Laboratorio II – 1° modulo
Lezione 5

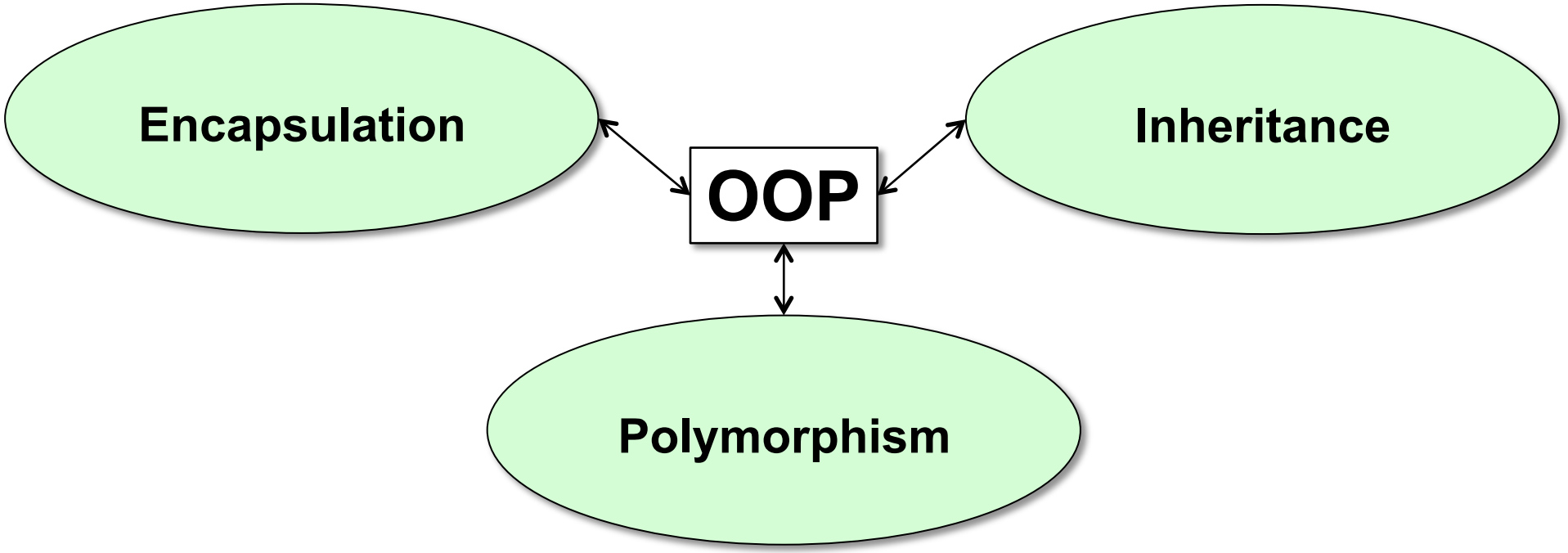
Programmazione ad oggetti

Indice

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

Programmazione ad oggetti

- La programmazione ad oggetti, **O**bject **O**riented **P**rogramming - **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 per puntatore o 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;
}
```

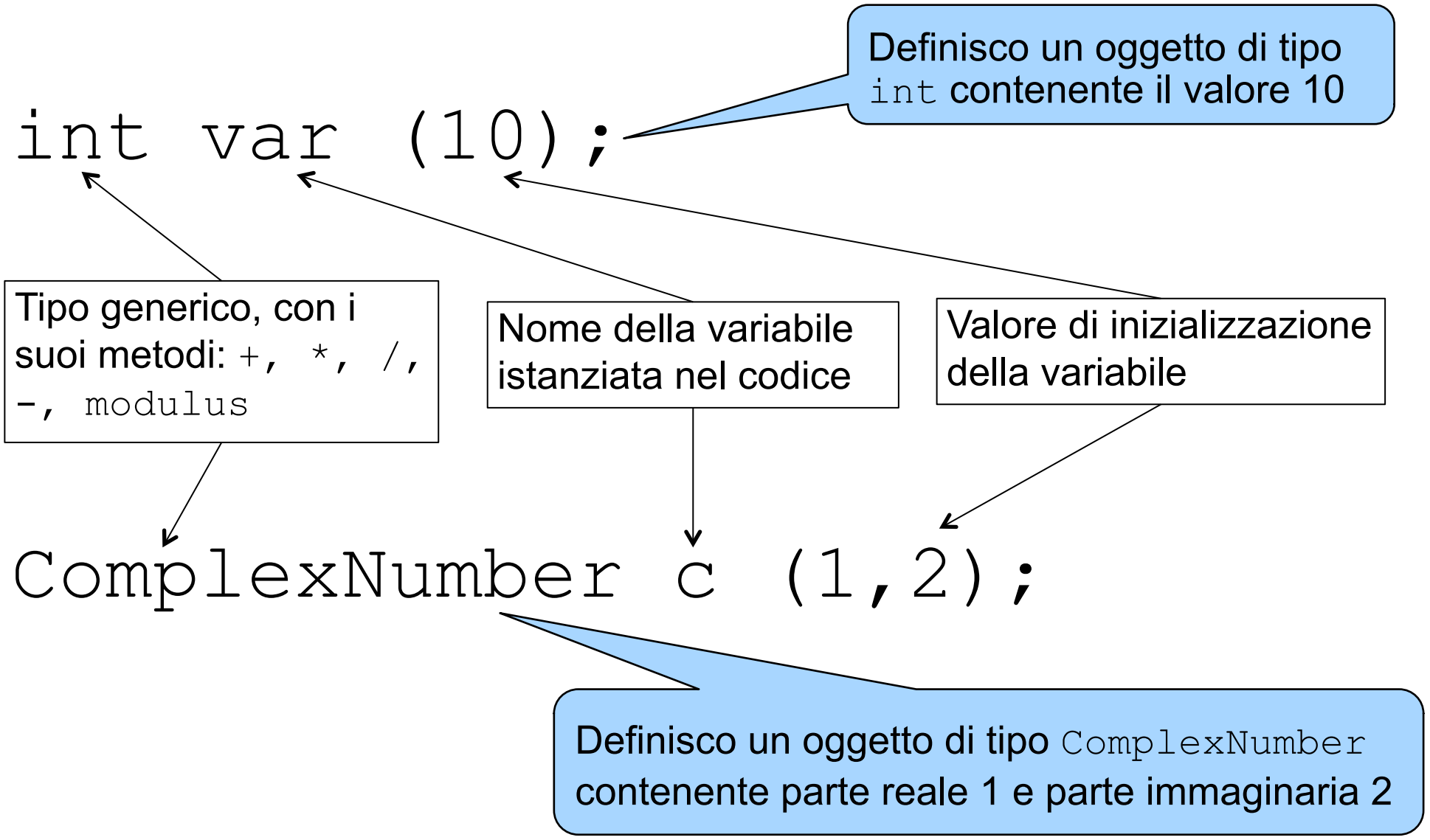
Definizione del nuovo tipo “numero complesso”

Metodi con cui operare sui dati (sono in genere pubblici, cioe` accessibli dall’esterno)

Dati (sono in genere privati, cioe` non accessibili dall’esterno)

Istanzio 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 …


Definisco un oggetto di tipo int contenente il valore 10
int var (10);
Tipo generico, con i suoi metodi: +, *, /, -, modulus
Nome della variabile istanziata nel codice
Valore di inizializzazione della variabile
ComplexNumber c (1,2);
Definisco un oggetto di tipo ComplexNumber contenente parte reale 1 e parte immaginaria 2

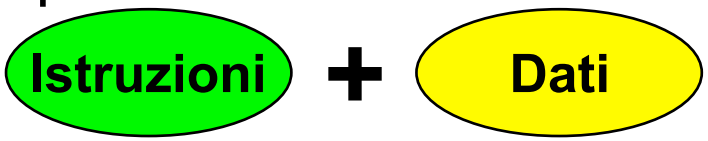
Le classi

- 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** con 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

"Superclass"

```
#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;
};
...
```

"Subclass"

```
...
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;
};
...
```

```
...
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

Un ulteriore elemento caratteristico della programmazione ad oggetti è il **polimorfismo**. Cioè la facoltà di specializzare un oggetto a **compiletime** o **runtime**

- 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()`
- Istanzio una variabile di tipo `Shape` e decido a runtime se questa è un `Circle` oppure uno `Square` e quando chiamo il metodo `getPerimeter()` mi fornisce la risposta appropriata:

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

Per maggiori detagli 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 risultate 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 la classe istogramma …

L'istogramma "artigianale"

Nelle scorse lezioni abbiamo sviluppato un insieme di strumenti adatti a "gestire" una pdf discreta:

Calcolo di statistiche

$$\sigma^2 = E[x^2] - \mu^2, \; E[x^2] = \frac{\sum x_i^2 \times N_i}{\sum N_i}$$

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 ;
}
```

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] ;
 }
```

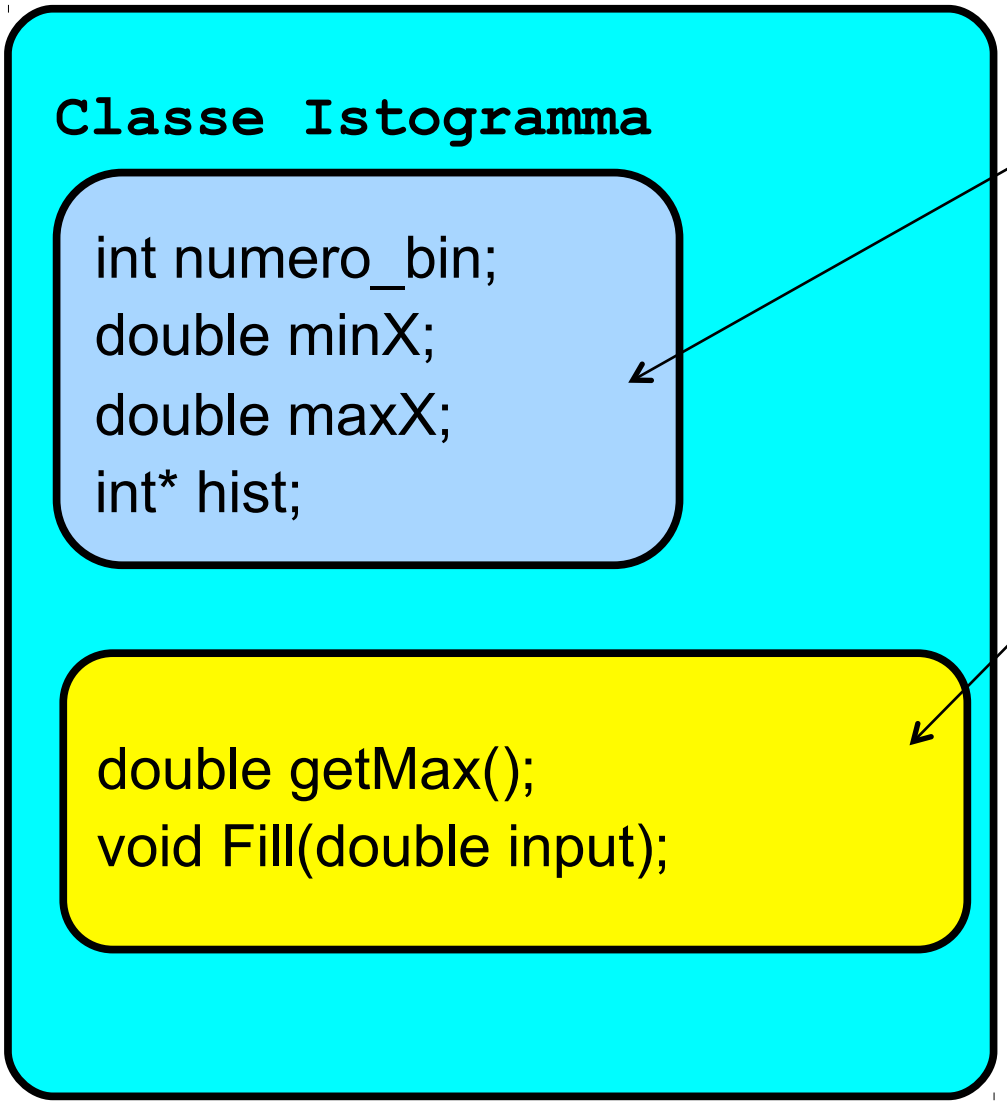
Visualizzazione

Inizializzazione

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

Si può mettere tutto insieme?

La classe istogramma



In una classe ci sono:

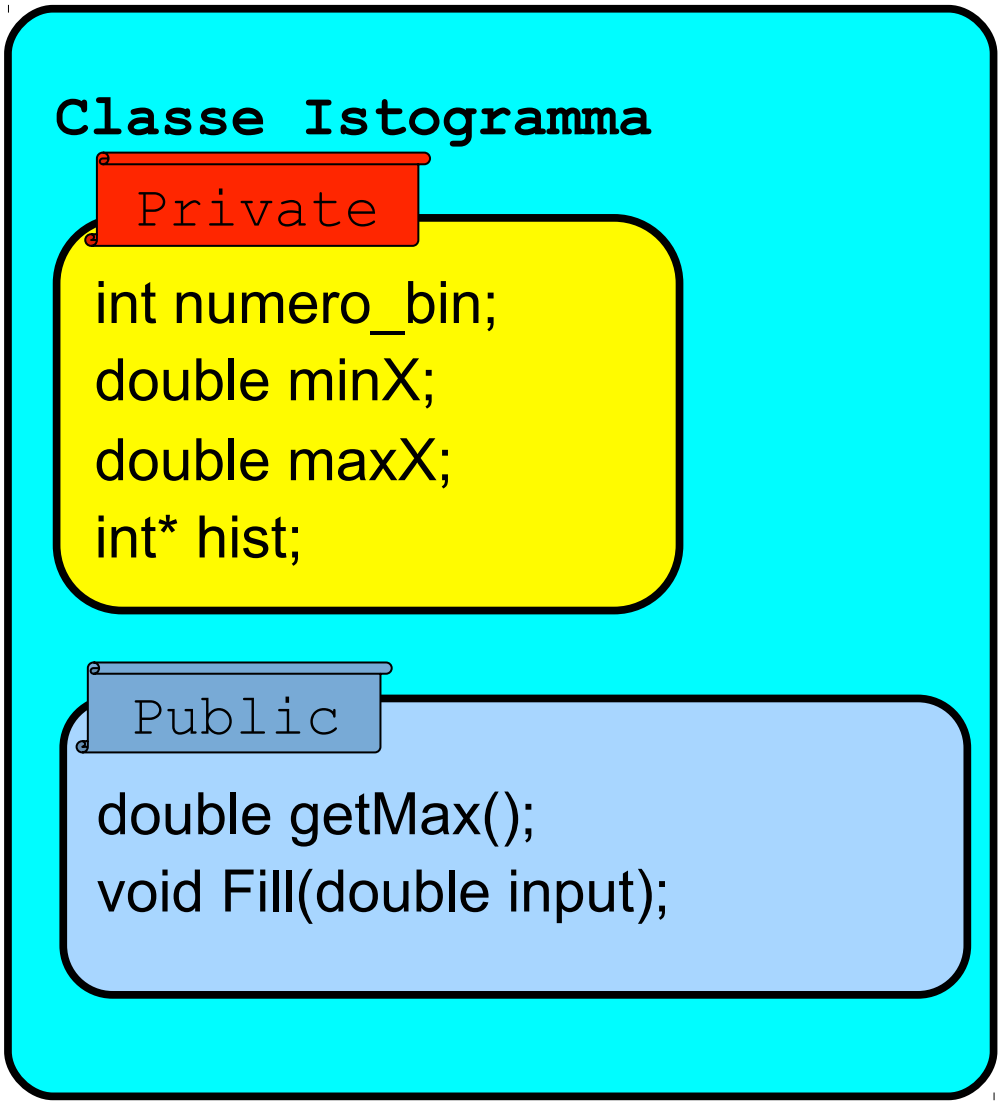
- Attributi

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

- Metodi

sono "funzioni" interne della classe; e.g. dimmi il bin con il numero di entry maggiore, riempi un bin con un entry, …

La classe istogramma



Di solito:

- Gli attributi sono nella parte "privata"
- I metodi nella parte "pubblica"

Per avere maggior controllo, l'interazione con le variabili avviene tramite opportuni metodi (Get o Set)

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 ad esempio il comando:

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

Come usare il `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;
int const C1 = 10;
```

`C1`: un intero il cui valore è costante

```
const int * C2;
int const * C2;
```

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

```
int * const C3;
```

`C3`: un puntatore costante ad un intero variabile

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

`C4`: un puntatore costante ad un intero costante

Il `const` nelle classi

- 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 attributo della classe
  - Se una classe è stata istanziata come `const` si possono usare solo i suoli metodi `const`
  - I metodi "`get`" possono essere definiti `const`

Definizione della classe

```
class istogramma
{

 public:

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

  // Destructor
  ~istogramma ();

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

 private:

  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 "private"

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 ();

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

 private:

  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
- Attributi: definizione degli “ingredienti” della classe

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: dove si pulisce 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 gli attributi in questo modo:

```
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;

  // Istanzio 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;

  // Construtor
  istogramma histo(nBin, min, max);

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

  // Stampo
  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`

```
int istogramma::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 aumentato

Il metodo `Print`

```
void istogramma::Print() const
{
  // Normalizza l'istogrammma 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 << "        +---------------------------------------->" << 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 << std::endl;
    }

  std::cout << "        |\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'intervallo [a,b) in cui generare i numeri: 0 10
Inserisci quanti numeri casuali vuoi generare: 10000
Inserisci gli estremi dell'istogramma [min,max): 0 10
Inserisci il numero di bin dell'istogramma: 20
Mean = 5.0028
RMS  = 0.84812
       +-------------------------------------------------------->
    0.00|
    0.50|
    1.00|
    1.50|
    2.00|
    2.50|##
    3.00|#########
    3.50|######################
    4.00|########################################
    4.50|##################################################
    5.00|##############################################
    5.50|#####################################
    6.00|########################
    6.50|#########
    7.00|##
    7.50|
    8.00|
    8.50|
    9.00|
    9.50|
        |
```

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()` dovrò ora scrivere:

```
c->modulus();
```

Cioè il carattere – "meno" seguito dal carattere "maggiore" >

Ricordatevi sempre di chiamare `delete c;` quando non usate più l'oggetto!

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

```
ComplexNumber c(1,2);

...

c.modulus();
```

Esercizi

- **Esercizio 1**: A partire dal codice di esempio della classe istogramma, ampliarla aggiungendo i seguenti metodi (e quindi anche gli attributi 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

Polimorfismo: esempio

Un ulteriore elemento caratteristico della programmazione ad oggetti è il **polimorfismo**. Cioè la facoltà di specializzare un oggetto a **compiletime** o **runtime**

```
#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()”
    << std::endl;
    return area;
  }
...
```

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

  int getColor()
  {
    return color;
  }

private:
  int color;
  int radius;
};

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;

  return 0;
}
```