

Laboratorio II – 1°modulo

Lezione 12

Esercizi

Laboratorio II – 1°modulo

Lezione 12

Esercizi

Indice

- **Esercizio 1:** Implementazione di una classe per l'analisi dati
- **Esercizio 2:** Confronto tra il metodo dei minimi quadrati e quello della massima verosimiglianza nel fit di un istogramma con un numero “piccolo” di conteggi
- **Esercizio 3:** Calcolo e visualizzazione delle orbite di pianeti
- **Esercizio 4:** Calcolo e visualizzazione del frattale di Mandelbrot
- Approfondimenti consigliati
 - Esercizio polimorfismo & ereditarietà

Esercizio 1

Si vuole implementare una **classe “Analyzer”** per l'**analisi dati** che sia in grado di eseguire le seguenti operazioni:

- Lettura dei dati da un file, riempimento di opportuni container per i valori letti, calcolo dei valori minimo e massimo della x , riempimento di un opportuno oggetto per la visualizzazione grafica ([TH1D](#) nel caso di esperimento di conteggio, [TGraphErrors](#) nel caso di misure)
- Calcolo della media campionaria, della deviazione standard campionaria e dell'errore sulla media (con la possibilità di usare la media pesata e il relativo errore quando, oltre ai valori numerici, vengono forniti gli errori ad essi associati)
- Fit dell'istogramma o del grafico (dovrà essere lasciata la possibilità, a chi utilizza la classe, di scegliere la funzione e il *range*)

Esercizio 1

- Calcolo del Chi-2, del numero di gradi di libertà e del p-value (vedere metodo `Prob(...)` della classe `TMath.h`) associati al fit dell'istogramma o del grafico
- Calcolo compatibilità tra due misure ($M_1 \pm \sigma_1$ e $M_2 \pm \sigma_2$) in base a un test Gaussiano oppure t-Student (selezionabili tramite un parametro)
- Calcolo della curva di livello per $\Delta M \text{ in } \text{rad}^2 = \text{delta}$ della funzione utilizzata per il fit

Realizzare in seguito un main che istanzi un oggetto di tipo `Analyzer` ed esegua tutte le operazioni descritte sopra, stampando a schermo i risultati opportuni

Su **e-learning** trovate i file `data1.txt` (*cfr. Lezione 10 – esercizio 1*) e `ese01Fit.txt` (*cfr. Lezione 11 – esercizio 1*) per testare il codice nel caso di istogramma e grafico rispettivamente, oltre che al file dei prototipi della classe, `Analyzer.h`

Esercizio 2

Si vuole realizzare un programma per confrontare il risultato del fit di un **istogramma** contenente **pochi conteggi** utilizzando i metodi dei **minimi quadrati** e della **massima verosimiglianza**

- Il file `datiS.txt` contiene N numeri casuali campionati da una pdf uniforme: $\text{pdf}(x) = 0.1$ per $0 \leq x \leq 10$, 0 altrimenti
- Lo si legge e si riempiono – con gli stessi dati – due istogrammi `TH1D` (che avranno contenuto identico). Si usa poi il metodo `Fit()` per interpolare gli histogrammi con una funzione costante, usando per il primo istogramma il metodo dei minimi quadrati (nessuna opzione) e per il secondo istogramma quello della Likelihood (opzione "`L`"). Si confrontano i risultati ottenuti nei due casi con il risultato atteso. Si verifica quindi come è calcolato il Chi-2 associato ai due fit usando le formule descritte nella traccia indicata qui di seguito

Esercizio 2

1. Definire due histogrammi (e.g. `TH1D* histoMin2 e TH1D* histoLike`) entrambi di 10 bin e di estremi 0 e 10. Riempirli con le N estrazioni lette dal file `datiS.txt` usando il metodo `Fill()`. Nel ciclo di lettura del file calcolare N . Stampare quindi a schermo:
 - il numero N di estrazioni effettuate dalla `pdf(x)` uniforme
 - il contenuto k_i di ciascun bin dell'istogramma (dove $i = 0, 1, \dots, 9$)
 - l'errore e_i che ROOT ha associato a ciascun bin dell'istogramma
 - un commento su quale sia la `pdf(k_i)` che descrive la variabile aleatoria k_i , e su come abbia fatto ROOT ad associare un errore a ciascun k_i
 - il valore di aspettazione $\mu = E[k_i]$ per la variabile random k_i e la sua deviazione standard $\sigma = \text{Var}[k_i]$ determinati conoscendo la `pdf(k_i)` e il numero N di estrazioni (ovviamente tutti i bin hanno lo stesso valore di aspettazione e la stessa varianza e deviazione standard, ricordate che i vostri histogrammi non sono una pdf perché non sono normalizzati per avere area pari a 1)

Esercizio 2

-
2. Fittare entrambi gli histogrammi con una funzione costante $k_i = K$ definita nell'intervallo $[0, 10]$:
 - fittare `TH1D*` `histoMin2` usando il metodo dei minimi quadrati, stampare a schermo il valore di K così stimato (sia K_C), il suo errore e il relativo Chi-2
 - fittare `TH1D*` `histoLike` con il metodo della Likelihood, stampare a schermo il valore di K così stimato (sia K_L), il suo errore e il relativo Chi-2
 3. Confrontare quantitativamente il valore ottenuto per la costante K nei due casi (K_C e K_L) con il valore atteso $\mu = E[k_i]$ calcolato al punto 1. (inserire un commento che spieghi che metodo è stato usato per fare il confronto e quale è il suo esito)
 4. Implementare una funzione `void computeChi2(TH1D* histo, double K)` che usando i valori di k_i contenuti nell'istogramma, e il valore stimato della costante K , determini il valore del Chi-2 associato ai due fit e calcoli il p-value per il test di bontà del fit considerando le seguenti due definizioni:

Esercizio 2

$$\chi_C = \sum_{i=1}^N \frac{(k_i - K)^2}{e^2 i}$$

a) solo per bin non vuoti

$$\chi_L = 2 \sum_{i=1}^N (K - k_i + k_i \log(k_i / K))$$

b) sia per bin vuoti che per bin non vuoti

- a) Questo primo metodo è la funzione che ROOT ha minimizzato per stimare K . Va confrontato con il valore che ROOT fornisce tramite il metodo `GetChisquare()`. Come si calcolano i gradi di libertà?
- b) Questo secondo metodo è usato da ROOT nel fit con il funzionale Likelihood, in cui si tiene conto anche dei bin a contenuto nullo (cfr. S.Baker and R.D.Cousins, “Clarification of the use of chi-square and likelihood functions in fits to histograms,” *Nucl. Instrum. Meth.* 221 (1984) 437). Derivazione dell'espressione di χ_L :

$$-2 \log \frac{L(k; K)}{L(k; k)} = -2 \log \frac{\prod_{i=1}^N \frac{e^{-K} K^{k_i}}{k_i!}}{\prod_{i=1}^N \frac{e^{-k_i} k_i^{k_i}}{k_i!}} = -2 \sum_{i=1}^N \log(k_i) + k_i \log(K) - K + 2 \sum_{i=1}^N \log(k_i) + k_i \log(k_i) - k_i = 2 \sum_{i=1}^N (K - k_i + k_i \log(k_i / K))$$

Si può dimostrare (teorema di Wilks) che la distribuzione di probabilità di χ_L , per N grande, tende ad una distribuzione di tipo χ^2 con $N-n_{\text{par}}$ gradi di libertà (nel nostro caso $n_{\text{par}} = 1$)

Esercizio 2

$$\chi_C = \sum_{i=1}^N \frac{(k_i - K)^2}{e^2 i}$$

a) solo per bin non vuoti

$$\chi_L = 2 \sum_{i=1}^N (K - k_i + k_i \log(k_i / K))$$

b) sia per bin vuoti che per bin non vuoti

χ_L così calcolato va confrontato con il valore che ROOT fornisce tramite

```
TFitResultPtr resultL = histoL->Fit("pol0", "SL");
```

```
chil = 2. * resultL->MinFcnValue();
```

In questo caso il numero di gradi di libertà è pari al numero di bin - 1

Stampare a schermo i due Chi-2, i Chi-2 ridotti, ed il p-value (vedere metodo Prob(...) della classe **TMath.h**)

5. Ripetere l'esecuzione del programma leggendo il file datil.txt. Cosa è cambiato? Secondo voi perché?

Esercizio 3

Dati 3 corpi celesti descritti da:

- masse m_1, m_2, m_3 ,
- posizioni iniziali (vettori nel piano dei 3 corpi) r_1, r_2, r_3
- velocità iniziali (vettori nel piano dei 3 corpi) v_1, v_2, v_3

calcolare, usando la legge di gravitazione di Newton (vettoriale), le orbite dei tre corpi e rappresentarle nel piano

Risolvere le equazioni del moto per via numerica utilizzando le seguenti espressioni (vettoriali):

$$v(t + \Delta t) = a \Delta t + v(t)$$

$$r(t + \Delta t) = v \Delta t + r(t)$$

- Dopo aver impostato le condizioni iniziali dei 3 corpi celesti (massa, posizione e velocità), aggiornare ad ogni step temporale Δt la velocità e la posizione di ciascuno
- Scegliere la lunghezza dello step temporale Δt in modo da ottenere soluzioni numeriche sufficientemente precise e, allo stesso tempo, ottimizzare il tempo computazionale

Esercizio 3

Gestire i calcoli utilizzando due classi:

- a) una classe vettore in 2D (**vett2d**) che memorizzi le coordinate x e y dei vettori, e consenta di fare su di essi tutti i calcoli necessari (+, -, =, mod(), etc...)
- b) una classe **planet** che memorizzi massa, posizione, velocità del corpo celeste, e consenta di fare con esse i calcoli necessari

In particolare, implementare il metodo **DoStep** che riceve in input i puntatori degli altri due corpi celesti del sistema e lo step temporale:

```
void DoStep(planet* p11, planet* p12, double dt);
```

Questo metodo deve:

- creare i vettori **a1** e **a2**, che corrispondono alle accelerazioni del corpo celeste in oggetto causate dalla presenza degli altri due corpi celesti (p11 e p12)
- calcolare l'accelerazione totale: **a = a1 + a2**
- aggiornare i vettori velocità e posizione del corpo celeste in oggetto con le formule riportate nella slide precedente

Utilizzare gli header file (.h) delle due classi contenenti più dettagli e commenti utili per la realizzazione delle classi

Esercizio 3

Creare un `main` in cui si definiscano tre corpi celesti (crearli tramite `new`)

Si definiscano anche 3 oggetti `TGraph` per rappresentare le orbite dei corpi celesti in un `TCanvas`

Impostare un ciclo `for` in cui, per ciascun corpo celeste:

- venga chiamato il metodo `DoStep`
- vengano inserite le coordinate nel `TGraph` corrispondente

Si suggerisce di partire dal calcolo di un sistema noto (Sole, Terra, Marte), per cui è ragionevole usare uno step $\Delta t = 2$ ore:

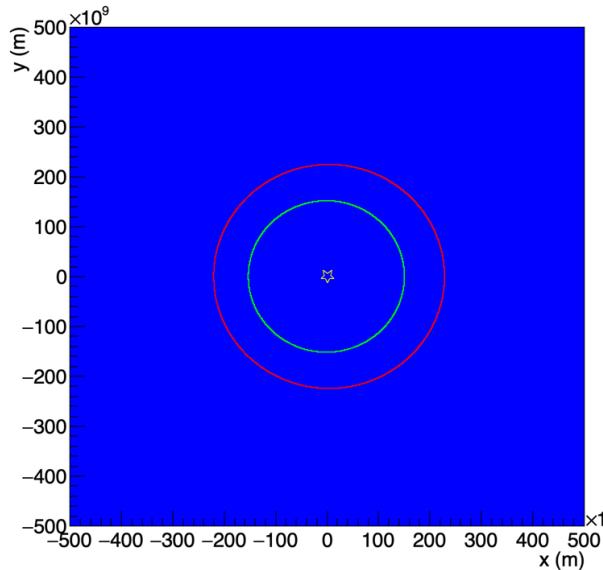
- **Sole:** $r=0$ $v=0$ $m=2 \times 10^{30}$ kg
- **Terra:** $r=150 \times 10^9$ m $v=30 \times 10^3$ m/s $m=5.97 \times 10^{24}$ kg
- **Marte:** $r=228 \times 10^9$ m $v=24 \times 10^3$ m/s $m=6.42 \times 10^{23}$ kg

Utilizzare i metodi `Modified()` e `Update()` per eseguire il *refresh* del Canvas ogni N iterazioni (e.g. $N=30$) ed ottenere così un'animazione del moto dei pianeti (a questo proposito vedere anche la slide successiva)

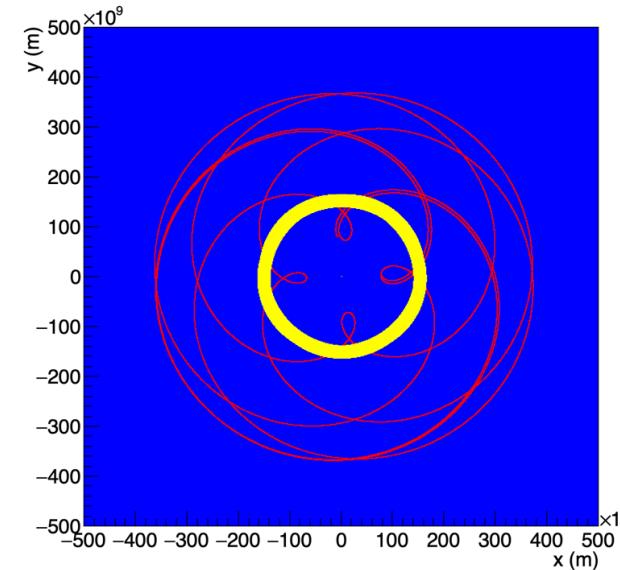
Esercizio 3

Modificare infine il codice del `main` per dare all'utente la possibilità di scegliere in quale sistema di riferimento (**SdR**) vuole rappresentare le orbite dei corpi celesti

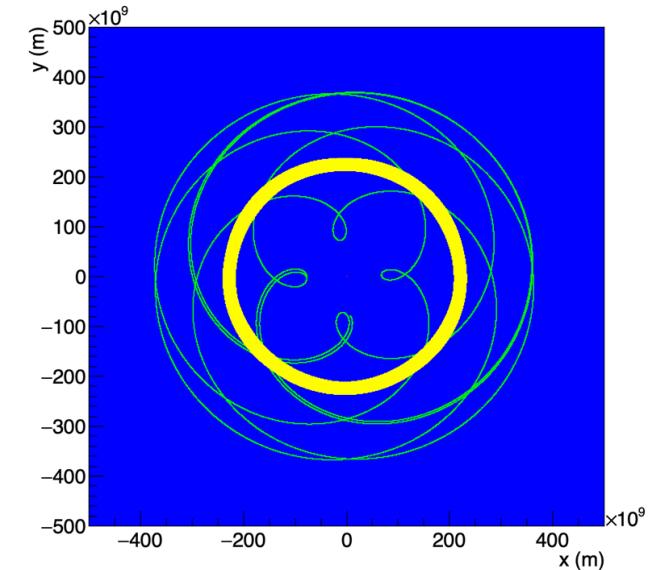
SdR: Sole



SdR: Terra



SdR: Marte

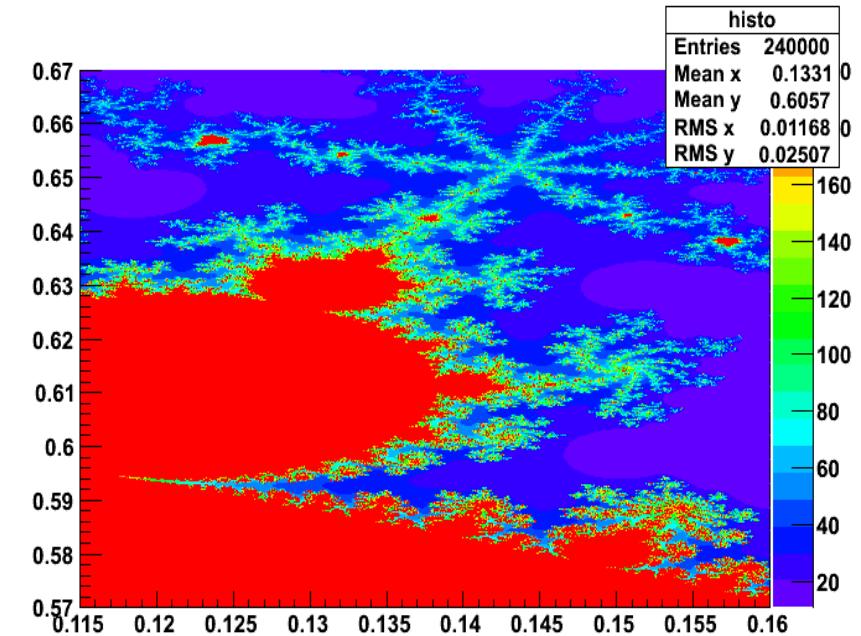


N.B.: per tenere fissato il *range* degli assi durante l'esecuzione del programma è necessario invocare i seguenti metodi nella sequenza:
`SetRangeUser` (*applicato a entrambi gli assi*), `Modified()`, `SetLimits` (*applicato a entrambi gli assi*), `Modified()`, `Update()`

Esercizio 4

Scrivere un codice per rappresentare la struttura frattale dell'insieme di Mandelbrot

- Si consideri il piano dei numeri complessi nel *range* $[-2, 1] \times [-1, 1]$
- Ad ogni punto c si associa un numero n che quantifichi la rapidità della divergenza della successione $z_{n+1} = z_n^2 + c$, con $z_0 = 0 \rightarrow$ per ogni c valutiamo a quale iterazione ***n*** si verifica la condizione $|z_n| > 2$
- Rappresentare graficamente i valori di ***n*** ottenuti campionando una fitta griglia di numeri complessi nella porzione di piano $[-2, 1] \times [-1, 1]$, utilizzando un istogramma 2D (consultare le slide seguenti per maggiori dettagli sulla definizione di istogrammi **TH2F** di ROOT)



Esercizio 4

- Scegliere un numero massimo di iterazioni $M = 100$
- Definire un `TH2F` nel *range* $[-2, 1] \times [-1, 1]$, in modo che i bin siano quadratini di lato 0.005
- In un doppio ciclo `for` considerare, di volta in volta, ciascun punto al centro dei bin del `TH2F`:

`h2->GetXaxis()->GetBinCenter(i);`

restituisce il centro dell' i -esimo bin sull'asse x (analogo per l'asse y)

- Inizializzato $z_0 = 0$, calcolare i valori della successione ricorrente:
$$z_{n+1} = z_n^2 + c$$
 (usare la classe dei numeri complessi)
e interrompere il ciclo quando si verifica la condizione $|z_n| > 2$
- Incrementare il bin corrispondente con un “peso” pari al numero n in cui il ciclo si è interrotto. È possibile usare il metodo:

`Fill(double x, double y, double n);`

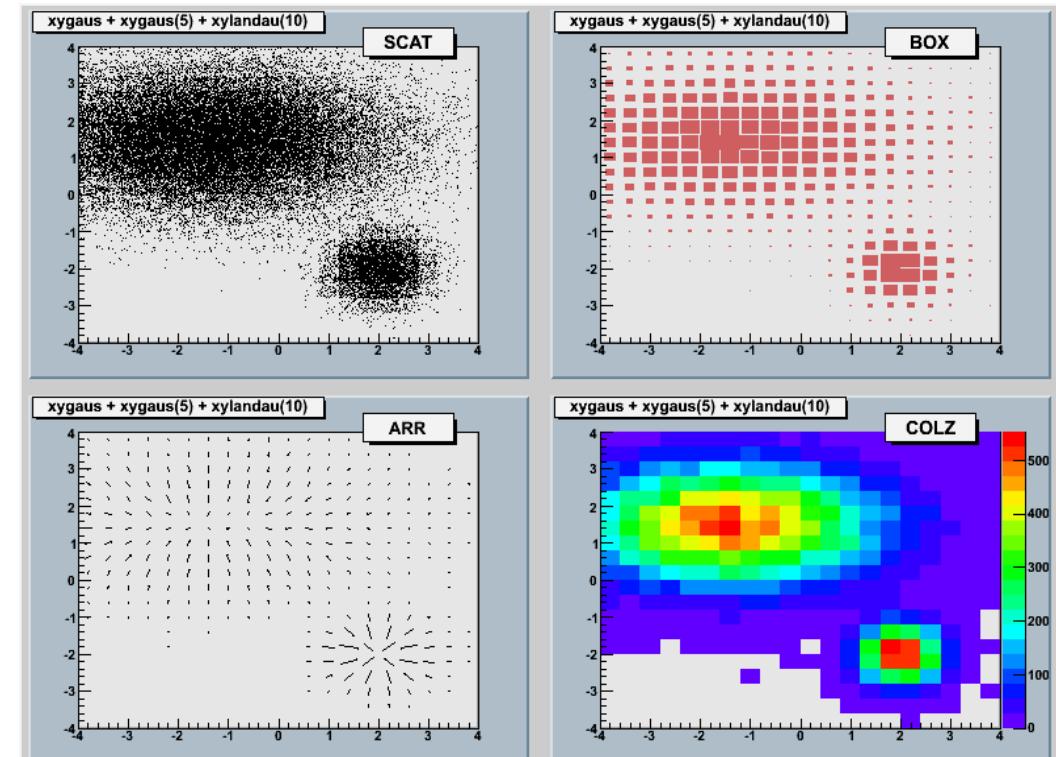
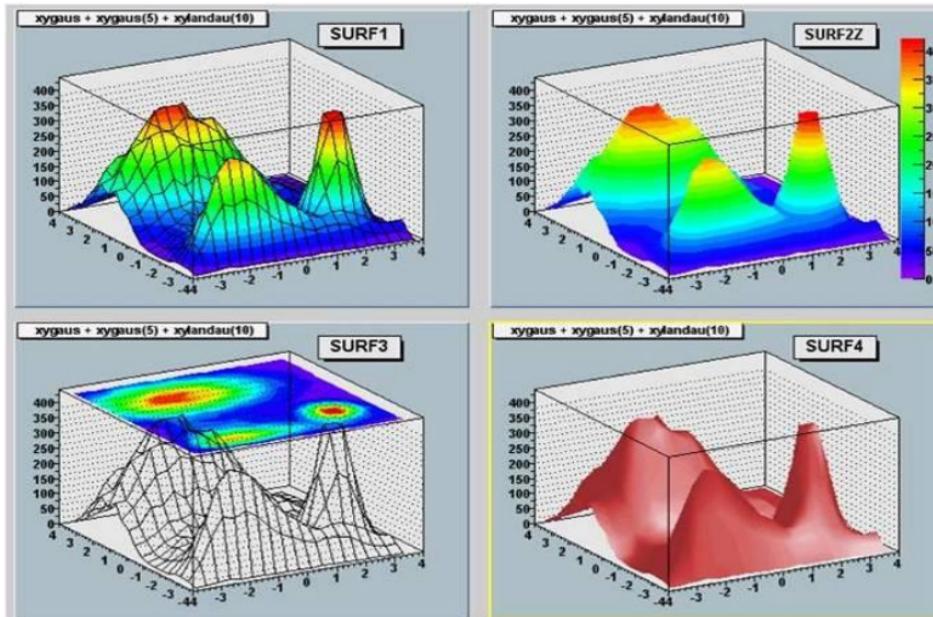
in cui n è il valore con cui si vuole incrementare il bin

Iistogrammi 2D (TH2)

```
#include "TH2.h"
```

In un TH2 ci sono **due variabili indipendenti** (rappresentate nel piano xy) → ogni **bin** è un **rettangolo**

I conteggi contenuti in ciascun bin sono rappresentati sull'asse z



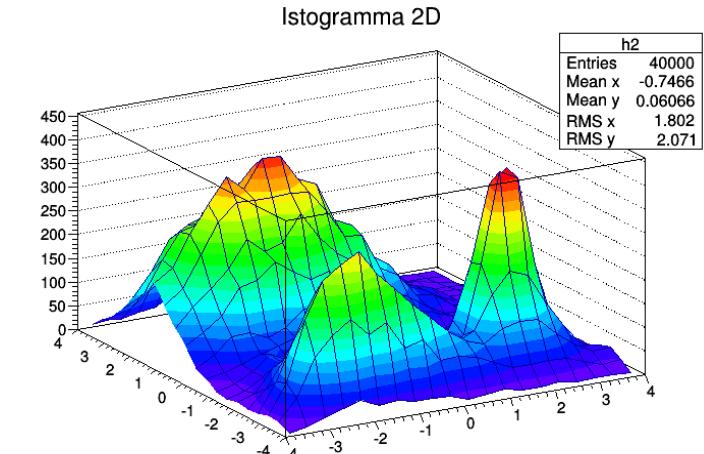
Definire un TH2F

Costruttori: ce ne sono di diversi (come per i TH1F), ad esempio:

- TH2F(`const char* name, const char* title, int nbinsx, double xlow, double xup, int nbinsy, double ylow, double yup`)
- TH2F(`const char* name, const char* title, int nbinsx, const double* xbins, int nbinsy, const double* ybins`)

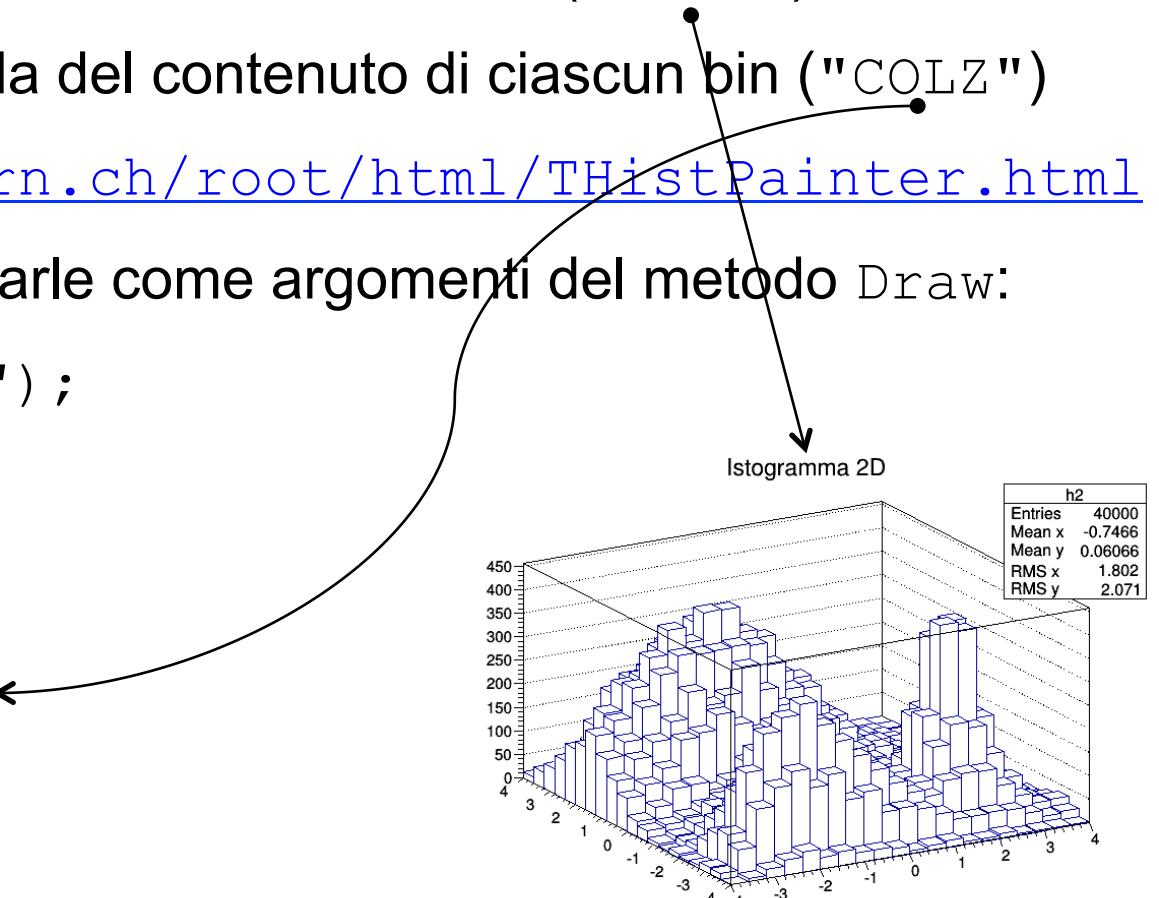
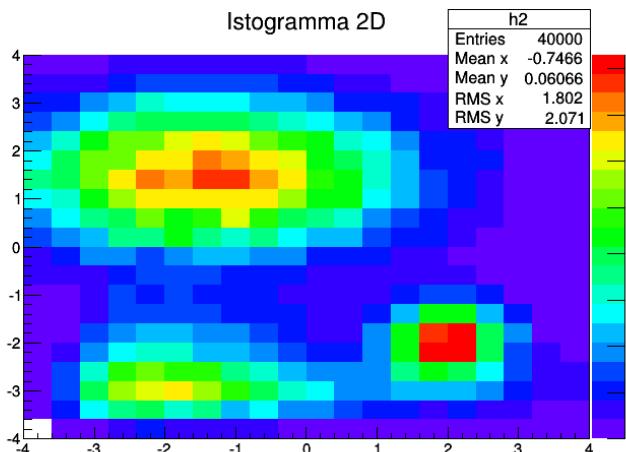
Per **riempire** un istogramma è possibile usare il metodo **Fill(x,y)**, al quale si passano le coordinate del punto in corrispondenza del quale si vuole incrementare il bin di una unità

```
TH2F* h1 = new TH2F("h1","Istogramma 2D",20,-4,4,20,-4,4);
ifstream in("DatiTH2F.txt");
double xrandom, yrandom;
while (true)
{
    in >> xrandom >> yrandom;
    if (in.eof() == true) break;
    h1->Fill(xrandom,yrandom);
}
h1->Draw("SURF1");
```



Disegnare un TH2F

- Esistono diverse opzioni grafiche per rappresentare un **TH2F**:
 - Densità di punti diversa a seconda del contenuto del bin ("SCAT")
 - Contenuto del bin rappresentato sull'asse z ("LEGO")
 - Colori diversi a seconda del contenuto di ciascun bin ("COLZ")
 - ... <http://root.cern.ch/root/html/THistPainter.html>
- Per usarle, basta specificarle come argomenti del metodo `Draw`:
 - `h2->Draw ("option") ;`



Approfondimenti consigliati

Per avere una più profonda conoscenza del linguaggio C++ si consiglia di approfondire i seguenti argomenti:

- C++
 - plugins
 - multithreading
- C++11
 - passare una funzione ad un'altra funzione
 - *lambda function*
- **Programmazione ad oggetti** (*cfr. Lezione 5* e l'esercizio nelle prossime slide)

Esercizio polimorfismo & ereditarietà

Sino ad ora è stato mostrato come implementare una delle tre caratteristiche della programmazione ad oggetti, l'**incapsulamento**. Vediamo ora tramite un esercizio guidato come è possibile implementare le altre due fondamentali caratteristiche: **ereditarietà** e **polimorfismo**

Problema: nel corso delle lezioni abbiamo incontrato due classi per la gestione di istogrammi: `TH1D.h` e `istogramma.h`, la prima implementata in ROOT, la seconda realizzata da noi. Le due classi possiedono metodi simili, ma ognuna li implementa a suo modo (pensiamo per esempio al metodo `Print()`). Sarebbe bello poter **realizzare un programma per l'istogrammazione dei dati senza dover far esplicito riferimento a quale delle due classi si vuole usare**. Per esempio si delega la scelta all'utente, che a **runtime** decide quale delle due implementazioni usare. Questo programma è realizzabile grazie alle caratteristiche di polimorfismo ed ereditarietà della programmazione ad oggetti. Vediamo ora come scrivere questo programma passo passo ...

Esercizio polimorfismo & ereditarietà

Innanzi tutto è necessario creare quella che viene chiamata un'**interfaccia virtuale**. L'interfaccia virtuale non è nient'altro che una classe (detta anche “*base class*” o “*abstract class*”) che contiene solo i prototipi dei metodi e delega l'implementazione di questi ultimi alle classi figlie (“*derived class*”) che ereditano da lei

Per fare ciò il C++ ci mette a disposizione l'attributo **virtual**, e.g.:

```
class A
{
public:
    virtual void foo() = 0;
};

class B: public A
{
public:
    void foo()
    {
        // Implementazione di foo()
    }
};
```

La classe `A` definisce il metodo **puramente virtuale** `foo()`

La classe `B`, che eredita dalla classe `A`, **DEVE** implementare il metodo `foo()`

Nota: i metodi posso essere **virtuali** oppure **puramente virtuali**. I metodi puramente virtuali sono, come in questo caso, definiti tramite `virtual ... = 0;`; I metodi virtuali invece sono definiti solo tramite l'attributo `virtual`, e ne è richiesta l'implementazione nella classe madre. In questo modo è a discrezione delle classi figlie se implementarne una loro versione oppure no

Esercizio polimorfismo & ereditarietà

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

class A
{
public:
    virtual void foo() = 0;
    virtual void bar()
    {
        cout << __PRETTY_FUNCTION__
        << endl;
    }
};

class B: public A
{
public:
    void foo()
    {
        cout << __PRETTY_FUNCTION__
        << endl;
    }

    void bar()
    {
        cout << __PRETTY_FUNCTION__
        << endl;
    }
};

void bar()
{
    cout << __PRETTY_FUNCTION__
    << endl;
}

int main ()
{
    B b;
    b.foo();
    b.bar();

    return 0;
}
```

- La classe A definisce il metodo **virtuale** bar(). La classe B, che eredita dalla classe A, non deve necessariamente implementare il metodo pippo()
- Provate il codice qui riportato commendando e/o modificando opportunamente le righe per capirne il funzionamento

Esercizio polimorfismo & ereditarietà

A questo punto possiamo iniziare a realizzare l'interfaccia virtuale alle classi istogramma. Nel file `interfacciaIstogramma.h` definiamo la classe `interfacciaIstogramma` con i seguenti metodi **puramente virtuali**:

```
Fill           (const double& value)  
Print          () const  
GetMean        () const  
GetRMS         () const  
GetMax         () const  
GetIntegral    () const  
GetBinCenter   (int i) const  
GetBinContent  (int i) const
```

Inoltre definiamo in maniera **virtuale** l'operator= e il distruttore

Nota sui distruttori: è buona norma definirli di tipo **virtual** (non puramente virtuale) per evitare *memory leak* quando si chiama il distruttore di una classe figlia

Esercizio polimorfismo & ereditarietà

```

#include <iostream>
using namespace std;

class A
{
public:
    virtual ~A()
    {
        cout << __PRETTY_FUNCTION__
           << endl;
    }
};

class B: public A
{
public:
    ~B()
    {
        cout << __PRETTY_FUNCTION__
           << endl;
    }
};

class C: public B
{
public:
    ~C()
    {
        cout << __PRETTY_FUNCTION__
           << endl;
    }
};

}

};

int main ()
{
    A* a = new B();
    delete a;

    cout << endl;

    A* aa = new C();
    delete aa;

    cout << endl;

    B b;

    return 0;
}

```

Provate il codice qui riportato
commendando e/o modificando
opportunamente le righe per
cavarne il funzionamento

Esempio di **distruttore virtuale** e di come la
distruzione si propaga dalle classi figlie alla classe
madre: se non avessimo definito virtuale il distruttore
di A, l'istruzione `delete a;` chiamerebbe solo il
distruttore di A e non quello delle classi figlie, non
liberando così la memoria che potenzialmente
avrebbero potuto allocare le classi figlie

Esercizio polimorfismo & ereditarietà

Ora che abbiamo scritto la nostra interfaccia virtuale alle classi istogramma è necessario apportare una piccola modifica alle classi che implementano gli histogrammi. Iniziamo da **istogramma.h**: dobbiamo fare in modo che erediti da **interfacciaIstogramma.h**, per fare questo dobbiamo semplicemente modificare l'intestazione della classe

```
class istogramma: public interfacciaIstogramma
```

Dovremmo fare la stessa cosa nella classe **TH1D.h**, ma siccome la classe **TH1D.h** fa parte di ROOT, dovremmo ricompilare tutto ROOT perché la modifica abbia effetto. Per evitare di dover attendere qualche ora per la ricompilazione di ROOT scegliamo una strada più furba che anche in questo caso fa uso di una delle caratteristiche della programmazione ad oggetti, **l'ereditarietà e la composition**

- Creiamo una classe “wrapper” della classe **TH1D.h** (**TH1Dwrapper.h**) che farà da interfaccia alla classe **TH1D.h**. In altre parole faremo in modo di riferirci a **TH1Dwrapper.h** come se ci stessimo riferendo a **TH1D.h**, quindi TH1Dwrapper conterrà un costruttore che istanzia un **TH1D** ed avrà come metodi gli stessi di **istogramma.h** ma implementati in maniera opportuna su un **TH1D**

Esercizio polimorfismo & ereditarietà

```
#ifndef TH1Dwrapper_h
#define TH1Dwrapper_h

#include <iostream>
#include "TH1D.h"
#include "TFile.h"

#include "interfacciaIstogramma.h"

class TH1Dwrapper:
    public interfacciaIstogramma
{
public:
    // Default constructor
    TH1Dwrapper ();

    // Constructor
    TH1Dwrapper (const int& nBin,
                 const double& min,
                 const double& max);

    // Copy constructor
    TH1Dwrapper (const TH1Dwrapper& original);

    // Operator=
    TH1Dwrapper& operator= (const TH1Dwrapper& original);

    // Destructor
    ~TH1Dwrapper () ;

    // Metodi
    int Fill (const double& value);
    void Print () const;
    double GetMean () const;
    double GetRMS () const;
    int GetMax () const;
    double GetIntegral () const;
    double GetBinCenter (int i) const;
    int GetBinContent (int i) const;

    // Attributi
private:
    TH1D* ROOThisto_p;
};
```

TH1Dwrapper eredita da interfacciaIstogramma

Relazione di tipo composition tra la classe TH1Dwrapper e TH1D

- Prototipo della classe **TH1Dwrapper.h**
- Realizzare l'implementazione di **TH1Dwrapper.h**, i.e. **TH1Dwrapper.cc**

Esercizio polimorfismo & ereditarietà

Oltre alle relazioni di ereditarietà tra le classi, è possibile creare altre dette: **association**, **aggregation**, e **composition**. La relazione risulta più “forte” passando dalla prima alla terza. Illustriamo brevemente tramite un programma cosa si intende:

```
class B
{
public:
    B() {}
};

// Association
class A
{
public:
    A(B b) : myB(b) {}

private:
    B myB;
};

// Aggregation
class A
{
```

L'oggetto che verrà istanziato di tipo A e l'oggetto che verrà istanziato di tipo B sono completamente indipendenti

Se l'oggetto che verrà istanziato di tipo B dovesse venire distrutto verrebbe perso anche da A

```
public:
    A(B* b) : myB(b) {}

private:
    B* myB;

};

// Composition
class A
{
public:
    A()
    {
        b = new B();
    }

private:
    B* b;
};
```

L'oggetto che verrà istanziato di tipo A possiederà completamente anche l'oggetto istanziato di tipo B

Esercizio polimorfismo & ereditarietà

Vediamo ora cosa implementare nel **main** program

Abbiamo come obiettivo quello di scrivere un codice che manipola un istogramma (lo riempie, lo visualizza, ne fa alcune manipolazioni elementari, e.g. calcolo di RMS, mean, etc...) indipendentemente dal tipo di implementazione dell'istogramma stesso (**istogramma.h** oppure **TH1D.h**). Per fare questo dovremo fare riferimento all'interfaccia virtuale creata in precedenza, **interfacciaIstogramma.h**

Dovremo innanzitutto inglobare in una funzione tutte le manipolazioni che vogliamo eseguire sull'istogramma. La funzione dovrà ricevere un puntatore di tipo **interfacciaIstogramma** e chiamare i metodi di questa interfaccia per manipolare l'istogramma, e.g.:

```
void manipHisto (interfacciaIstogramma* histo, const double min,  
const double max, const int N)  
{  
    // Riempimento  
    // Iistogrammazione  
    // RMS, mean, etc...  
}
```

Si consiglia di prendere come programma di partenza il **main** program realizzato per testare la classe **istogramma** (**cfr.** **Lezione 5**)

Esercizio polimorfismo & ereditarietà

Se ci fosse la necessità di chiamare un metodo specifico presente solo in una delle *derived class*, per esempio in `TH1D.h`, e non in `istogramma.h`, e quindi nemmeno nella *base class* `interfacciaIstogramma.h`, il C++ mette a disposizione un **cast** fatto apposta per gestire le conversione all'interno della scala dell'ereditarietà tra classi, l'istruzione **dynamic_cast**:

...

```
TH1D* tmpTH1D = dynamic_cast<TH1D*>(histo);
if (tmpTH1D != nullptr)
{
    int color = tmpTH1D->GetAxisColor();
    ...
}
```

...

Il **dynamic_cast** si usa per convertire puntatori (o referenze) ad oggetti, in altri tipi di puntatori (o referenze) ad altri oggetti, legati tra loro da una relazione di ereditarietà

Esercizio polimorfismo & ereditarietà

Nel main program verrà creata la variabile histo di tipo puntatore a interfacciaIstogramma, i.e. `interfacciaIstogramma*` histo;

L'istanza concreta di histo verrà scelta dall'utente tramite argv:

```
...
if (std::strcmp(argv[1], "1") == 0)
    histo = new istogramma(nBin, min, max);
else if(std::strcmp(argv[1],"2") == 0)
    histo = new TH1Dwrapper(nBin, min, max);

manipHisto(histo,min,max,N);
...
```

E` da notare che **non è possibile istanziare un oggetto puramente virtuale**, cioè non è possibile scrivere: `interfacciaIstogramma h();` e nemmeno `interfacciaIstogramma* h = new interfacciaIstogramma();`

Si può fare riferimento ad una classe puramente virtuale solo tramite puntatori o referenze come abbiamo fatto in `manipHisto`

Esercizio polimorfismo & ereditarietà

A questo punto proviamo a compilare il codice ...

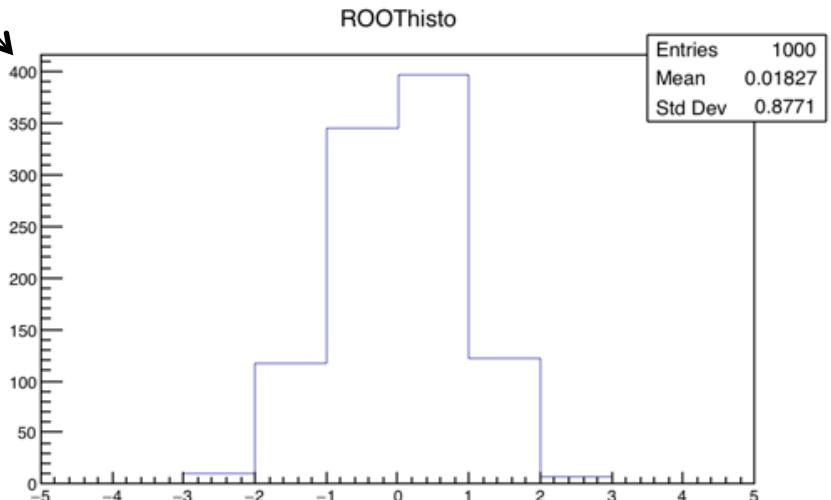
Ora dovremmo avere un programma funzionante che implementa tutte le caratteristiche della programmazione ad oggetti: **incapsulamento, ereditarietà e polimorfismo!**

Provate il codice commettendo e/o modificando opportunamente le righe per capirne il funzionamento

- Output con l'opzione “2”, i.e. usando `TH1Dwrapper.h`
- Output con l'opzione “1”: i.e. usando `istogramma.h`

```
Inserisci gli estremi dell'intervallo [a,b) in cui generare i numeri: -5 5
Inserisci quanti numeri casuali vuoi generare: 1000
Inserisci gli estremi dell'istogramma [min,max): -5 5
Inserisci il numero di bin dell'istogramma: 10
Mean = 0.018268
RMS = 0.87712
Max counts = 397
Integral [-5, 5] = 1000
+----->
-5.00|0
-4.00|0
-3.00|#11
-2.00|#####
-1.00|#####
0.00|#####
1.00|#####
2.00|7
3.00|1
4.00|0
|
```

```
Inserisci gli estremi dell'intervallo [a,b) in cui generare i numeri: -5 5
Inserisci quanti numeri casuali vuoi generare: 1000
Inserisci gli estremi dell'istogramma [min,max): -5 5
Inserisci il numero di bin dell'istogramma: 10
Mean = 0.018268
RMS = 0.87712
Max counts = 397
Integral [-5, 5] = 1000
[TH1Dwrapper::Print] Histogram written into the file ROOTHisto.root
```



Esercizio polimorfismo & ereditarietà

La risoluzione dell'esercizio mediante la tecnica della *composition* delle classi non è l'unica strada percorribile. E' infatti possibile risolvere l'esercizio con la pura ereditarietà. Creiamo una seconda classe "wrapper" della classe TH1D.h (**TH1DwrapperInherit.h**)

```
#ifndef TH1DwrapperInherit_h
#define TH1DwrapperInherit_h

#include <iostream>
#include "TH1D.h"
#include "TFile.h"
#include "interfacciaIstogramma.h"

class preTH1DwrapperInherit:
public interfacciaIstogramma
{
public:
    int GetBinContent(int i) const
    { return helperFunction(i); }
protected:
    virtual int helperFunction(int i) const = 0;
};

class TH1DwrapperInherit:
public preTH1DwrapperInherit, public TH1D
{
public:
```

```
    public:
        // Default constructor
        TH1DwrapperInherit () {};

        // Constructor: direct call to a
        // specific base class constructor (the
        // base class default constructor would
        // be called automatically)
        TH1DwrapperInherit (const int& nBin,
                           const double& min, const double&
                           max) : TH1D("ROOTHisto", "ROOTHisto",
                           nBin, min, max) {};

        // Copy constructor
        TH1DwrapperInherit (const
                           TH1DwrapperInherit& original);

        // Operator=
        TH1DwrapperInherit& operator= (const
                                         TH1DwrapperInherit& original);
}
```



Multiple inheritance è permessa in C++

Esercizio polimorfismo & ereditarietà

```
// Destructor
~TH1DwrapperInherit () {};

// Metodi
int Fill          (const double& value);
void Print        () const;
double GetMean    () const;
double GetRMS     () const;
int GetMax        () const;
double GetIntegral () const;
double GetBinCenter (int i) const;

protected:
    int helperFunction(int i) const { return static_cast<int>(
        this->TH1D::GetBinContent(i)); }
};

#endif
```

E` interessante notare che il metodo GetBinContent nella classe **TH1D.h** è definito come:

```
virtual double GetBinContent(int i) const;
```

Mentre in **interfacciaIstogramma.h** è definito come:

```
virtual int GetBinContent(int i) const = 0;
```

Esercizio polimorfismo & ereditarietà

Quando una classe figlia pretende di nominare un metodo con lo stesso nome di quello della classe madre, il metodo deve obbligatoriamente possedere anche lo stesso prototipo se è definito di tipo `virtual` nella classe madre. Quindi in `TH1DwrapperInherit.h`, se avesse ereditato da `interfacciaIstogramma.h`, oltre che da `TH1D.h`, avremmo dovuto scrivere:

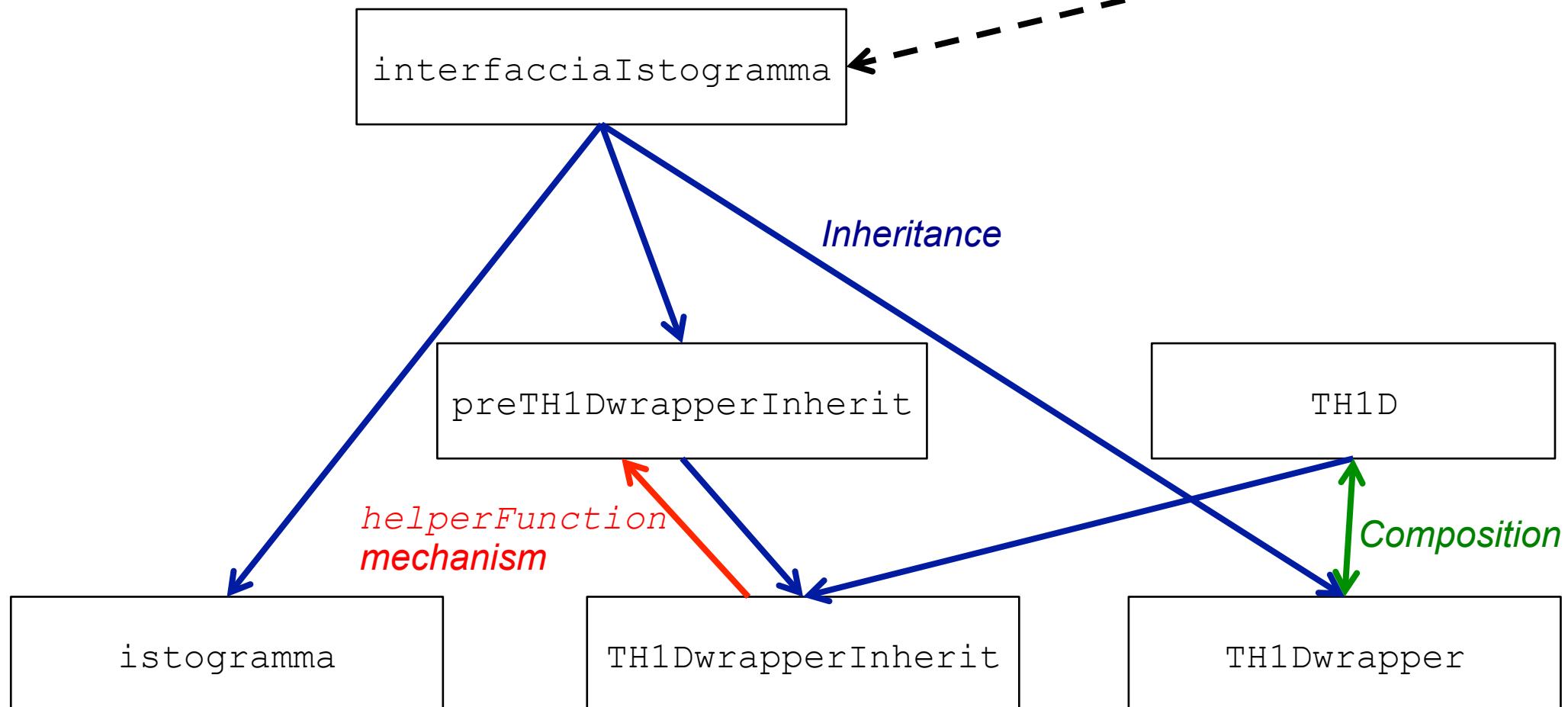
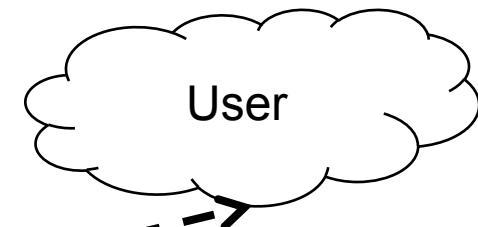
```
double GetBinContent(int i) const;
```

ma questo contrasta con il prototipo di `GetBinContent` in `interfacciaIstogramma.h`. Abbiamo quindi dovuto usare l'espeditivo adottato nella `class preTH1DwrapperInherit` e cioè la classe intermedia `preTH1DwrapperInherit` implementa la `GetBinContent` nel modo richiesto da `interfacciaIstogramma`. Questa trasmette poi il risultato corretto tramite il meccanismo implementato con la `helperFunction`

Realizzare l'implementazione di `TH1DwrapperInherit.h`, i.e. `TH1DwrapperInherit.cc`

Esercizio polimorfismo & ereditarietà

Vediamo ora riassunto in un diagramma la relazione che sussiste tra le diverse classi che abbiamo implementato



L'attributo **friend**

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

class B;

class A
{
public:
    A(int id);
    void showB(B b);
private:
    int id;
};

class B
{
    friend class A; // class A can access all
    public, private and protected attributes of B
public:
    B(int id)
    {
        cout << __PRETTY_FUNCTION__ << endl;
        this->id = id;
    }
    void showA(A a)
    {
        // cout << "A::id " << a.id << endl;
    }
}
```

Oltre a poter definire una classe come “*figlia*”, è possibile definirla anche come “*amica*” mediante l’attributo **friend**. Questo semplicemente consente alla classe A di accedere a tutti gli attributi della classe B. **N.B.:** non è ne` una proprietà transitiva, ne` reciproca

```
private:
    int id;
};

A::A(int id)
{
    cout << __PRETTY_FUNCTION__ << endl;
    this->id = id;
}
void A::showB(B b)
{
    cout << "B::id " << b.id << endl;
}

int main ()
{
    A a(1);
    B b(2);

    a.showB(b);
    b.showA(a);
    return 0;
}
```

Provate il codice qui riportato commentando e/o modificando opportunamente le righe per capirne il funzionamento