

---

# Classi in C++: incapsulamento, data hiding ed ereditarietà

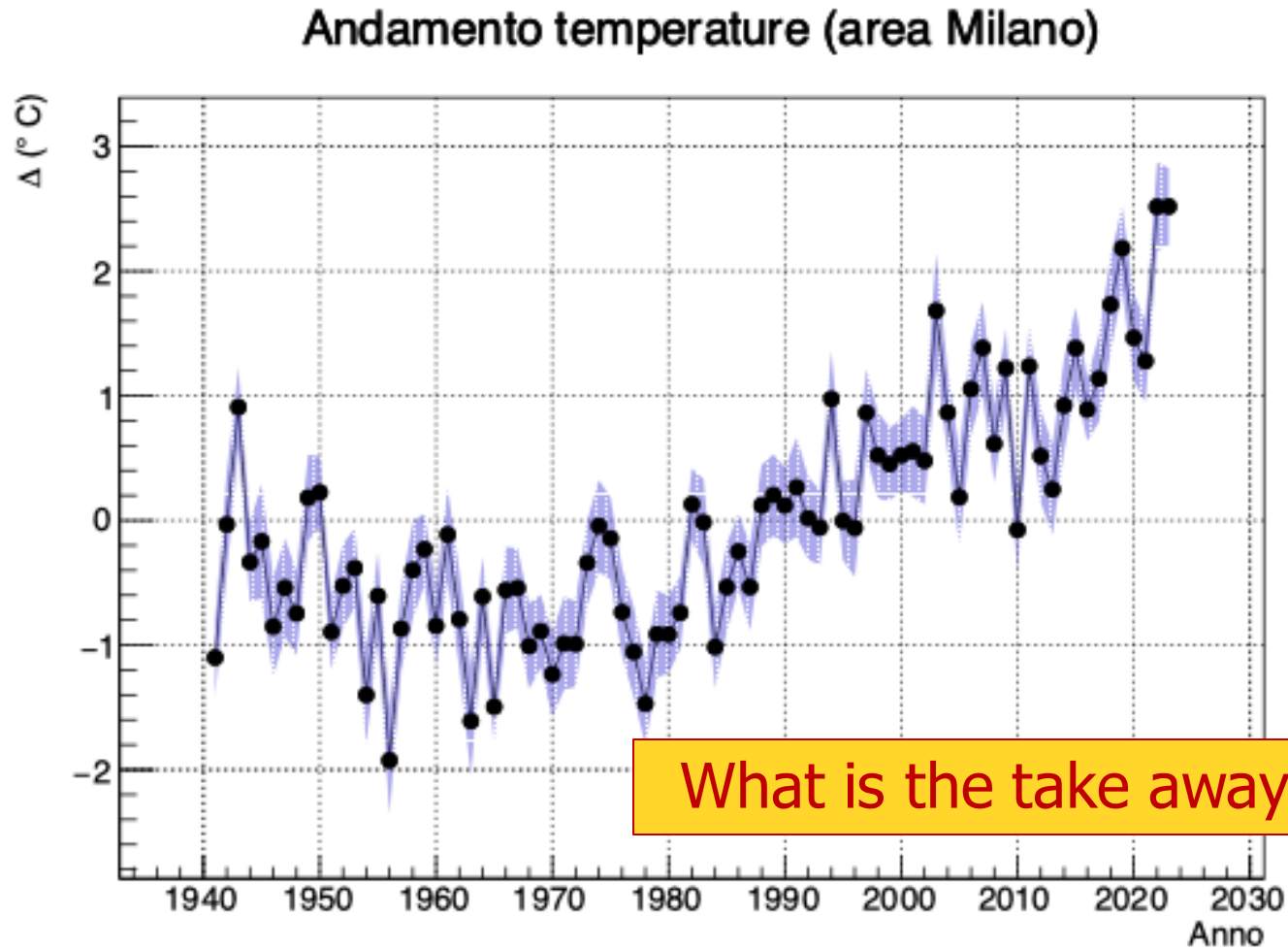
Laboratorio Trattamento Numerico dei Dati Sperimentali

Prof. L. Carminati  
Università degli Studi di Milano

- ❑ Evaluate the evolution with time of major climatic indicators ( eg. Temperature)
- ❑ Use [ERA5](#) re-analysis data :
  - ❑ “Reanalysis combines model data with observations from across the world into a globally complete and consistent dataset using the laws of physics. This principle, called data assimilation, is based on the method used by numerical weather prediction centres, where every so many hours (12 hours at ECMWF) a previous forecast is combined with newly available observations in an optimal way to produce a new best estimate of the state of the atmosphere, called analysis, from which an updated, improved forecast is issued. Reanalysis works in the same way, but at reduced resolution to allow for the provision of a dataset spanning back several decades. Reanalysis does not have the constraint of issuing timely forecasts, so there is more time to collect observations, and when going further back in time, to allow for the ingestion of improved versions of the original observations, which all benefit the quality of the reanalysis product”
- ❑ Average temperature of air at 2m above the surface available for each day since 1941.
- ❑ Data has been re-gridded to a regular (lat,lon) grid of 0.25 degrees for the reanalysis
- ❑ Reference portal : <https://open-meteo.com/>

- ❑ Goal of the analysis : estimate the evolution of the temperature of air at 2m above the surface in the Milano area from 1941 to 2023.
- ❑ What we have done :
  - ❑ For each day of the year compute the average temperature in the 1941-2023 time-range
  - ❑ For each day of the year compute the difference ( $\Delta$ ) between the actual value of the temperature and the average over 1941-2023 time-range in this day.
  - ❑ For each year we produced a file with the  $\Delta$ s in each day ( a column with 365 entries )
- ❑ What you will find :
  - ❑ Total of 82 files ( eg "1941.txt" )
- ❑ What you will do :
  - ❑ Compute average  $\Delta$ s ( + uncertainties ) for each year and : if no climate change, we would expect a gaussian peaked at 0
  - ❑ Draw the evolution of the average deltas with time ( + uncertainty band )

## Motivation : simple example of climate data analysis



---

## Can we conclude something ? Hypothesis test

- ❑ We have  $n$  measurements, and we want to understand if a certain model (typically a function) is an acceptable description of the measurements
- ❑ We can think of several different estimators but there's one which has interesting properties, the  $\chi^2$

$$\chi^2 \equiv \sum_{i=1}^n \frac{(x_i - x_{ti})^2}{\sigma_i^2}$$

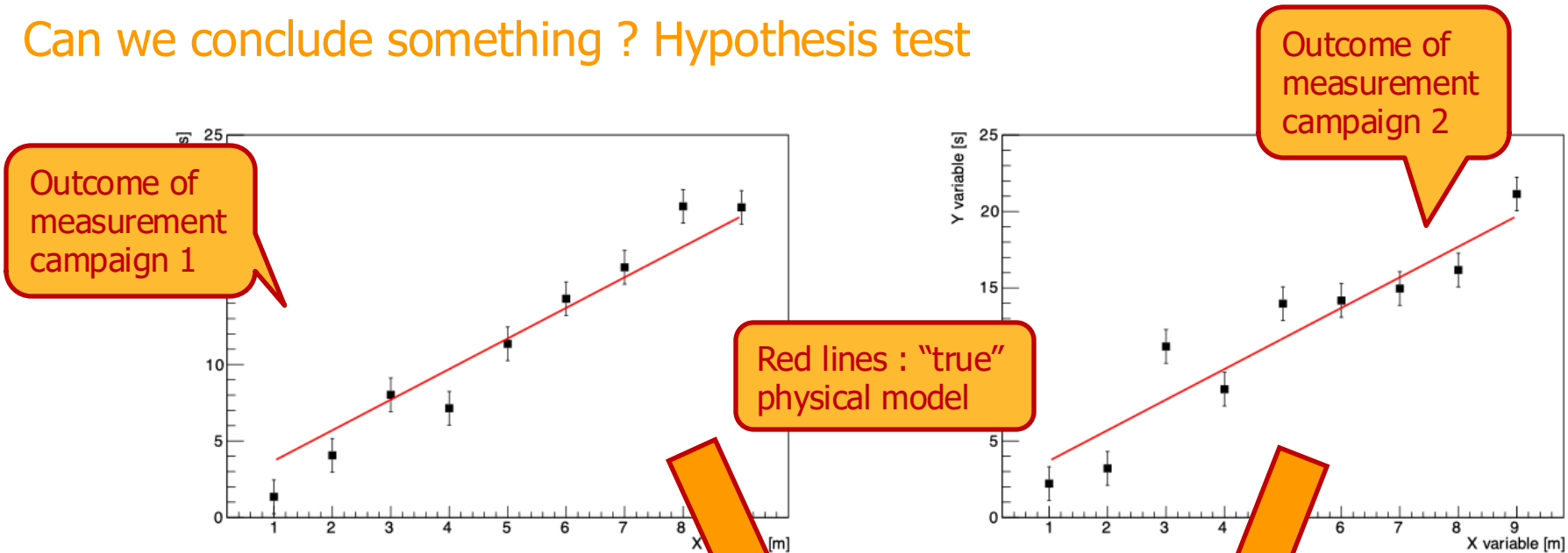
where  $x_i$  are the measured values,  $x_{ti}$  are the value predicted by the model,  $\sigma_i$  are the uncertainties on the measured value and  $n$  is the number of measurements

- ❑ Why  $\chi^2$  ? Because we know the distribution of this variable !
  - ❑ If data really follows the model then the  $\chi^2$  follows a well defined distribution : the  $\chi^2$  estimator is distributed as the following variable where  $k$  is  $n - (\text{\# of parameters of the model})$  and  $x_i$  are independent variables distributed as  $N(0,1)$

$$\chi^2(k) = \sum_{i=1}^k x_i^2 = x_1^2 + \dots + x_k^2$$

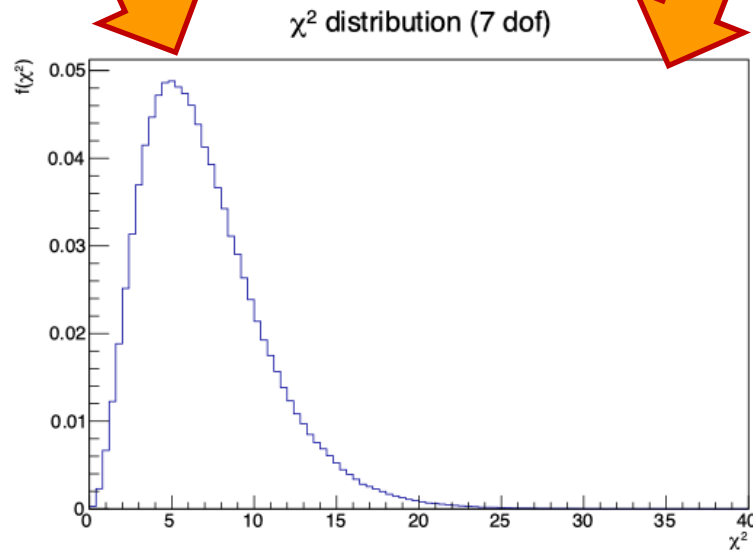
- ❑ We can use its distribution to assess the goodness of the model

## Can we conclude something ? Hypothesis test



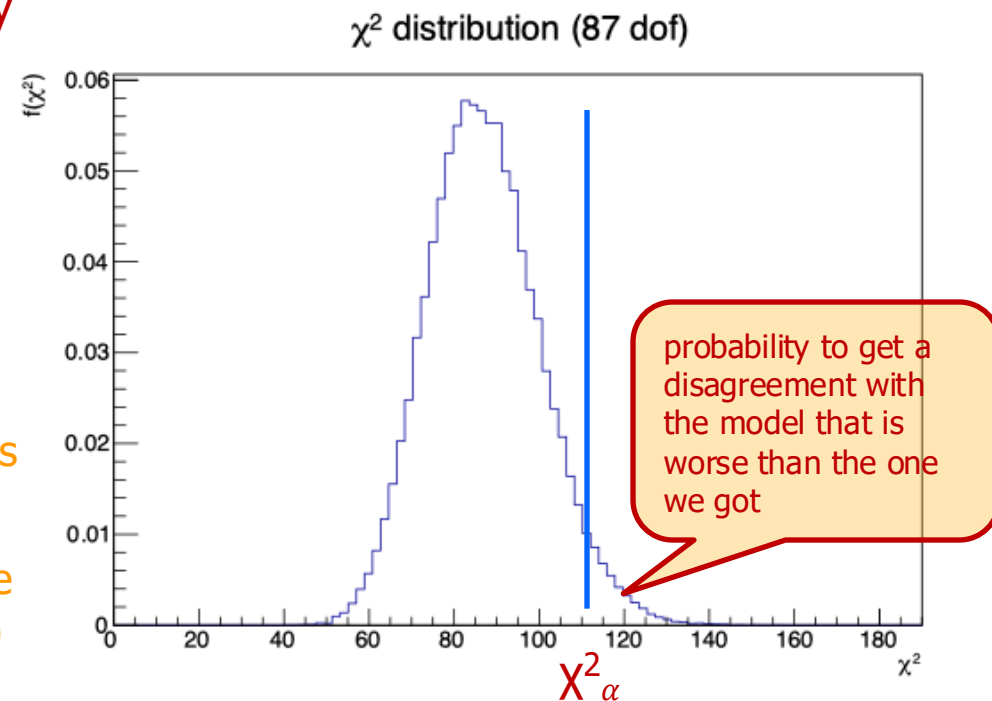
Assuming the physical phenomenon under study is really described by a linear relation (red line)

- ❑ each measurement campaign will give you a different set of points and a different  $\chi^2$
- ❑ the fantastic point is that the  $\chi^2$  distribution is known !

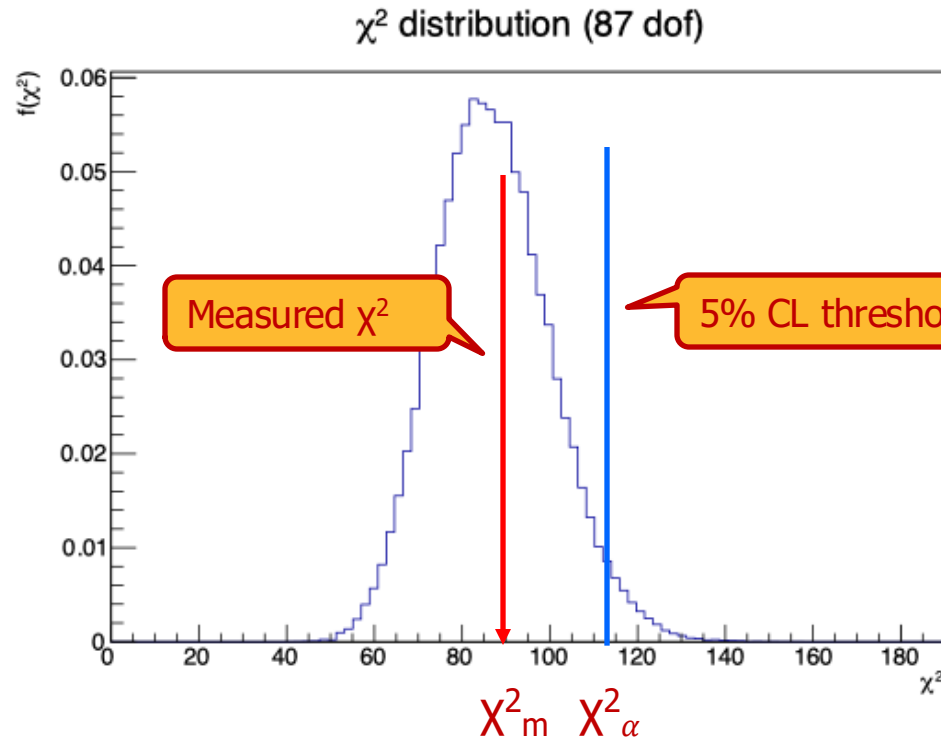


## Can we conclude something ? Hypothesis test

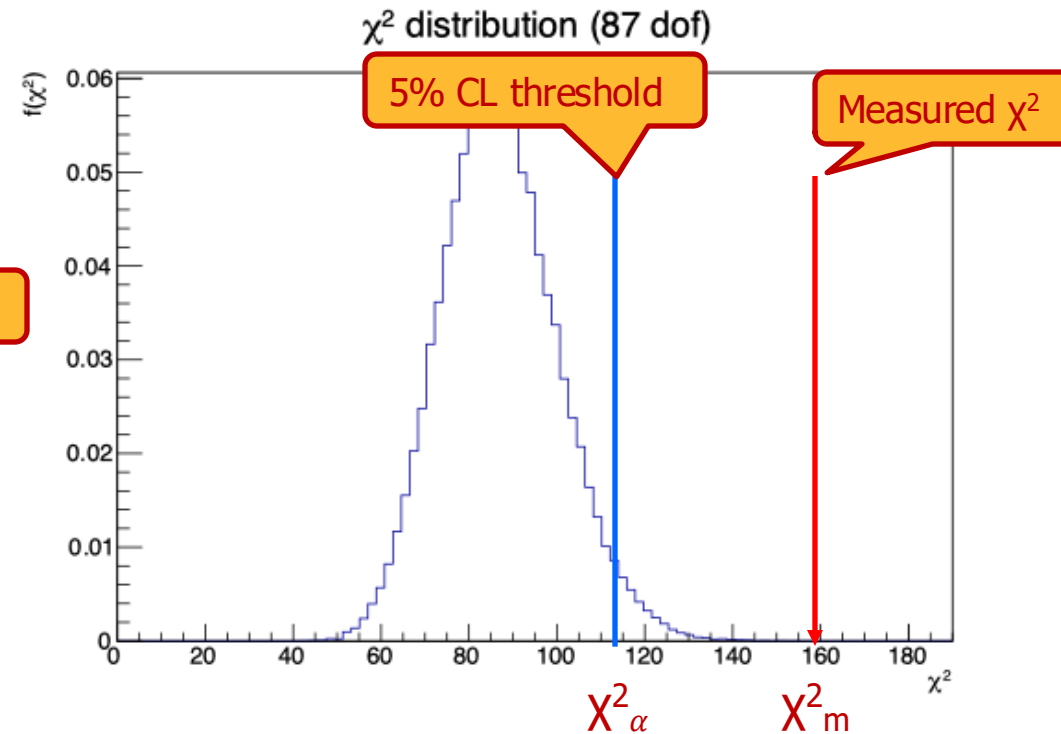
- ❑ Since we know how the  $\chi^2$  should be distributed for the correct model (typically unknown) we can design a test to check if a given model is good or bad !
- ❑ Define a null hypothesis : the dataset is described by a model  $m(x, \mathbf{p})$  where  $\mathbf{p}$  are a set of parameters
  - ❑ In our case  $m(x, \mathbf{p}) = 0$  ( meaning there's no climate change  $\Delta(\text{year})=0$  )
- ❑ We do not confirm an hypothesis we can only reject an hypothesis
  1. Consider the distribution of the estimator  $f(\chi^2)$  for the given number of dof
  2. Define a confidence level we can tolerate (typically 5%) and compute the  $\chi^2_{\alpha}$  value for which  $P(\chi^2 > \chi^2_{\alpha}) = 0.05$
  3. Compute  $\chi^2_m$  from the measured data points
  4. We reject the null hypothesis if  $\chi^2_m > \chi^2_{\alpha}$  , in other words if the probability to get a worse disagreement with the model is small (<5%)



## Can we conclude something ? Hypothesis test



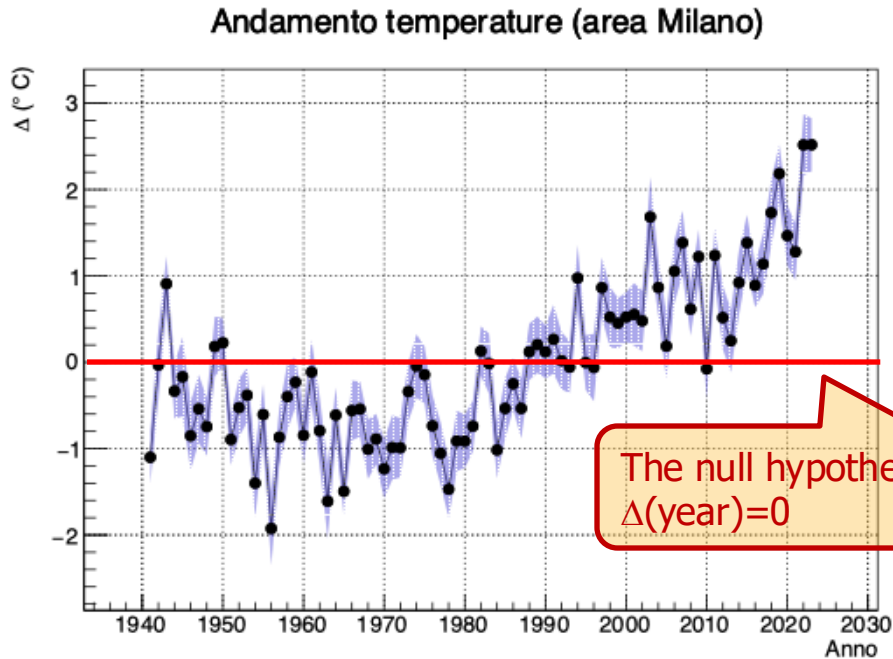
Can't reject the null hypothesis, I can't exclude the null hypothesis is correct, I can adopt it ( it doesn't mean the null hypothesis is true ! )



We can safely reject the null hypothesis, sufficient evidence in the data supports the alternative hypothesis.



## Is the temperature changing ?



The null hypothesis  
 $\Delta(\text{year})=0$

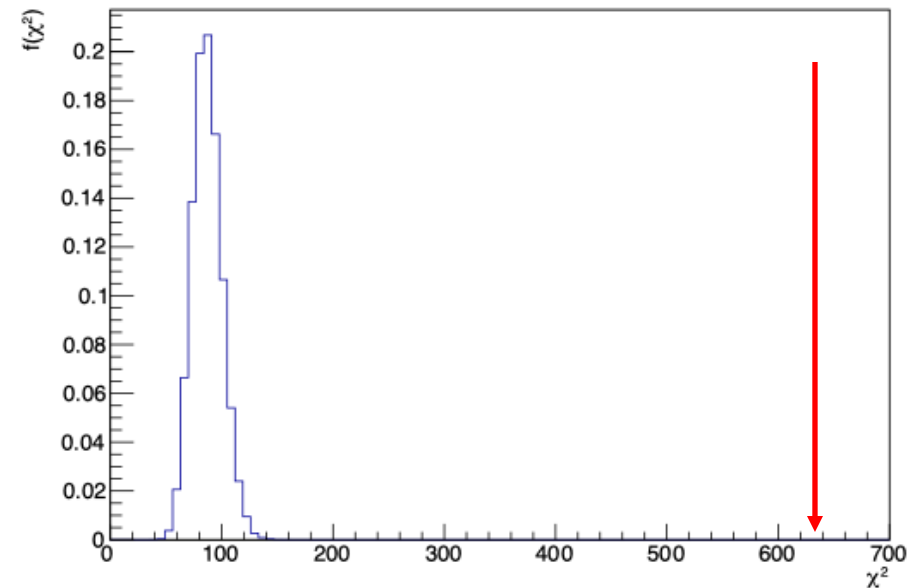
We can certainly reject the null hypothesis  
( our null hypothesis was  $\Delta(\text{year})=0$  ) :

- ☐ the  $\Delta(\text{year})$  is not 0
- ☐ Climate is changing, no doubts !

☐ More speculations need more work : no fits with empirical functions please !

- ☐ Is this variation 'natural' or anthropic ?
- ☐ What is the origin ?  $\text{CO}_2$  ?

$\chi^2$  distribution (87 dof)



## Un paio di precisazioni

- ❑ Per ogni classe che scrivete, un costruttore di default e un copy constructor di default sono sempre disponibili.
  - ❑ Nel momento in cui dichiarate e implementate un qualunque costruttore o un copy constructor quelli di default vengono rimossi
  - ❑ Se non li dichiarate e implementate userete quelli di default: spesso non c'è bisogno di sovrascriverli, tipicamente se non c'è allocazione dinamica
- ❑ Per ogni classe che scrivete un distruttore di default è sempre disponibile :
  - ❑ Dichiaratelo e implementatelo solo se necessario ( ie quando c'è da de-allocare memoria allocata dinamicamente
  - ❑ Se lo dichiarate dovete implementarlo ! ( oppure lo omettete completamente )

```
~Posizione() {} ;
```

Implementazione vuota

## Esempio la classe Posizione :

Riprendiamo la classe Posizione di cui abbiamo parlato

```
class Posizione {
public :

    Posizione() ;
    Posizione(double x , double y, double z ) ;

    double getX() const { return m_x ; } ;
    double getY() const { return m_y ; } ;
    double getZ() const { return m_z ; } ;

    double getR()      const { ... } ;
    double getPhi()    const { ... } ;
    double getTheta()  const { ... } ;
    double getRho()     const { ... } ;

    void setX(double x) { m_x = x ; } ;
    void setY(double y) { m_y = y ; } ;
    void setZ(double z) { m_z = z ; } ;

    double getDistance() const ;
    double getDistance( Posizione p ) const ;
    void printPosition() const ;

    bool operator< (const Posizione & b) const { return getDistance()<b.getDistance(); };

    Posizione operator+( Posizione p1 ) const ;
    Posizione operator/( double d ) const ;

private :

    double m_x , m_y , m_z ;
};
```

Aggiungiamo altre funzionalità alla classe : per esempio la conversione in coordinate sferiche o polari

Calcolo della distanza dall'origine o da un'altra posizione

Somma di posizioni e rapporto tra una posizione e una costante double

## Il concetto di incapsulamento e data hiding

```
#include "Posizione.h"

int main() {

    Posizione p(1,1,1);

    // accedo ad una componente

    double xval = p.getX();
    // double xval = p.m_x ; NO !!!

    // modifico una componente

    p.setX(9);
    // p.m_x = 9; NO !!

    cout << "Distanza dall'origine = " << p.GetDistance() << endl;
}
```

- ❑ Il main non può accedere direttamente ai dati della classe (i data membri sono dichiarati `private`)
- ❑ L'accesso è regolato da interfacce (costruttori, metodi `get` e `set`)
- ❑ Queste due regole di programmazione sono la base del concetto di **incapsulamento** o **data hiding**

Spesso il `main` e le classi sono scritti da persone diverse

- ❑ L'incapsulamento garantisce l'implementazione di regole nella comunicazione tra `main` e classi e riduce le possibilità di introdurre errori
- ❑ L'incapsulamento garantisce che un cambiamento all'interno della classe non abbia alcun impatto sul `main` (in termini di codifica) se le interfacce sono modificate in modo coerente

## Esempio di incapsulamento con la classe posizione :

```
#include "Posizione.h"
#include <cmath>

// costruttore di default
Posizione::Posizione() {
    m_x=0.;
    m_y=0.;
    m_z=0.;
}

// costruttore a partire da una terna cartesiana
Posizione::Posizione(double x, double y, double z) {
    m_x=x;
    m_y=y;
    m_z=z;
}

// distruttore (puo' essere vuoto)
Posizione::~Posizione() {}

// Coordinate cartesiane
double Posizione::getX() const {
    return m_x;
}

double Posizione::getY() const {
    return m_y;
}

double Posizione::getZ() const {
    return m_z;
}

// Coordinate sferiche
double Posizione::getR() const {
    return sqrt(m_x*m_x+m_y*m_y+m_z*m_z);
}

double Posizione::getPhi() const {
    return atan2(m_y,m_x);
}

double Posizione::getTheta() const {
    return acos(m_z/R());
}

// raggio delle coordinate cilindriche
double Posizione::getRho() const {
    return sqrt(m_x*m_x+m_y*m_y);
}
```

```
#include "Posizione.h"
#include <cmath>

// costruttore di default
Posizione::Posizione() {
    m_x=0.;
    m_y=0.;
    m_z=0.;
}

// costruttore a partire da una terna cartesiana
Posizione::Posizione(double x, double y, double z) {
    m_x=sqrt(x*x+y*y+z*z); // R
    m_y=atan2(y,x); // phi
    if ( m_x>0. ) m_z=acos(z/m_x); // theta
    else m_z=0.;
}

// distruttore (puo' essere vuoto)
Posizione::~Posizione() {}

// Coordinate cartesiane
double Posizione::getX() const {
    return m_x*cos(m_y)*sin(m_z);
}

double Posizione::getY() const {
    return m_x*sin(m_y)*sin(m_z);
}

double Posizione::getZ() const {
    return m_x*cos(m_z);
}

// Coordinate sferiche
double Posizione::getR() const {
    return m_x;
}

double Posizione::getPhi() const {
    return m_y;
}

double Posizione::getTheta() const {
    return m_z;
}

// raggio delle coordinate cilindriche
double Posizione::getRho() const {
    return m_x*sin(m_z);
}
```

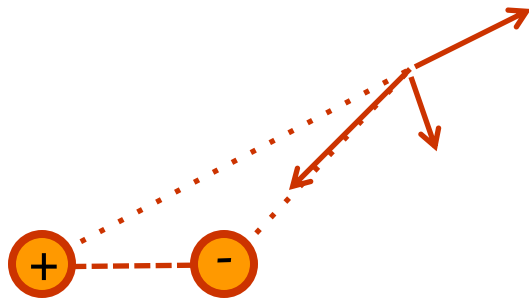
Per qualche ragione decidiamo che le tre variabili interne della classe non siano più le coordinate cartesiane ma quelle sferiche (redesign completo della classe). Possiamo adattare i metodi di accesso e i costruttori



Nessun cambiamento nel main!

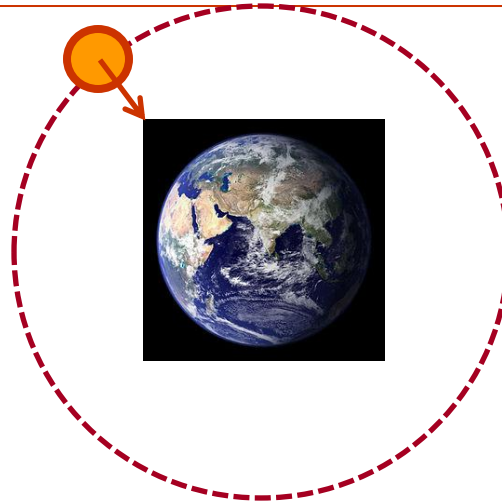
(se il main avesse avuto accesso diretto a `m_x`, `m_y` e `m_z` avrei dovuto modificare anche il main causando danni enormi agli utilizzatori della mia classe ! )

## Problemi fisici interessanti

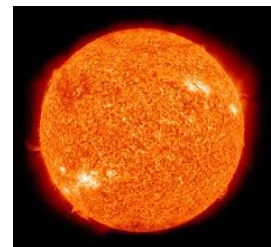


Campo generato da un  
dipolo ( e magari un  
multipolo anche)

Campo gravitazionale su  
un satellite



Potenziale gravitazionale  
terra-luna e sole-luna



Elementi comuni a questi tipi di problemi : posizioni, sorgenti di campo (particelle cariche e/o massive), campi vettoriali. Come li descriviamo in un codice numerico C++ ? Classi !

## Calcolo campo generato da un dipolo elettrico : come modellizzare il calcolo ?

- ❑ Introdurre il concetto di posizione (tre coordinate spaziali): `class Posizione`
- ❑ Introdurre il concetto di particella (massa e carica) : `class Particella`



- ❑ Introdurre il concetto di sorgente del campo (Particella + Posizione) : `class PuntoMateriale`

- ❑ Introdurre il concetto di campo vettoriale per rappresentare il campo (una terna di numeri in una posizione): `class CampoVettoriale`

---

## Problemi fisici interessanti

Elementi comuni a questi tipi di problemi : posizioni, sorgenti di campo (particelle cariche e/o massive), campi vettoriali. Come li descriviamo in un codice numerico C++ ? Classi.

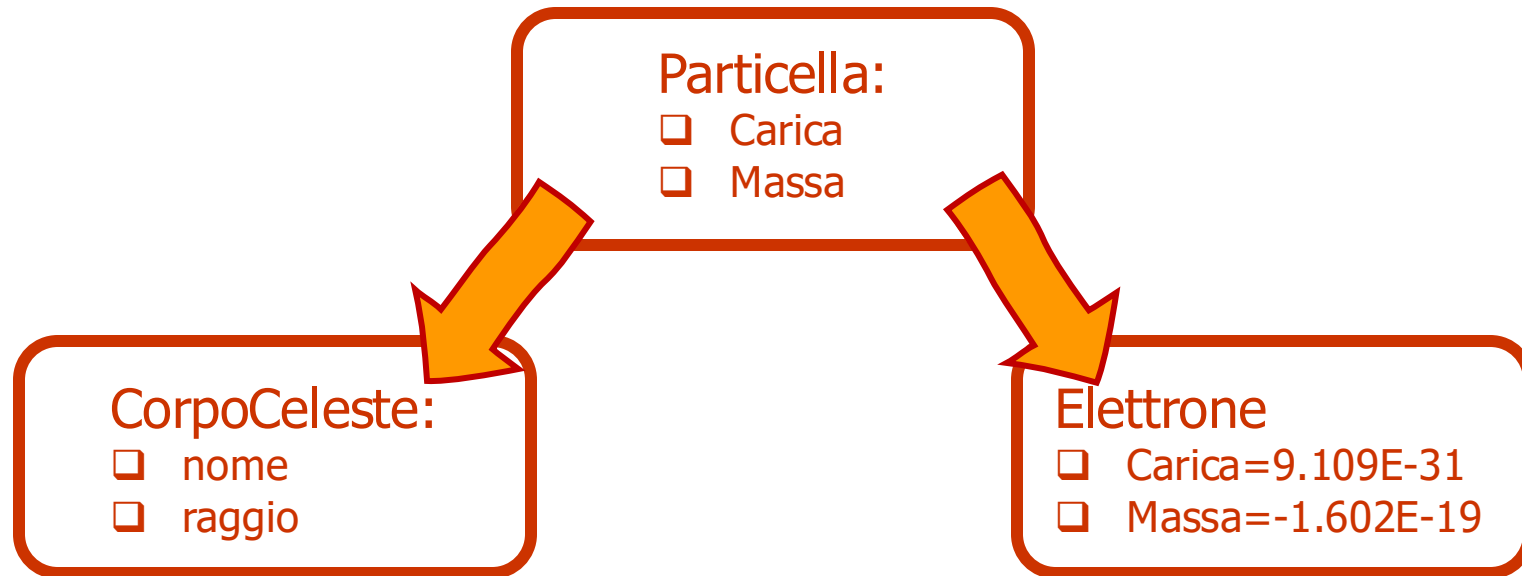
- ❑ Il Sole, la Luna e la Terra hanno proprietà in comune con la particella (massa e carica nulla ) ma hanno caratteristiche aggiuntive come per esempio una estensione (raggio)
- ❑ Elettrone : l'elettrone può essere rappresentato mediante una classe che contiene un valore per la sua massa  $m_e$  e la sua carica  $e$ . Ma anche un protone formalmente può essere rappresentato allo stesso modo, mediante la sua massa e la sua carica.
  - ❑ Sia l'elettrone sia il protone li possiamo pensare come generiche particelle con proprietà comuni e proprietà caratterizzanti

Il C++ cerca di trarre vantaggio dalle relazioni tra gli oggetti di cui abbiamo bisogno per risolvere il nostro problema attraverso una proprietà chiamata **ereditarietà**



## Ereditarietà :

Le classi in C++ possono essere estese, creando nuove classi che mantengono le caratteristiche della classe base. Questo processo, noto come ereditarietà, coinvolge una classe base e una classe derivata: la classe derivata eredita i membri della classe base, alla quale può aggiungere i propri membri.



- ☐ **CorpoCeleste** avrà tutte le caratteristiche di **Particella** + nome e raggio
- ☐ **Elettrone** sarà semplicemente una **Particella** con carica e massa definita

## Esempio di ereditarietà

### Classe madre (base class)

```
class Particella {  
public:  
    // costruttori  
    Particella(double massa, double carica);  
    // distruttore  
    ~Particella();  
    // metodi  
    double GetMassa() const {return m_massa;}  
    double GetCarica() const {return m_carica;}  
    void Print() const;  
protected:  
    double m_massa;  
    double m_carica;  
};
```

### Classe figlia (derived class)

```
class CorpoCeleste : public Particella {  
public:  
  
    // costruttori  
    CorpoCeleste(std::string nome, double massa, double raggio);  
    // distruttore  
    ~CorpoCeleste();  
    // nuovi metodi  
    void SetNome(std::string nome) {m_nome=nome;}  
    void SetMassa(double massa) {m_massa=massa;}  
    void SetRaggio(double raggio) {m_raggio=raggio;}  
    std::string GetNome() const {return m_nome;}  
    double GetRaggio() const {return m_raggio;}  
    void Print() const;  
  
protected:  
    // nuovi data membri  
    std::string m_nome;  
    double m_raggio;  
};
```

- ❑ La classe derivata `CorpoCeleste` avrà tutte le proprietà della classe madre `Particella` :

- ❑ Dato un `CorpoCeleste c` posso invocare `c.GetMassa()`
- ❑ Data una `Particella p` non posso invocare `p.SetRaggio(22)`;
- ❑ La classe figlia può sovrascrivere un metodo della classe madre (`Print`)

## Esempio di ereditarietà

### Classe madre (base class)

```
class Particella {  
public:  
    // costruttori  
    Particella(double massa, double carica);  
    // distruttore  
    ~Particella();  
    // metodi  
    double GetMassa() const {return m_massa;}  
    double GetCarica() const {return m_carica;}  
    void Print() const;  
protected:  
    double m_massa;  
    double m_carica;  
};
```

### Classe figlia (derived class)

```
class Elettrone : public Particella {  
public:  
    // costruttore  
    Elettrone();  
    // distruttore  
    ~Elettrone();  
    //  
    void Print() const;  
};
```

- ❑ La classe derivata elettrone avrà tutte le proprietà della classe madre particella :
  - ❑ Dato un Elettrone e posso invocare e.GetMassa()
- ❑ Si procede dalla classe più generale (che ha meno elementi) a quella più dettagliata (che possiede tutti gli elementi della madre più altri suoi specifici)

---

## Note sull'ereditarietà :

- ❑ Notate il nuovo tipo di qualificatore di accesso: `protected`
  - ❑ Questo qualificatore indica che i metodi o data membri sono accessibili da dentro la classe e da tutte le classi figlie
  - ❑ Un data membro o metodo `private` è accessibile solo all'interno della classe ( non dalle figlie e ovviamente non dal main )
  
- ❑ Dichiarazione della classe: `Class Elettrone : public Particella`
  - ❑ Questa istruzione dichiara che `Elettrone` è una classe derivata da `Particella`. Il qualificatore `public` indica il livello massimo con cui i metodi e data membri sono ereditati (ciò che è pubblico resta pubblico, ciò che è `protected` resta `protected`, ciò che è `private`... non viene comunque ereditato)

## Costruttori/distruttori ed ereditarietà : CorpoCeleste


### ❑ Costruttore della classe madre (Particella):

```
Particella::Particella(double massa, double carica) {  
    m_massa = massa;  
    m_carica = carica;  
}
```

### ❑ Costruttori della classe CorpoCeleste:

#### ❑ specificare i costruttori delle classi madri

```
CorpoCeleste::CorpoCeleste(string nome, double massa, double raggio) : Particella(massa,0) {  
    m_nome = nome;  
    m_raggio = raggio;  
}
```



#### ❑ oppure il compilatore assume esistenza costruttore senza argomenti della madre e lo invoca

```
CorpoCeleste::CorpoCeleste(string nome, double massa, double raggio) {  
    m_nome = nome;  
    m_raggio = raggio;  
    m_massa = massa;  
    m_carica = 0;  
}
```

ATTENZIONE: non funziona se non c'è costruttore senza argomenti nella classe madre

## Costruttori/distruttori ed ereditarietà : Elettrone

Tecnicamente costruttori e distruttori non si ereditano ma vengono chiamati dai costruttori e distruttori delle classi figlie (automaticamente o specificando quali ), vediamo alcuni esempi:

### ❑ Costruttore della classe madre (Particella):

```
Particella::Particella(double massa, double carica) {  
    m_massa = massa;  
    m_carica = carica;  
}
```

### ❑ Costruttore della classe Elettrone : il costruttore della classe elettrone semplicemente invoca il costruttore della classe madre (attraverso l'operatore ":") ma non fa nulla di più

```
Elettrone::Elettrone() : Particella(9.1093826E-31,-1.60217653E-19) {  
    // Invoco il costruttore della classe base con i parametri  
    // opportuni, ma poi non c'e' altro da fare  
}
```

## Come si usano le classi ?

```
#include "particella.h"
#include <iostream>

using namespace std;

int main() {

    Particella a (1.,1.6E-19);
    Elettrone *e    = new Elettrone();
    CorpoCeleste *c = new CorpoCeleste("Terra",5.9742E24,6.372797E6) ;

    // Metodi della classe base
    cout << "Particella " << a.GetMassa() << "," << a.GetCarica() << endl;
    a.Print();

    // Metodi della classe derivata (inherited)
    cout << "Elettrone " << e->GetMassa() << "," << e->GetCarica() << endl;
    e->Print();

    // Metodi della classe derivata (inherited)
    cout << "CorpoCeleste " << c->GetMassa() << " , " << c->GetCarica() << endl;
    c->Print();
    // Metodi aggiuntivi
    cout << c->GetNome() << endl;

    Particella b(a) ;    // costruisco una Particella a partire da una Particella
    Particella d(*e) ;    // costruisco una Particella a partire da un Elettrone
    Elettrone f(d) ;    // costruisco un Elettrone a partire da una Particella

    return 0;
}
```

Creiamo un oggetto o un puntatore

Viene sempre fornito un costruttore di copia: non è necessario fare overloading se non c'è niente di speciale nella nostra classe

Avremmo dovuto aggiungere delete e; delete c; Per rilasciare la memoria

C'è chiaramente qualcosa di illogico !

## Smart pointers ( only for curious kids )

<https://www.internalpointers.com/post/beginner-s-look-smart-pointers-modern-c>

- ❑ I puntatori nei linguaggi C e C++ sono estremamente potenti, eppure molto pericolosi: la loro gestione è completamente lasciata a te. Ogni oggetto allocato dinamicamente (ovvero `new T`) deve essere seguito da una de-allocazione manuale (ovvero l'eliminazione di `T`). Se la deallocazione non viene effettuata correttamente incorreremo certamente in un bel ***memory leak***.
- ❑ Inoltre, gli array allocati dinamicamente (ovvero `new T[N]`) richiedono di essere eliminati con un operatore diverso (ovvero `delete[]`). Questo obbliga il programmatore a tenere traccia mentalmente di quanto è stato assegnato, e a chiamare di conseguenza l'operatore giusto.
- ❑ Gli **smart pointers** sono nati per risolvere tutti questi problemi. Essenzialmente forniscono una gestione automatica della memoria: quando uno smart pointer non è più in uso, cioè quando esce di scope, la memoria a cui punta viene deallocata automaticamente. I puntatori tradizionali sono ora conosciuti anche come *raw pointers*.

```
std::unique_ptr<int>          p1 = std::make_unique<int>();  
std::unique_ptr<int[]>       p2 = std::make_unique<int[]> (50);  
std::unique_ptr<Eletttrone> e  = std::make_unique<Eletttrone>
```



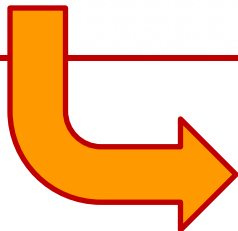
## Smart pointers ( only for curious kids )

```
class Particella {
public:
    Particella(double massa, double carica) {
        m_massa = massa;
        m_carica = carica;
        cout << "Constructor : m = " << massa << " q = " << carica << endl;
    };

    ~Particella() {
        cout << "Destructor : m = " << m_massa << " q = " << m_carica << endl;
    };

    void Print() const {
        cout << " Print : m = " << m_massa << " q = " << m_carica << endl;
    };

private:
    double m_massa;
    double m_carica;
};
```



```
// check the difference between raw pointers and smart pointers
int main() {

    for (auto i = 0; i < 10; i++) {

        cout << "=====>>>>> Starting loop = " << i << endl;

        // Particella *p = new Particella(i, 100 * i);
        // Particella p(i, i * 100);
        // unique_ptr<Particella> p = make_unique<Particella>(i, 100 * i);

        cout << "vediamo se fa casino" << endl;

        // do something
        // ...

    }

    return 0;
}
```

Try to uncomment  
one of the three

## Ereditarietà multipla :

### PuntoMateriale.h

```
#ifndef __PuntoMateriale_h__
#define __PuntoMateriale_h__

#include "Particella.h"
#include "Posizione.h"

#include "CampoVettoriale.h"

class PuntoMateriale : public Particella, public Posizione {
public:
    PuntoMateriale(double massa, double carica, const Posizione&);
    PuntoMateriale(double massa, double carica, double x, double y, double z);
    ~PuntoMateriale() { ; } ;
};

#endif // __PuntoMateriale_h__
```

Non c'è bisogno di fare  
overloading del distruttore in  
questo caso

### PuntoMateriale.cpp

```
PuntoMateriale::PuntoMateriale(double massa, double carica, const Posizione& r) :
    Particella(massa,carica), Posizione(r) {
}

PuntoMateriale::PuntoMateriale(double massa, double carica, double x, double y, double z) :
    Particella(massa,carica), Posizione(x,y,z) {
}

PuntoMateriale::~PuntoMateriale() {
}
```

## Ereditarietà multipla, i costruttori

```
PuntoMateriale::PuntoMateriale(double massa, double carica, const Posizione& r) :  
    Particella(massa,carica), Posizione(r) {  
}
```

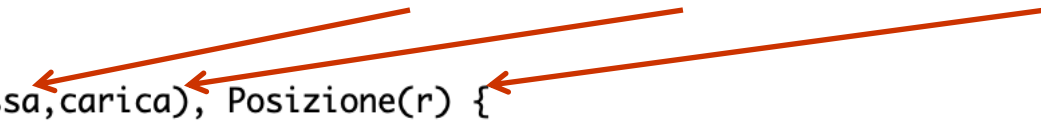


Diagram illustrating the constructor call: `Particella(massa,carica)` and `Posizione(r)` are highlighted with arrows pointing to the corresponding parameters in the `PuntoMateriale` constructor signature: `double massa`, `double carica`, and `const Posizione& r`.

- ❑ Questo costruttore di accetta massa, carica e una Posizione
- ❑ Usa questi tre parametri come input per i costruttori delle classi madri

```
int main() {  
    Posizione r(3,4,5) ;  
    PuntoMateriale P ( 1, 1, r );  
}
```

```
PuntoMateriale::PuntoMateriale(double massa, double carica, double x, double y, double z) :  
    Particella(massa,carica), Posizione(x,y,z){  
}
```

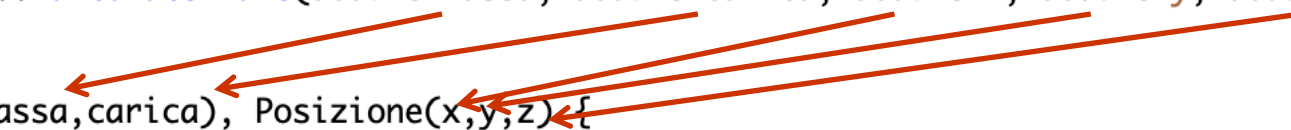


Diagram illustrating the constructor call: `Particella(massa,carica)` and `Posizione(x,y,z)` are highlighted with arrows pointing to the corresponding parameters in the `PuntoMateriale` constructor signature: `double massa`, `double carica`, `double x`, `double y`, and `double z`.

- ❑ Questo costruttore accetta massa, carica e tre coordinate
- ❑ Usa questi tre parametri come input per i costruttori delle classi madri

```
int main() {  
    PuntoMateriale Q ( 1, 1, 3, 4, 5);  
}
```

## Possibile struttura delle classi : generico campo vettoriale

```
#ifndef __CampoVettoriale_h__  
#define __CampoVettoriale_h__
```

```
#include "Posizione.h"
```

```
class CampoVettoriale : public Posizione {
```

```
public:
```

```
    CampoVettoriale(const Posizione&);  
    CampoVettoriale(const Posizione&, double Fx, double Fy, double Fz);  
    CampoVettoriale(double x, double y, double z, double Fx, double Fy, double Fz);  
    ~CampoVettoriale();
```

costruttori

```
    CampoVettoriale & operator+=( const CampoVettoriale & ) ;  
    CampoVettoriale operator+( const CampoVettoriale & ) const;
```

Overloading operatori

```
    double getFx() const {return m_Fx;}  
    double getFy() const {return m_Fy;}  
    double getFz() const {return m_Fz;}
```

Aggiungere anche i metodi set

```
    double Modulo();
```

Calcola modulo del campo

```
private:
```

```
    double m_Fx, m_Fy, m_Fz;
```

```
};
```

Tre componenti del campo

```
#endif // __CampoVettoriale_h__
```

```
CampoVettoriale(const Posizione&);
CampoVettoriale(const Posizione&, double Fx, double Fy, double Fz);
CampoVettoriale(double x, double y, double z, double Fx, double Fy, double Fz);
```

// costruttori

```
CampoVettoriale::CampoVettoriale(const Posizione& r) : Posizione(r.getX(), r.getY(), r.getZ()) {
    m_Fx=0.;
    m_Fy=0.;
    m_Fz=0.;
}
```

```
CampoVettoriale::CampoVettoriale(const Posizione& r, double Fx, double Fy, double Fz) : Posizione(r) {
    m_Fx=Fx;
    m_Fy=Fy;
    m_Fz=Fz;
}
```

```
CampoVettoriale::CampoVettoriale(double x, double y, double z, double Fx, double Fy, double Fz) : Posizione(x,y,z) {
    m_Fx=Fx;
    m_Fy=Fy;
    m_Fz=Fz;
}
```

## Nota II : overloading operator+ e operator+=

La somma restituisce un nuovo oggetto (by value)

```
// overloading operatori

CampoVettoriale CampoVettoriale::operator+( const CampoVettoriale & v ) const {

    if ( ( v.getX()!= getX() ) || ( v.getY()!= getY() ) || ( v.getZ()!= getZ() ) ) {
        std::cout << "Somma di campi vettoriali in punti diversi non ammessa" << std::endl;
        exit (-11) ;
    }

    CampoVettoriale sum ( Posizione( getX(),getY(), getZ() ) ) ;
    sum.setFx( getFx() + v.getFx() ) ;
    sum.setFy( getFy() + v.getFy() ) ;
    sum.setFz( getFz() + v.getFz() ) ;

    return sum;
}

CampoVettoriale & CampoVettoriale::operator+=( const CampoVettoriale & v ) {
    return (*this) = (*this)+v;
}
```

Operator+= lavora sull'oggetto,  
restituisce una reference a se stesso

Prima di sommare si potrebbe  
aggiungere un controllo che  
la posizione dei due vettori  
sia la stessa

Si puo' anche scrivere esplicitamente l'operator+= e  
implementare operator+ usando operator+=

## Possibile struttura delle classi : completiamo la sorgente del campo

```
#ifndef __PuntoMateriale_h__
#define __PuntoMateriale_h__

#include "particella.h"
#include "Posizione.h"

#include "CampoVettoriale.h"

class PuntoMateriale : public Particella, Posizione {
public:
    PuntoMateriale(double massa, double carica, const Posizione&);
    PuntoMateriale(double massa, double carica, double x, double y, double z);
    ~PuntoMateriale();

    CampoVettoriale CampoElettrico(const Posizione&) const ;
    CampoVettoriale CampoGravitazionale(const Posizione&) const;
};

#endif // __PuntoMateriale_h__
```

Campo calcolato in una  
determinata posizione

## Implementare il campo elettrico e gravitazionale

```
CampoVettoriale PuntoMateriale::CampoElettrico(const Posizione& b) const {  
  
    double Ex = ... ;  
    double Ey = ... ;  
    double Ez = ... ;  
  
    return CampoVettoriale(b.getX(),b.getY(),b.getZ(),Ex,Ey,Ez);  
}  
  
CampoVettoriale PuntoMateriale::CampoGravitazionale(const Posizione& b) const {  
  
    double Ex = ... ;  
    double Ey = ... ;  
    double Ez = ... ;  
  
    return CampoVettoriale(b.getX(),b.getY(),b.getZ(),Ex,Ey,Ez);  
}
```



## Calcolo campo generato da un dipolo elettrico : come modellare il calcolo ?

❑ Introdurre il concetto di posizione (tre coordinate spaziali): `class Posizione`

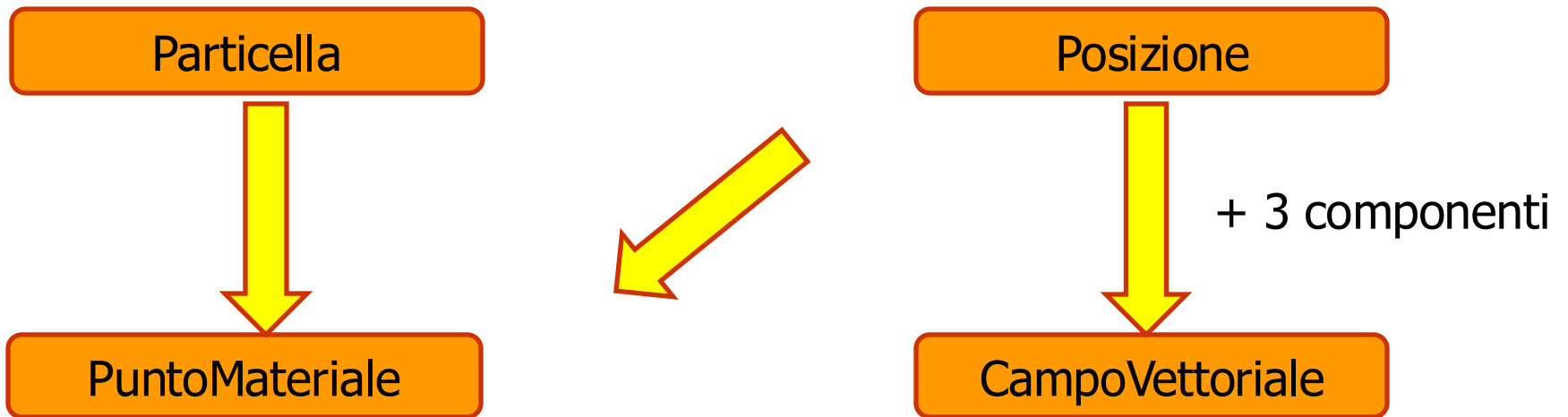
❑ Introdurre il concetto di particella (massa e carica) : `class Particella`

❑ Introdurre il concetto di campo vettoriale per rappresentare il campo (una terna di numeri in una posizione): `class CampoVettoriale`



❑ Introdurre il concetto di sorgente del campo (Particella + Posizione) : `class PuntoMateriale`

## Possibile struttura delle classi



CampoVettoriale CampoElettrico(const Posizione& r)  
CampoVettoriale CampoGravitazionale(const Posizione& r)

CampoVettoriale & operator+=( const CampoVettoriale & v )  
CampoVettoriale operator+(const CampoVettoriale & v )

## Il programma finale

```
#include "PuntoMateriale.h"
#include "CampoVettoriale.h"

#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char** argv) {
    if (argc!=4) {
        cerr << "Usage: " << argv[0] << " <x> <y> <z>" << endl;
        exit(-1);
    }

    double x=atof(argv[1]);
    double y=atof(argv[2]);
    double z=atof(argv[3]);
    Posizione r(x,y,z);

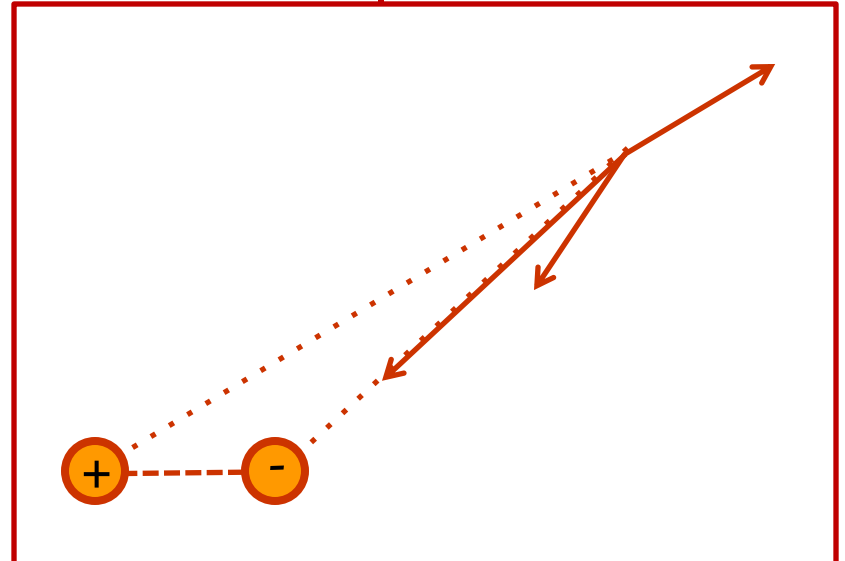
    const double e =1.60217653E-19 ;
    const double me=9.1093826E-31;
    const double mp=1.67262171E-27;
    const double d =1.E-10;

    PuntoMateriale elettrone(me,-e,0.,0., d/2.);
    PuntoMateriale protone (mp, e,0.,0.,-d/2.);

    CampoVettoriale E = elettrone.CampoElettrico(r) + protone.CampoElettrico(r) ;

    cout << "E=(" << E.getFx() << "," << E.getFy() << "," << E.getFz() << ")" << endl;

    return 0;
}
```



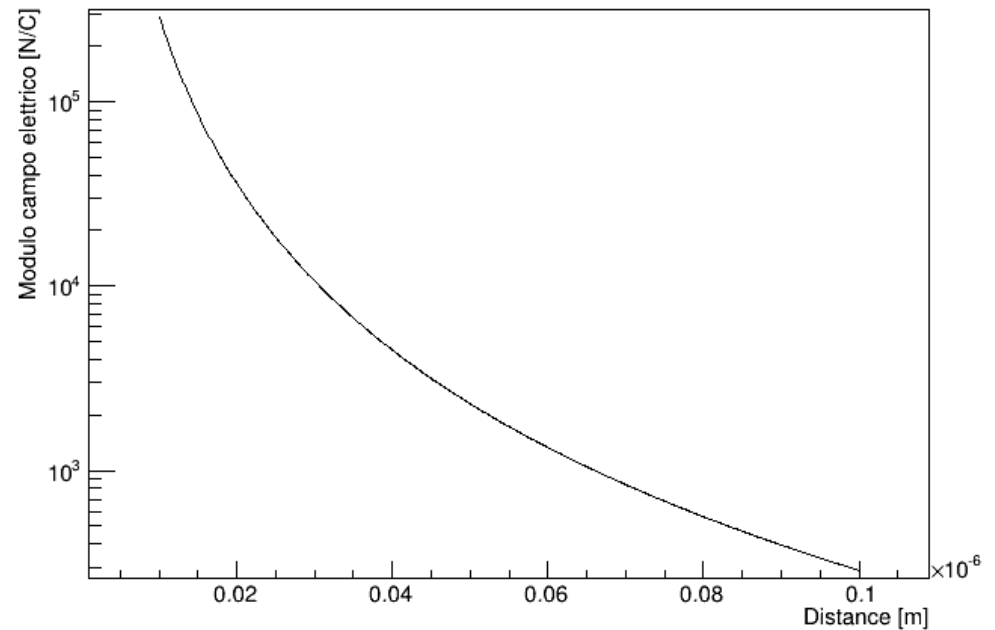
## Il programma finale

```
TGraph field;  
Posizione P(0,0,0);  
double E1 = 0;  
double E2 = 0;  
  
for ( int k = 100 ; k <= 1000 ; k++ ) {  
    P.setZ( k*d ) ;  
    CampoVettoriale Ed = elettrone.CampoElettrico(P) + protone.CampoElettrico(P) ;  
  
    if ( k == 100 )    E1 = Ed.Modulo() ;  
    if ( k == 1000 )   E2 = Ed.Modulo() ;  
  
    field.SetPoint(k-100, k*d, Ed.Modulo() );  
}  
  
field.Draw("ALP");
```

Loop sulla distanza

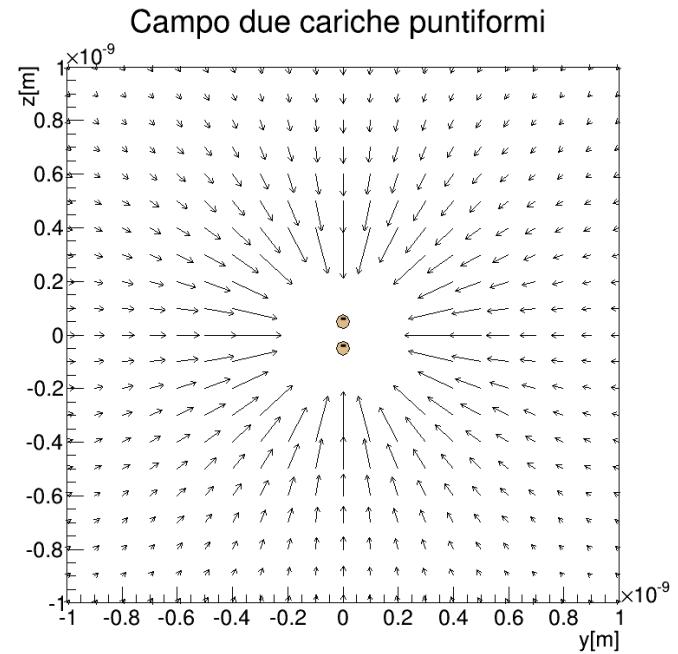
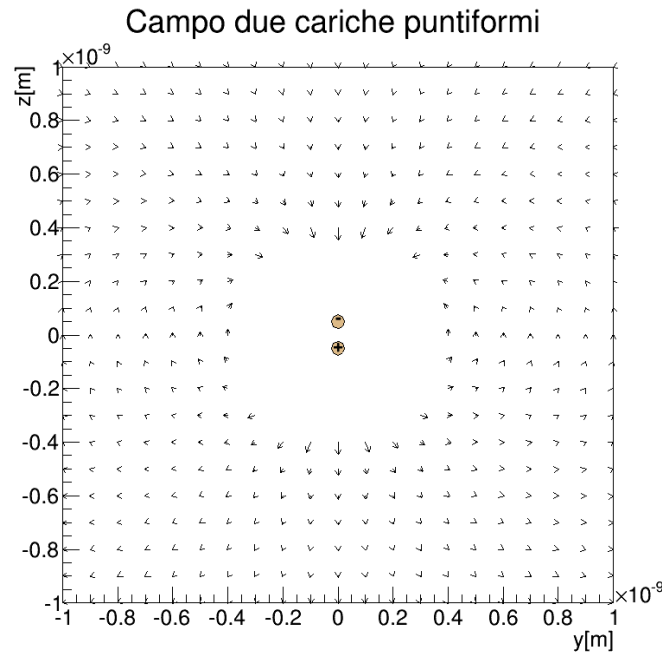
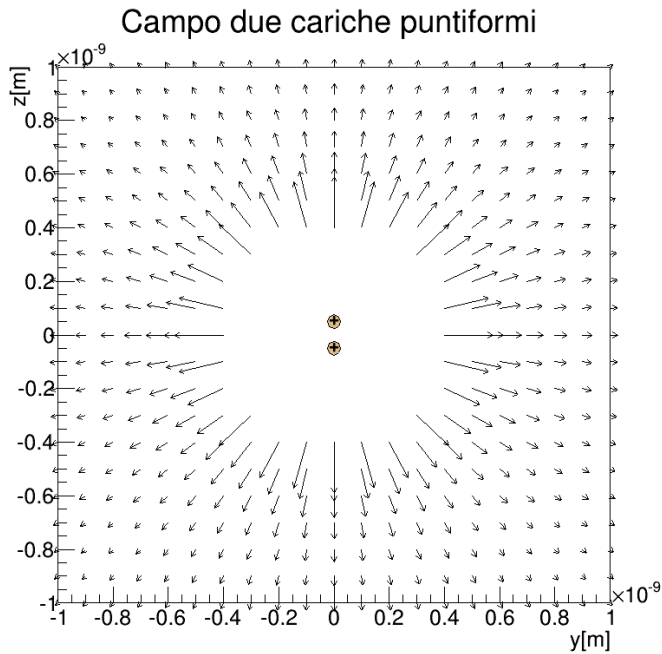
Muovo la posizione e  
calcolo il campo totale

Campo elettrico dipolo

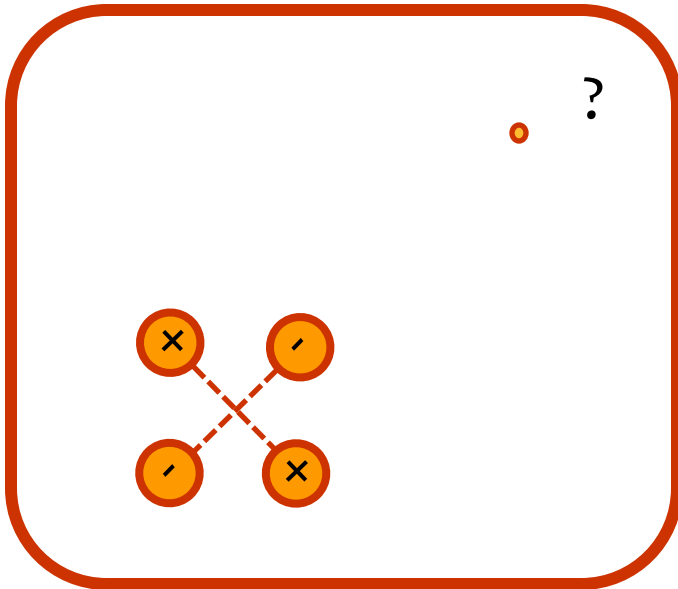


## Alcune varianti sulla rappresentazione del dipolo

Riuscite a dare una rappresentazione 2D del campo ? Si possono usare delle TArrow per indicare direzione e intensità del campo in ogni punto del piano



## E se voglio studiare un multipolo ? Facile !



```
#include "CampoVettoriale.h"
#include "PuntoMateriale.h"

#include <cstdlib>
#include <cmath>
#include <iostream>

using namespace std;

int main(int argc, char** argv) {

    if ( argc!= 2) {
        cerr << "Usage: " << argv[0] << " <number_of_poles>" << endl;
        exit(-1);
    }

    const int n=atoi(argv[1]);
    const double e =1.60217653E-19 ;
    const double me=9.1093826E-31;
    const double r =0.5E-10;

    const double R1 = 1.E-8 ;

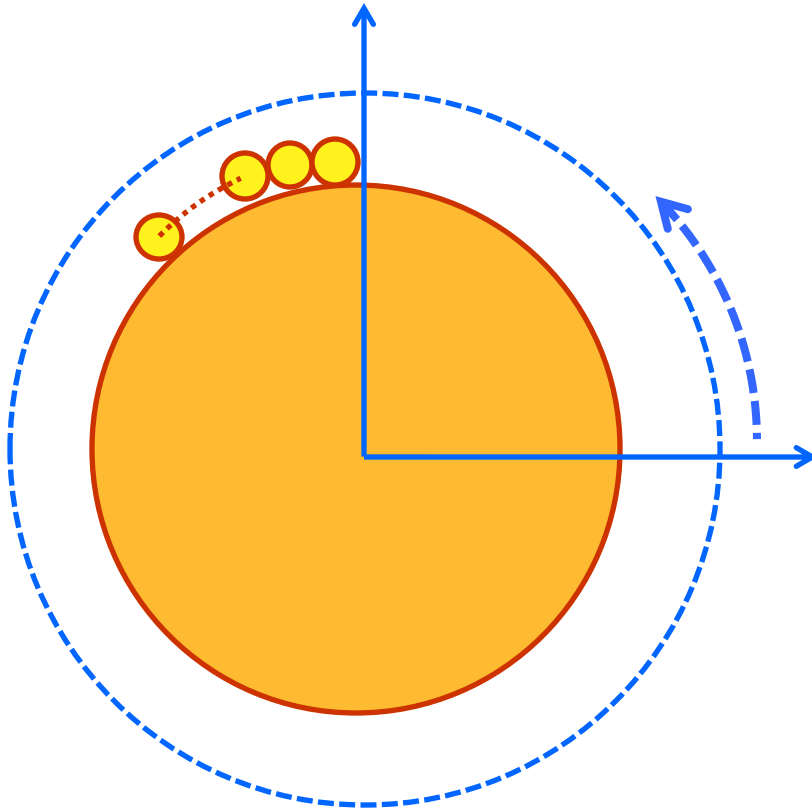
    Posizione p(R1,0.,0.);
    CampoVettoriale E(p);

    for (int i=0; i<n; i++) {
        double phi = i*(2.*M_PI/n);
        PuntoMateriale particella(me,pow(-1.,i)*e,r*cos(phi),r*sin(phi),0.);
        E+=particella.CampoElettrico( p );
    }

    cout << "E=(" << E.getFx() << "," << E.getFy() << "," << E.getFz() << ")" << endl;

    return 0;
}
```

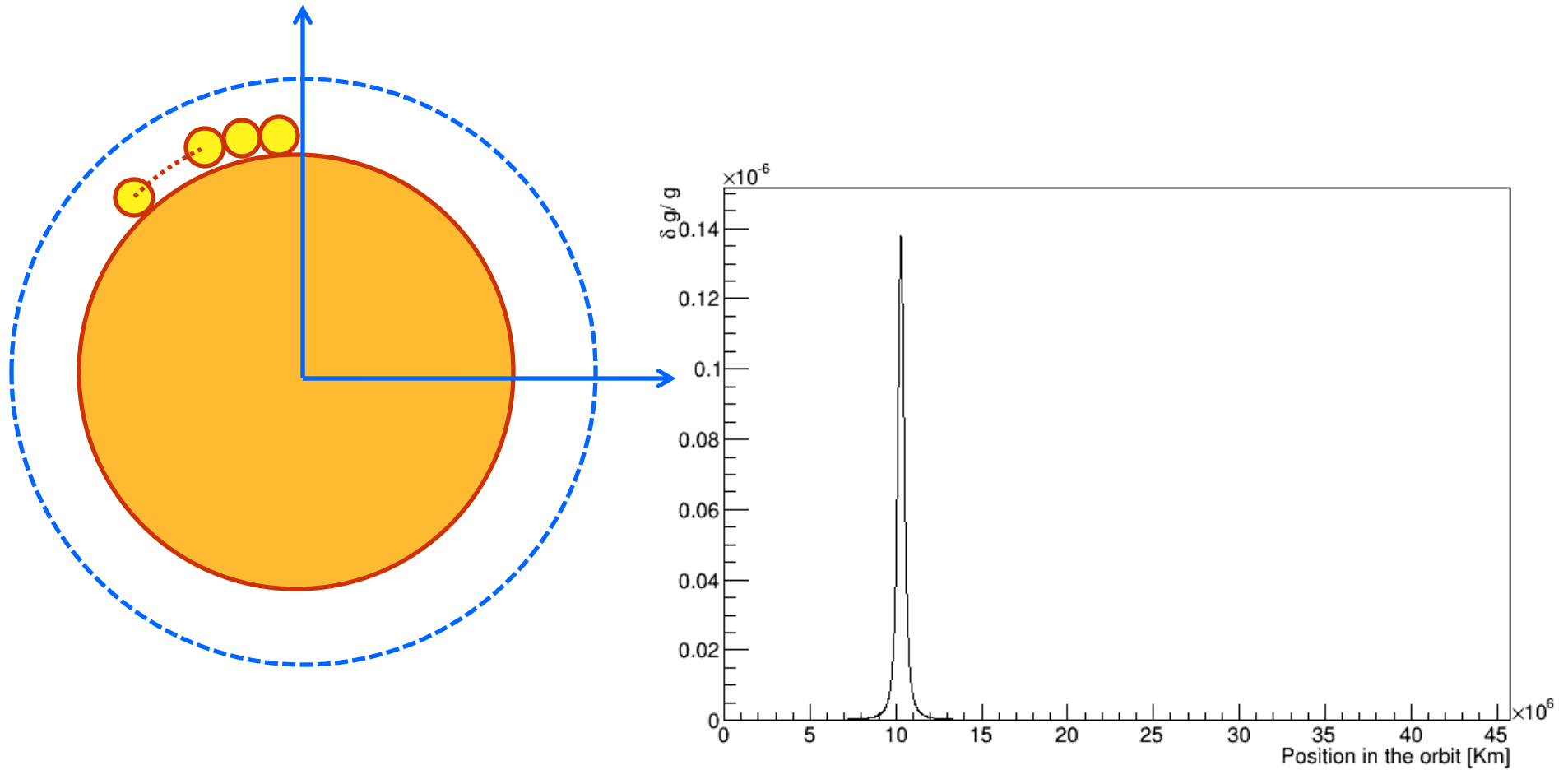
## Effetto di distribuzioni di massa non omogenea



Il satellite GOCE si trova in orbita a 250 km dalla superficie terrestre e dotato di accelerometri in grado di misurare variazioni dell'accelerazione di gravità fino a  $\delta g/g = 10^{-13}$ . Scopo della missione include una mappatura dettagliata del campo gravitazionale terrestre, prodotto dalle distribuzioni non omogenee di massa.

- ☐ Si calcoli l'accelerazione di gravità sul satellite per una Terra perfettamente sferica e dimensioni pari al raggio medio.
- ☐ Calcolare poi la variazione relativa di  $g$  prodotta da una catena montuosa, schematizzata come una fila di 100 sfere di 1 km di raggio poste sopra la superficie media della Terra (usare  $3000 \text{ kg/m}^3$  come densità della roccia).

## Effetto di distribuzioni di massa non omogenea





---

## Le string in C++

- ❑ Per rappresentare “parole” in C++ possiamo usare un array di char dal C :

```
❑ char myWord[10] ;  
   myWord[0] = 'h' ;  
   myWord[1] = 'e' ;  
   myWord[2] = 'l' ;  
   myWord[3] = 'l' ;  
   myWord[4] = 'o' ;
```

- ❑ Il C++ non possiede un vero e proprio tipo specifico per trattare le sequenze di caratteri :
- ❑ Le librerie standard del C++ forniscono la classe `string`, (`#include <string>`) che evita tutte le problematiche di allocazione dei buffer
  - ❑ la stringa e' considerata internamente come un semplice array di caratteri terminata da un carattere NULL (`\0`).
  - ❑ Serie di metodi utili per semplificare le operazioni

---

## Le string

Qualche caratteristica ( il resto in <http://www.cplusplus.com/reference/string/string/> ):

- ❑ `getline(cin, mystring)`: this function is used to store a stream of characters as entered by the user in the object memory.
- ❑ `mystring.push_back('s')`: used to input a character at the end of the string.
- ❑ `mystring.pop_back()`: used to delete the last character from the string.
- ❑ `mystring.capacity()`: returns the capacity allocated to the string, which can be equal to or more than the size of the string. Additional space is allocated so that when the new characters are added to the string, the operations can be done efficiently.
- ❑ `mystring.length()`: returns the length of the string
- ❑ `mystring.begin()`: returns an iterator to beginning of the string.
- ❑ `mystring.end()`: returns an iterator to end of the string.

# Le string

```
#include <string>
#include <iostream>
#include <vector>

using namespace std;

int main() {

    string nome = "Leonardo";
    cout << "Il mio nome = " << nome << endl;

    char c_cognome[30] = "Carminati";
    string cognome = c_cognome;

    cout << "Cognome = " << cognome << endl;
    cout << "Lunghezza cognome = " << cognome.size() << endl;

    string record = nome + " " + cognome;
    cout << record << endl;

    record.insert(nome.size()+1, "Carlo ");
    cout << record << endl;

    cout << record.find( "Carlo", 0) << endl;

    vector<string> cognomi;

    cout << "Inserisci cognome " << endl;
    cin >> cognome ;
    cognomi.push_back(cognome);

}
```

Create a string from an array of characters

Concatenate strings

Insert a substring in a given position

Fill a string from standard input

---

## Le string e char\*

Spesso è necessario convertire `string` in `char*` e viceversa ( per esempio nomi e titoli degli oggetti di ROOT sono `char *` ).

- ❑ Si può convertire una `string` in un `char*` :

```
string nome_s = "Leonardo";  
const char * nome_c = nome_s.c_str();
```

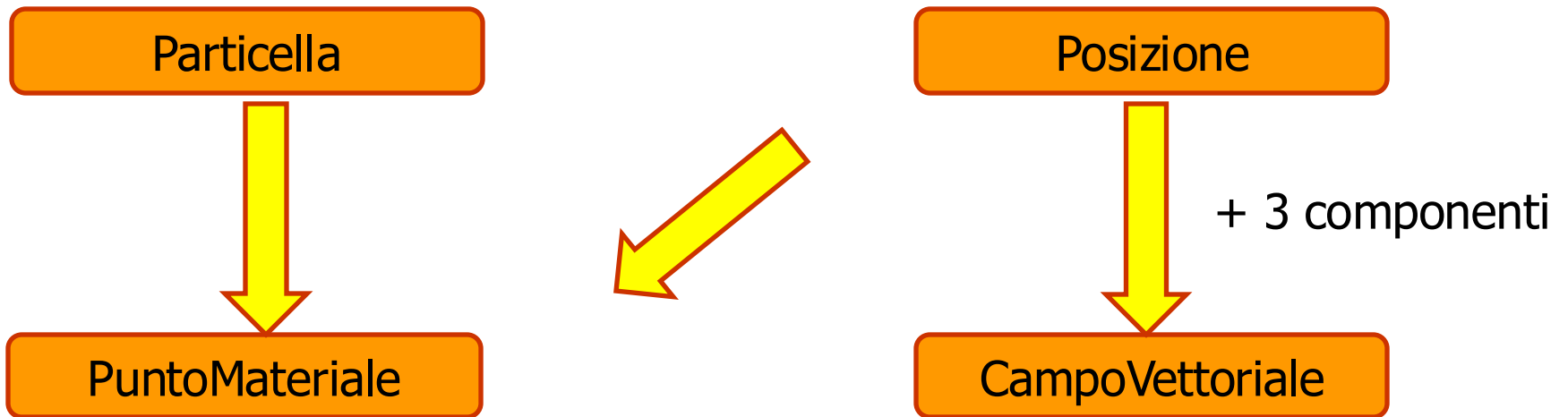
- ❑ Si può convertire un `char*` in una `string`

```
const char * nome_c = "Leonardo";  
string nome_s ( nome_c ) ;
```

---

**Backup**

## Possibile struttura delle classi



CampoVettoriale CampoElettrico(const Posizione& r)  
CampoVettoriale CampoGravitazionale(const Posizione& r)

CampoVettoriale & operator+=( const CampoVettoriale & v )  
CampoVettoriale operator+(const CampoVettoriale & v )

## Calcolo potenziale generato dalla terra e da sole sulla luna

```
#include "Posizione.h"

class CorpoCeleste : public Particella {
public:

    CorpoCeleste(std::string nome, double massa, double raggio);

    ~CorpoCeleste();

    void SetNome(std::string nome) {m_nome=nome;}
    void SetMassa(double massa) {m_massa=massa;}
    void SetRaggio(double raggio) {m_raggio=raggio;}
    std::string GetNome() const {return m_nome;}
    double GetRaggio() const {return m_raggio;}
    virtual void Print() const;
    void SetPosizione(const Posizione& r) {m_r=r;}
    Posizione GetPosizione() const {return m_r;}

    double PotGrav(const Posizione& r) const;

protected:

    std::string m_nome;
    double m_raggio;
    Posizione m_r;

public:
    static const double Ggrav=6.67428E-11;
};
```

Calcolo il potenziale generato dal  
corpo celeste in una posizione

Aggiungo una Posizione  
come data-member

## Calcolo potenziale generato dalla terra e da sole sulla luna

```
#include "particella.h"
#include "Posizione.h"

#include <iostream>
using namespace std;

int main() {

    CorpoCeleste Sole("Sole",1.9891E30,1.39095E9);
    CorpoCeleste Terra("Terra",5.9742E24,6.372797E6);
    Terra.SetPosizione(Posizione(0.,0.,1.52097701E11));
    CorpoCeleste Luna("Luna",7.347673E22,1.738E6);
    Luna.SetPosizione(Posizione(0.,4.05696E8,1.52097701E11));

    cout << "Potenziale Sole su Luna = " << Sole.PotGrav(Luna.GetPosizione()) << endl;
    cout << "Potenziale Terra su Luna = " << Terra.PotGrav(Luna.GetPosizione()) << endl;

    return 0;
}
```



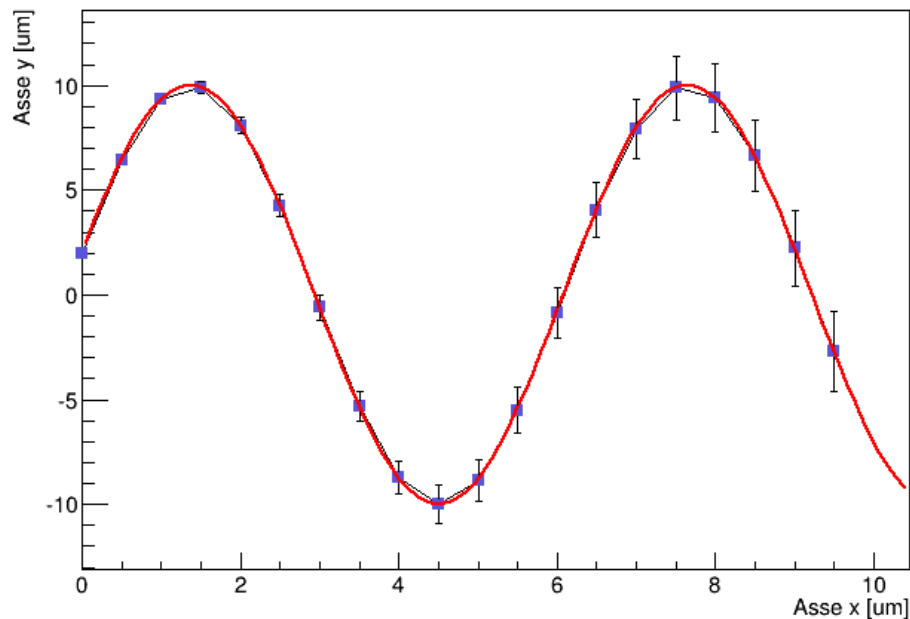
## Plot x vs y ( modulo del campo vs r ) : TGraph

Ci sono due classi di ROOT interessanti per questo tipo di rappresentazione :

- ❑ TGraph ( x vs y senza errori )
- ❑ TGraph2D ( f(x,y) )
- ❑ TGraphErrors ( x vs y con errori )

<https://replit.com/join/tksetxvgju-lcarmina>

Grafico



Grafico

