
```

Cioè il carattere `-` “meno” seguito dal carattere “maggiore” `>`

E` necessario ricordarsi di chiamare `delete c;` quando non si usa più l'oggetto, per liberare l'area di memoria dinamica allocata

- Nel caso invece in cui si faccia uso della memoria statica:

```
ComplexNumber c(1, 2);
```

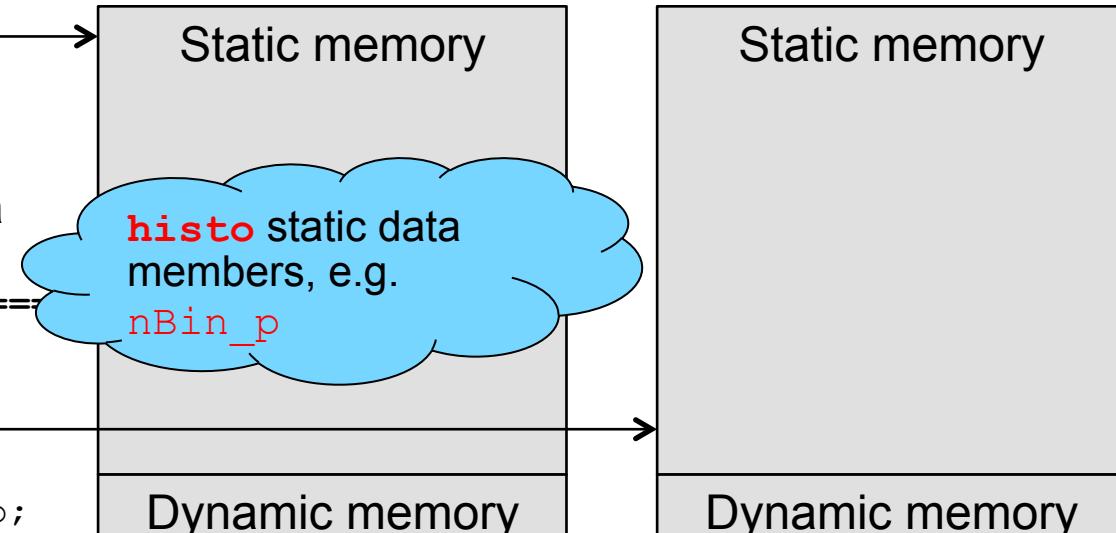
...

```
c.modulus();
```

Quando viene invocato il distruttore?

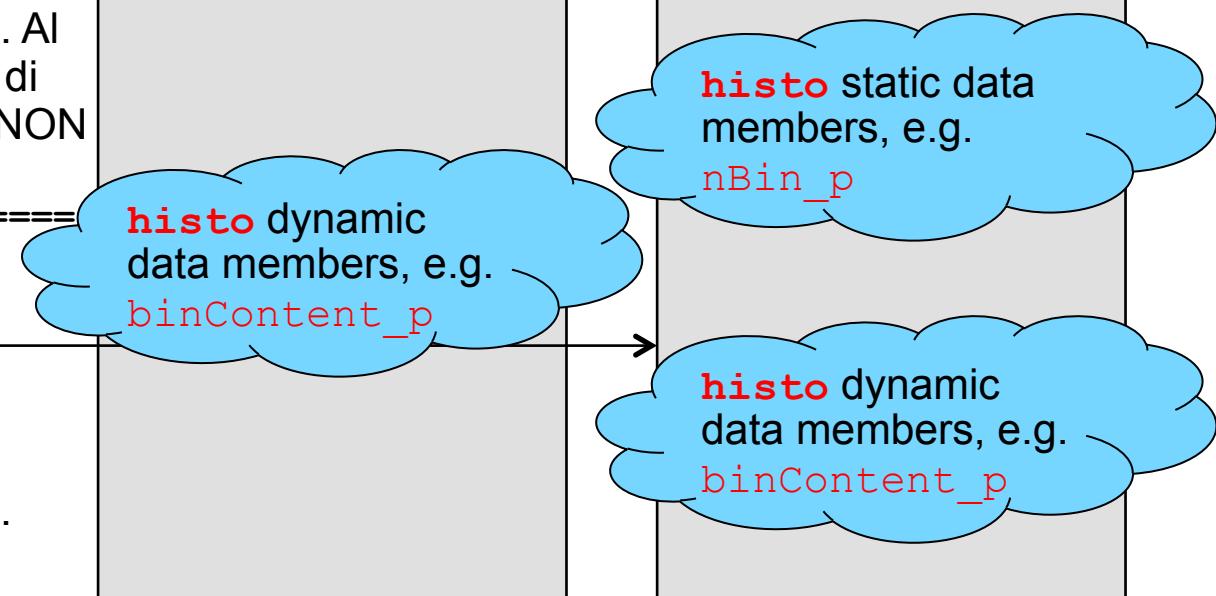
```
{
    istogramma histo(100,0,10);
    ...
}
```

Al termine dello scope viene invocato automaticamente il distruttore dell'istogramma e viene liberata l'area di memoria statica



```
{
    istogramma* histo =
        new istogramma (100, 0, 10);
    ...
    // Non e` stato invocato delete histo;
}
```

`histo` allocato nell'area di memoria dinamica. Al termine dello scope NON viene liberata l'area di memoria dinamica riservata all'istogramma e NON viene invocato il distruttore dell'istogramma



```
{
    istogramma* histo =
        new istogramma (100, 0, 10);
    ...
    delete histo;
}
```

`histo` allocato nell'area di memoria dinamica. L'istruzione `delete histo;` libera l'area di memoria dinamica riservata all'istogramma e invoca a sua volta il distruttore dell'istogramma

Esercizi

Esercizio 1: Scrivere la classe istogramma ampliandola rispetto a quanto presentato aggiungendo i seguenti metodi (e quindi anche i data member necessari):

- `double GetMean ()` → calcola il valor medio
- `double GetRMS ()` → calcola la deviazione standard
- `double GetIntegral ()` → calcola l'integrale
- `int GetOverflow ()` → restituisce il numero di conteggi registrati al di sopra del valor massimo
- `int GetUnderlow ()` → restituisce il numero di conteggi registrati al di sotto del valor minimo

Esercizi

Esercizio 2: Trovare e correggete gli errori riguardanti l'uso dell'attributo **const**

```
#include <iostream>
int fun01 (const int a)
{
    a = 2;
    return a;
}
void fun02 (const int* a)
{
    *a = 2;
}
void fun03 (int* const a)
{
    *a = 2;
}
void fun04 (int* const *a, int* const *b)
{
    *a = *b;
}
void fun05 (int const &a)
{
    a = 2;
}
...
```

```
...
int main(int argc, char** argv)
{
    const int a = 1;
    a = 6;
    std::cout << "a = " << a << std::endl;

    const int* b = new int(1);
    *b = 2;
    int* c = new int(3);
    b = c;
    std::cout << "b = " << *b << std::endl;

    int* const d = new int(1);
    *d = 2;
    int* e = new int(3);
    d = e;
    std::cout << "d = " << *d << std::endl;

    const int f = 1;
    std::cout << "fun01 = " << fun01(f)
              << std::endl;
    int g = 1;
    std::cout << "fun01 = " << fun01(g)
              << std::endl;
...
```

Esercizi

```
...
const int* h = new int(1);
fun02(h);
std::cout << "h = " << *h << std::endl;
int* i = new int(1);
fun02(i);
std::cout << "i = " << *i << std::endl;

const int* j = new int(1);
fun03(j);
std::cout << "j = " << *j << std::endl;
int* k = new int(1);
fun03(k);
std::cout << "k = " << *k << std::endl;

int* l = new int(1);
int* m = new int(2);
fun04(&l, &m);
std::cout << "l = " << *l << std::endl;

const int n = 1;
fun05(n);
std::cout << "n = " << n << std::endl;

return 0;
}
```

Polimorfismo: esempio

Il terzo, forse più importante, elemento caratteristico della programmazione ad oggetti è il **polimorfismo**. Cioè la facoltà di specializzare un oggetto a **compile time** o addirittura a **run time**

```
#include <iostream>
#include <cmath>

class Shape
{
public:
    Shape(double a)
    {
        area = a;
    }

    ~Shape()
    {
        std::cout << "I'm the "
        << "Shape::destructor"
        << std::endl;
    }

    double getArea()
    {
        std::cout << "Shape::getArea() ";
        return area;
    }
    ...
}
```

```
...
virtual double getPerimeter()
{
    std::cout << "Shape::getPerimeter() ";
    return -1;
}

private:
    double area;
};

class Circle: public Shape
{
public:
    Circle(double a, int c) : Shape(a)
    {
        color = c;
        radius = sqrt(a / M_PI);
    }

    ~Circle()
    {
        std::cout << "I'm the "
        << "Circle::destructor"
        << std::endl;
    }
    ...
}
```

Polimorfismo: esempio

```
...
double getPerimeter()
{
    std::cout << "Circle::getPerimeter() ";
    return 2.*M_PI*sqrt(Shape::getArea() / M_PI);
}

int getColor()
{
    return color;
}

private:
    int color;
    int radius;
};

class Square: public Shape
{
public:
    Square(double a, int c) : Shape(a)
    {
        color = c;
        side = sqrt(a);
    }

    ~Square()
    {
        std::cout << "I'm the "
        << "Square::destructor"
        << std::endl;
    } ...
}
```

```
...
double getPerimeter()
{
    std::cout << "Square::getPerimeter() ";
    return 4.*sqrt(Shape::getArea());
}

int getColor()
{
    return color;
}

private:
    int color;
    int side;
};

...
```

Polimorfismo: esempio

```
...
int main()
{
    Shape* sh = new Shape(2);

    std::cout << "Area: "
    << sh->getArea() << " perimeter: "
    << sh->getPerimeter() << std::endl;

    delete sh;
    sh = new Circle(4,0);

    std::cout << "Area: "
    << sh->getArea() << " perimeter: "
    << sh->getPerimeter() << std::endl;

    delete sh;
    sh = new Square(4,0);

    std::cout << "Area: "
    << sh->getArea() << " perimeter: "
    << sh->getPerimeter() << std::endl;

    return 0;
}
```

Approfondimenti

- **struct**
- **typedef**

Il tipo struct

La struttura dati `struct` consente di raggruppare insieme elementi di tipo diverso sotto uno stesso nome. Gli elementi sono chiamati membri e la sintassi per dichiarare una `struct` è la seguente:

```
struct type_name
{
    member_type1 member_name1;
    member_type2 member_name2;
    member_type3 member_name3;
    ...
} object_names;
```

Esempio:

```
struct book
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} book1, book2;
```

Abbiamo definito un nuovo tipo, il tipo `struct book`

Abbiamo definito due oggetti, il cui nome è `book1` e `book2`, di tipo `struct book`

Dopo aver dichiarato gli oggetti è possibile accedere ai membri della `struct` tramite il punto (.) tra il nome dell'oggetto ed il membro, e.g. `book1.title`

Il tipo struct

```
#include <iostream>
#include <cstring>
using namespace std;

struct book
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

void printBook (struct book myBook)
{
    cout << "Book title : " << myBook.title << endl;
    cout << "Book author : " << myBook.author << endl;
    cout << "Book subject : " << myBook.subject << endl;
    cout << "Book id : " << myBook.book_id << endl;
}

int main()
{
    struct book book1, book2;

    strcpy(book1.title, "The C++ programming language");
    strcpy(book1.author, "Bjarne Stroustrup");
    strcpy(book1.subject, "C++ Programming");
    book1.book_id = 123456;
    ...
}
```

Non dichiaro nessun oggetto in questo punto, lo faccio nel main

Il tipo struct

```
...
    strcpy(book2.title, "Thinking in C++");
    strcpy(book2.author, "Bruce Eckel");
    strcpy(book2.subject, "C++ Programming");
    book2.book_id = 654321;

    printBook(book1);
    printBook(book2);

    return 0;
}
```

Ovviamente è anche possibile definire delle variabili che puntano a delle **struct**:

struct book* book1;

In questo caso si fa riferimento ai membri della **struct** mediante la freccia (**->**), e.g.
book1->title

Esercizio: Modificare il programma precedente in maniera tale che la funzione printBooks riceva in input un array di **struct book**: e.g. **void** printBooks (**struct book*** myBooks, **unsigned int** nBooks). Modificare inoltre il main in maniera tale che faccia uso di un array di **struct book**

typedef

La sintassi del C++ mette a disposizione un modo per ridefinire i tipi al fine di avere nomi più semplici e corti da utilizzare, i.e. tramite il `typedef`

Per esempio, invece di scrivere `struct book` ogni volta che facciamo riferimento a questo nuovo tipo, possiamo ridefinirlo in questo modo:

```
typedef struct
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} book;
book book1, book2;
```

Oppure:

```
struct book_
{
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};
typedef struct book_ book;
book book1, book2;
```

Nuovo nome assegnato
al tipo `struct book`

typedef

Altri esempi di uso del `typedef` sono:

```
typedef char C;  
typedef unsigned int WORD;  
typedef char* pCHAR;
```

ecc...

Possono quindi essere usati come normali tipi:

```
C myChar;  
WORD myWord;  
pCHAR myPtrC;  
ecc...
```

Esiste un'altra sintassi per fare la stessa cosa del `typedef` che è più “C++ oriented”:

```
using new_type_name = old_type_name;
```

E.g.

```
struct book_  
{  
    char title[50];  
    char author[50];  
    char subject[100];  
    int book_id;  
};  
using book = struct book_;
```