

## Possibili implementazioni alternative di CampoVettoriale(r) :

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

Ok, utilizzo il copy constructor  
(implicito) della classe Posizione

```
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) {  
    m_x = g.GetX();  
    m_y = g.GetY();  
    m_z = g.GetZ();  
    m_Fx=0.;  
    m_Fy=0.;  
    m_Fz=0.;  
}
```

Questo funziona ma non è  
consigliato ( meglio il primo !)

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

**Questo invece è proprio sbagliato !**

## Ereditarietà multipla : metodo presente in entrambi i genitori ?

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

// =====

class first {
public :
    void print () { cout << "First" << endl; } ;
};

// =====

class second {
public :
    void print () { cout << "Second" << endl; } ;
};

// =====

class third : public first, public second {
public :
    void print () { first::print(); } ;
};

// =====

int main() {
    third p3 ;
    p3.print() ;
}

// =====
```

La classe `first` ha un metodo `print()`

La classe `second` ha un metodo `print()`

La classe `third` eredita da `first` e `second`

Devo specificare nella classe figlia `third` cosa deve fare il metodo `print()` ( altrimenti mi darà un errore di compilazione )

## Un commento su operator+

La somma può anche essere implementata come funzione esterna indipendente invece che come metodo della classe ( CampoVettoriale in questo caso )

```
CampoVettoriale operator+( const CampoVettoriale & p1 , const CampoVettoriale & p2 ) {  
    if ( ( p1.getX() != p2.getX() ) || ( p1.getY() != p2.getY() ) || ( p1.getZ() != p2.getZ() ) ) {  
        std::cout << "Somma di campi vettoriali in punti diversi non ammessa" << std::endl;  
        exit (-11) ;  
    }  
  
    CampoVettoriale sum ( Posizione( p1.getX(), p1.getY(), p1.getZ() ) ) ;  
    sum.setFx( p1.getFx() + p2.getFx() ) ;  
    sum.setFy( p1.getFy() + p2.getFy() ) ;  
    sum.setFz( p1.getFz() + p2.getFz() ) ;  
  
    return sum;  
}
```

## Come costruire un grafico

```
#include "TApplication.h"
#include "TGraph.h"
#include "TCanvas.h"
#include "TAxis.h"

// ...

using namespace std;

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

    TApplication myApp("app",0,0);
    TGraph field;
    Posizione P(0,0,0);

    // ...

    for ( int k = 100 ; k <= 1000 ; k++ ) {

        P.setZ( k*d ) ;
        CampoVettoriale Ed = elettrone.CampoElettrico(P) + protone.CampoElettrico(P) ;
        field.SetPoint(k-100, k*d, Ed.Modulo() );

    }

    TCanvas can("can","Campo elettrico dipolo");
    can.SetLogy();
    field.SetTitle("Campo elettrico dipolo");
    field.Draw("ALP");
    field.GetAxis()->SetTitle("Distance [m]");
    field.GetAxis()->SetTitle("Modulo campo elettrico [N/C]");

    myApp.Run();

    return 0;

}
```

### ❑ Blocchi logici principali :

- ❑ creare un TCanvas ( TCanvas can (...) ; )
- ❑ entrare in un TCanvas ( can.cd() )
- ❑ Disegnare ( graph.Draw("ALP") )
- ❑ Aggiungere gli #include di tutti gli oggetti che si utilizzano

Crea un TGraph vuoto

Aggiungo punti al TGraph

Crea un TCanvas dedicato ( non è necessario se si ha solo un TGraph )

Modifica l'aspetto grafico del TGraph

## More on Makefile

Mettere sempre in testa il target principale

dipendenze

```
INCS=`root-config --cflags`  
LIBS=`root-config --libs`  
  
esercizio5.3: esercizio5.3.o Posizione.o PuntoMateriale.o CampoVettoriale.o Particella.o  
    g++ -o esercizio5.3 esercizio5.3.o Posizione.o PuntoMateriale.o CampoVettoriale.o Particella.o ${LIBS}  
  
esercizio5.3.o: esercizio5.3.cxx  
    g++ -c esercizio5.3.cxx -o esercizio5.3.o ${INCS}  
  
Particella.o: Particella.cxx Particella.h  
    g++ -c Particella.cxx -o Particella.o  
  
Posizione.o: Posizione.cxx Posizione.h  
    g++ -c Posizione.cxx -o Posizione.o  
  
PuntoMateriale.o: PuntoMateriale.cxx PuntoMateriale.h Posizione.h Particella.h  
    g++ -c PuntoMateriale.cxx -o PuntoMateriale.o  
  
CampoVettoriale.o: CampoVettoriale.cxx CampoVettoriale.h Posizione.h  
    g++ -c CampoVettoriale.cxx -o CampoVettoriale.o  
  
clean:  
    rm *.o
```

Librerie da linkare

Non confondere le dipendenze  
con gli `#include` (ruolo  
molto diverso)

-c option : compilare  
ma non linkare

## More on Makefile

```
INCS=`root-config --cflags`
LIBS=`root-config --libs`

esercizio5.3: esercizio5.3.o Posizione.o PuntoMateriale.o CampoVettoriale.o Particella.o
    g++ -o esercizio5.3 esercizio5.3.o Posizione.o PuntoMateriale.o CampoVettoriale.o Particella.o ${LIBS}

esercizio5.3.o: esercizio5.3.cxx
    g++ -c esercizio5.3.cxx -o esercizio5.3.o ${INCS}

Particella.o: Particella.cxx Particella.h
    g++ -c Particella.cxx -o Particella.o

Posizione.o: Posizione.cxx Posizione.h
    g++ -c Posizione.cxx -o Posizione.o

PuntoMateriale.o: PuntoMateriale.cxx PuntoMateriale.h Posizione.h Particella.h
    g++ -c PuntoMateriale.cxx -o PuntoMateriale.o

CampoVettoriale.o: CampoVettoriale.cxx CampoVettoriale.h Posizione.h
    g++ -c CampoVettoriale.cxx -o CampoVettoriale.o

clean:
    rm *.o
```

`${LIBS}` è necessario nella fase di linking (link alle librerie pre-compilate)

`${INCS}` è necessario quando si compila il codice utente che usa librerie di ROOT (specifica dove sono i files .h)

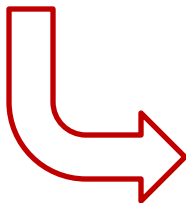
## Errore frequente :

```
g++ -o esercizio5.3 esercizio5.3.cpp Posizione.o CampoVettoriale.o Particella.o `root-config --cflags` `root-config --libs`
Undefined symbols for architecture x86_64:
  "PuntoMateriale::PuntoMateriale(double, double, double, double, double)", referenced from:
    _main in esercizio5-2fd4bf.o
  "PuntoMateriale::~PuntoMateriale()", referenced from:
    _main in esercizio5-2fd4bf.o
  "PuntoMateriale::CampoElettrico(Posizione const&) const", referenced from:
    _main in esercizio5-2fd4bf.o
ld: symbol(s) not found for architecture x86_64
clang: error: linker command failed with exit code 1 (use -v to see invocation)
make: *** [esercizio5.3] Error 1
```

PuntoMateriale.o missing!

### Possibili cause

- ❑ Metodo dichiarato ma non implementato ( successo molte volte con il distruttore )
- ❑ Makefile non corretto ( manca un file .o quando si fa il linking )



```
INCS=`root-config --cflags`
LIBS=`root-config --libs`

esercizio5.3: esercizio5.3.o Posizione.o CampoVettoriale.o Particella.o
g++ -o esercizio5.3 esercizio5.3.o Posizione.o CampoVettoriale.o Particella.o ${LIBS}

esercizio5.3.o: esercizio5.3.cxx
g++ -c esercizio5.3.cxx -o esercizio5.3.o ${INCS}

Particella.o: Particella.cxx Particella.h
g++ -c Particella.cxx -o Particella.o

Posizione.o: Posizione.cxx Posizione.h
g++ -c Posizione.cxx -o Posizione.o

PuntoMateriale.o: PuntoMateriale.cxx PuntoMateriale.h Posizione.h Particella.h
g++ -c PuntoMateriale.cxx -o PuntoMateriale.o

CampoVettoriale.o: CampoVettoriale.cxx CampoVettoriale.h Posizione.h
g++ -c CampoVettoriale.cxx -o CampoVettoriale.o

clean:
rm *.o
```

## Makefiles troppo lunghi ? Provate questo

Definisce una regola generale per generare i files oggetto.

```
myLIBS=Posizione.o PuntoMateriale.o CampoVettoriale.o Particella.o
RFLAGS:='root-config --cflags' `root-config --libs`

esercizio5.3: esercizio5.3.cpp ${myLIBS}
    g++ -Wall -o $@ $^ ${RFLAGS}

%.o : %.cpp %.h
    g++ -Wall -c $< ${RFLAGS}
```

Indica la prima dipendenza  
( i.e. il file cpp )

$\$^$  indica la lista delle  
dipendenze

$\$@$  indica il target  
name



---

# Ricerca di zeri di una funzione (e polimorfismo in C++)

Laboratorio Trattamento Numerico dei Dati Sperimentali

Prof. L. Carminati  
Università degli studi di Milano

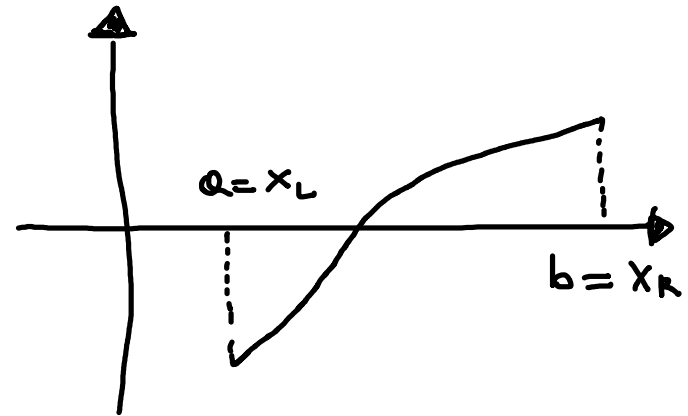
---

## Algoritmi numerici

- ❑ Capita spesso in fisica di affrontare problemi complessi per i quali non si sa impostare una soluzione analitica o non si è in grado di risolvere analiticamente le equazioni corrispondenti.
- ❑ In questo caso le tecniche di calcolo numerico consentono di ottenere soluzioni con il grado di precisione richiesto.
- ❑ Nelle prossime lezioni vedremo esempi di tecniche numeriche che si possono applicare ad alcuni problemi scelti come rappresentativi
  - ❑ Durante le prossime lezioni proveremo ad impostare da zero alcuni algoritmi per capire meglio dall'interno come funzionano
  - ❑ Prima di affrontare un problema (serio) di calcolo numerico consultare la letteratura (eg. "Numerical recipes in C") o verificare l'esistenza di librerie dedicate ( eg [BOOST](#) nel caso del C++)

## Ricerca zeri di una funzione

- Utilizziamo come usecase la ricerca degli zeri di una funzione continua in un intervallo  $[a,b]$



### Definizione : zeri di una funzione

I punti  $x_i$  tali che  $f(x_i) = 0$  sono detti zeri della funzione e radici di  $f(x) = 0$

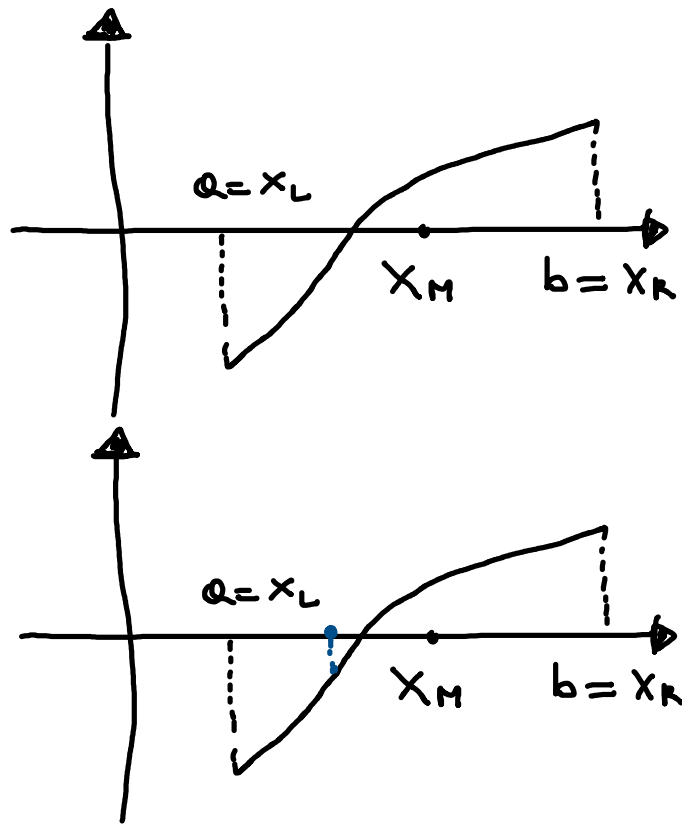
- Ricerca degli zeri di una funzione: affrontiamo il problema numericamente cercando di costruire algoritmi iterativi che forniscano approssimazioni sempre più precise dello zero cercato. Per fare questo ci viene in aiuto il teorema di Bolzano :

### Teorema di Bolzano

Se  $f$  è continua in  $[a, b]$  e  $f(a)f(b) < 0 \rightarrow \exists$  almeno uno zero in  $(a, b)$

## Ricerca zeri di una funzione : metodo della bisezione

1. Partiamo da un intervallo di ricerca per cui valgono le ipotesi del teorema di Bolzano
2. Calcoliamo il punto medio dell'intervallo e determiniamo in quale dei due sotto-intervalli  $[a, x_M]$  o  $(x_M, b]$  si trova lo zero.
3. Procediamo in modo iterativo a determinare il punto medio del nuovo intervallo



$$x_M = \frac{x_R + x_L}{2}$$

$$\text{if } f(x_M) * f(x_L) < 0 \quad (x^* \in [x_L, x_M])$$

$$x_R = x_M$$

$$\text{else} \quad (x^* \in [x_M, x_R])$$

$$x_L = x_M$$

$$x_M = \frac{x_R + x_L}{2}$$

$$\Delta = \frac{x_R - x_L}{2}$$

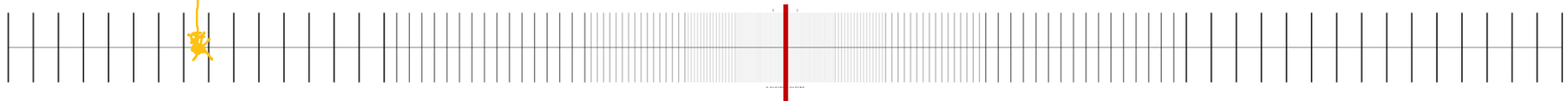
- All'ultima iterazione l'algoritmo restituisce come stima dello zero il punto di mezzo ( $x_M$ ) e come errore il valore  $\Delta/2$

## Quando interrompo l'algoritmo ??

Un metodo di ricerca iterativo deve prevedere una condizione di arresto : quando termino la ricerca? In generale almeno due condizioni:

- ❑ Inserire dall'esterno una tolleranza  $\varepsilon$ : quando la dimensione dell'intervallo di ricerca  $\Delta x$  diventa inferiore alla tolleranza l'algoritmo si ferma
  - ❑ Attenzione al valore minimo dell'intervallo di ricerca : non ha senso che sia più piccolo dell'errore di rappresentazione del numero. La precisione  $\Delta x$  con cui è noto un numero  $x$  rappresentato come double dipende dal suo valore e può essere maggiorato dalla quantità

$$\Delta x \leq \varepsilon_M x \quad (\varepsilon_M = \text{DBL\_EPSILON} = 2 \cdot 10^{-16} \text{ accessibile in } \text{<cmath>})$$



- ❑ Controllo sul numero massimo di iterazioni : evitare che un algoritmo entri in una condizione di loop infinito. Dopo un numero di iterazioni massimo  $N_{\max}$  l'algoritmo si ferma e restituisce un messaggio d'errore.
  - ❑ Il valore di  $N_{\max}$  viene solitamente impostato dallo sviluppatore lasciando la possibilità all'utente di variarlo ( a suo rischio ) con un tipico `setNmax(unsigned int);`

## Ricerca zeri di una funzione : metodo della bisezione

Cerchiamo ora di capire quanto è buono il metodo di ricerca degli zeri :

1. Per prima cosa assicuriamoci che la successione di approssimazioni successive dello zero  $\{x_0, x_1, x_2 \dots\}$  che abbiamo costruito converga allo zero reale  $x^*$
2. Se il metodo converge possiamo provare a stimare una "velocità" di convergenza del metodo

### Definizione : convergenza

Una successione  $x_n \rightarrow x^*$  se  $\forall \varepsilon > 0 \exists \bar{n}$  tale che  $|x_n - x^*| < \varepsilon \quad \forall n > \bar{n}$

Cerchiamo di verificare se il metodo della bisezione converge :

$$|x_n - x^*| < \Delta_n = \frac{|b - a|}{2^n} = \varepsilon \rightarrow \bar{n} = \log_2 \frac{|b - a|}{\varepsilon}$$

Il metodo della bisezione converge sempre : data una precisione richiesta  $\varepsilon$  ci si può sempre arrivare a patto di fare  $\bar{n}$  bisezioni. Inoltre non dipende dal punto di innesco dell'algoritmo.

## Ricerca zeri di una funzione : convergenza e metodo della bisezione

### Definizione : ordine di convergenza

$x_n \rightarrow x^*$  con ordine  $p$  se  $\forall k > 0 \exists c > 0$  ( $c < 1$  se  $p = 1$ ) tale che

$$\frac{|x_{n+1} - x^*|}{|x_n - x^*|^p} \leq c \quad \forall n > k$$

(caveat : il metodo della bisezione non ha una convergenza monotona allo zero quindi tecnicamente non si potrebbe definire un ordine di convergenza. Si possono però fare comunque delle considerazioni )

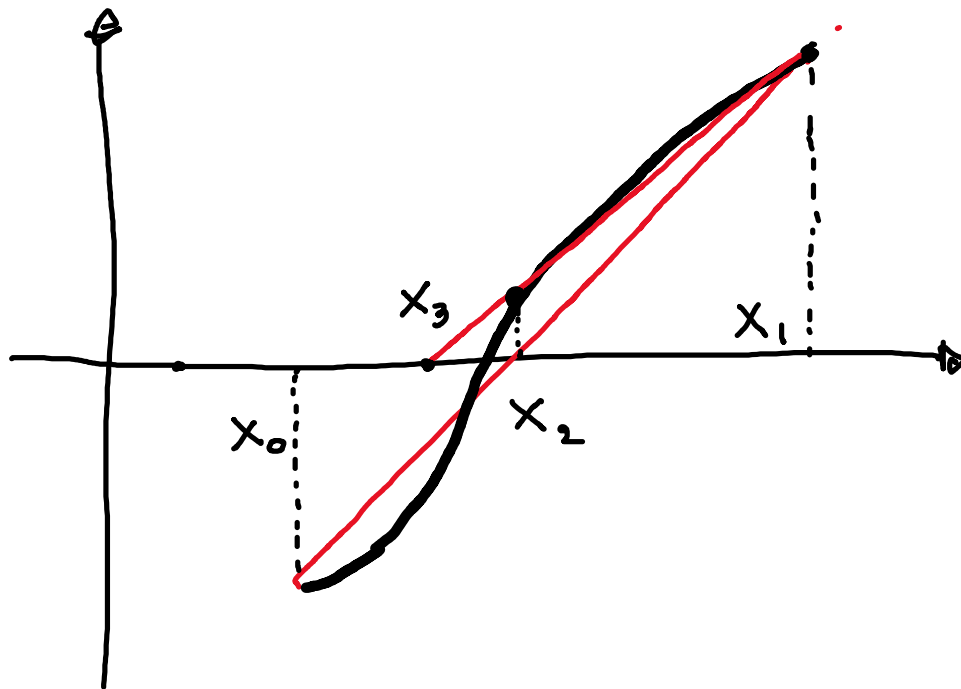
$$\begin{aligned} |x_{n+1} - x^*| &\cong \Delta_{n+1} = \frac{b-a}{2^{n+1}} \\ |x_{n+1} - x^*| &\leq c |x_n - x^*|^p \\ \frac{b-a}{2^{n+1}} &\leq c \left( \frac{b-a}{2^n} \right)^p \\ \text{const} \leq \frac{b-a}{2} &\leq c \frac{(b-a)^p}{2^{np-n}} = c \frac{(b-a)^p}{2^{n(p-1)}} \end{aligned}$$

Questa disuguaglianza può valere per ogni  $n$  solo se  $p=1$  ( e di conseguenza  $c = 1/2$  ) : convergenza lineare

## Ricerca zeri di una funzione : metodo delle secanti

Si può pensare di usare un secondo algoritmo iterativo per cercare lo zero di una funzione in un intervallo  $[a,b]$  :

- ❑ Definiamo i punti di innesco  $x_0 = a$  e  $x_1 = b$
- ❑ tracciamo la retta tra i punti  $P_0(x_0, f(x_0))$  e  $P_1(x_1, f(x_1))$  e consideriamo come approssimazione dello zero il punto  $x_2$  di intersezione della secante con l'asse delle  $x$ .
- ❑ Iteriamo la procedura tra il nuovo punto  $P_2(x_2, f(x_2))$  e il punto precedente  $P_1$



$$\frac{r(x) - f(x_1)}{f(x_0) - f(x_1)} = \frac{x - x_1}{x_0 - x_1}$$

$$r(x) = f(x_1) + \frac{f(x_0) - f(x_1)}{x_0 - x_1} (x - x_1)$$

$$r(x) = 0 \Rightarrow x = x_1 - \frac{f(x_1)(x_1 - x_0)}{f(x_1) - f(x_0)} = x_2$$

$$x_{k+1} = x_k - \frac{f(x_k)(x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$$



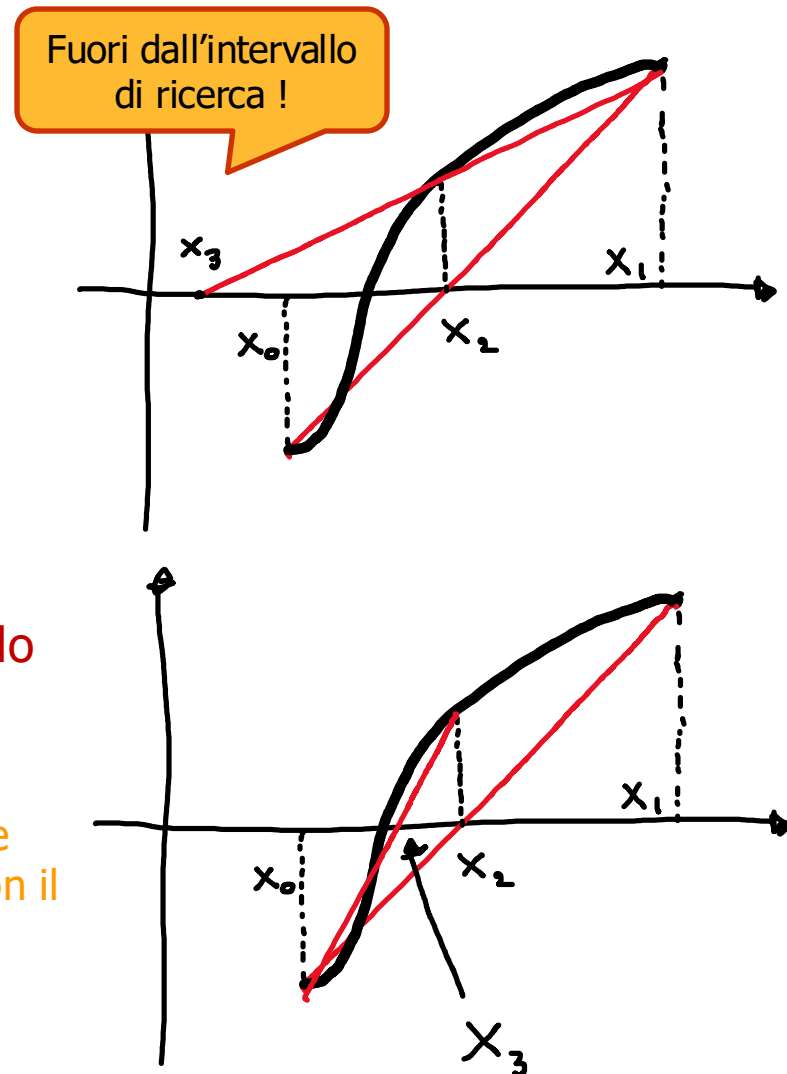
## Ricerca zeri di una funzione : metodo delle secanti

### Alcune considerazioni sull'algoritmo delle secanti

- ❑ Si può mostrare che l'ordine di convergenza è  $p = 1.6$  (superlineare)
- ❑ Convergenza non è garantita !
- ❑ Arbitrarietà nella definizione dello start ( trovato  $x_2$  faccio la secante con  $x_0$  o  $x_1$ )?

Algoritmo delle secanti migliorato con Regula Falsi : rende l'algoritmo di sicura convergenza (a spese dell'ordine di convergenza) introducendo un controllo sull'effettiva posizione dello zero :

- ❑ Una volta trovato  $x_2$  non procedo ciecamente a fare la secante con il punto precedente  $x_1$  ma vado a verificare se lo zero sta in  $[x_0, x_2]$  o  $[x_2, x_1]$  e faccio la secante con il punto scelto
- ❑ Bisezione con punto medio migliorato o secante con correzione dell'intervallo



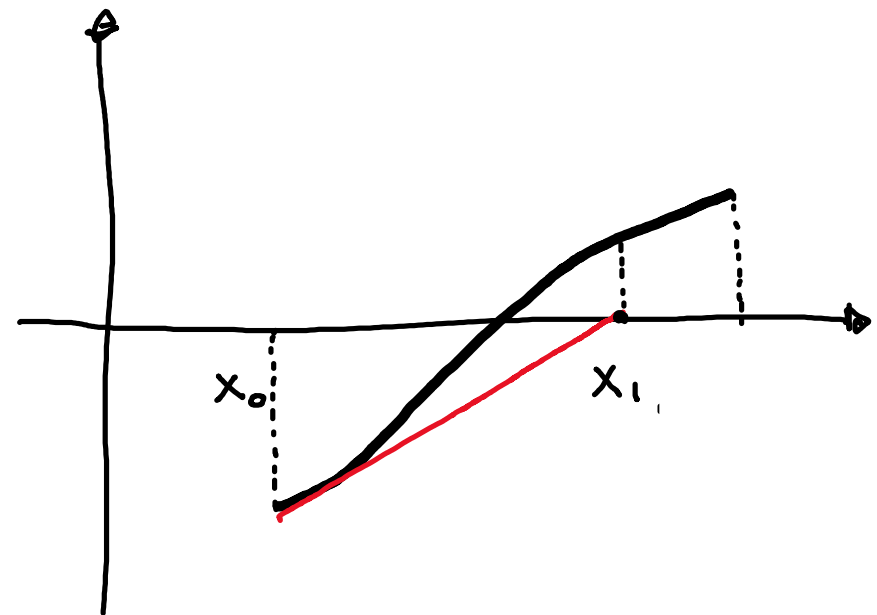
## Ricerca zeri di una funzione : metodo di Newton

Terzo algoritmo iterativo per cercare lo zero di una funzione in un intervallo  $[a, b]$  :

- ❑ Definiamo un punto di innesco  $x_0 = a$
- ❑ Partendo dal punto  $P_0(x_0, f(x_0))$  si traccia la tangente in quel punto e si considera come nuova approssimazione dello zero il punto  $x_1$  di intersezione della tangente con l'asse delle  $x$ .
- ❑ Iteriamo la procedura calcolando la tangente in  $P_1(x_1, f(x_1))$

$$r(x) = f(x_0) + f'(x_0)(x - x_0)$$
$$r(x) = 0 \Rightarrow x = x_0 - \frac{f(x_0)}{f'(x_0)} = x_1$$

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$



- ❑ Convergenza non è garantita !
- ❑ Arbitrarietà nella scelta del punto di innesco
- ❑ L'ordine di convergenza è  $p = 2$  (quadratico)
- ❑ Conoscenza della derivata della funzione !

## Intermezzo : calcolo della derivata di una funzione

- ❑ Un esempio molto istruttivo (che ci permetterà di fare interessanti osservazioni) di costruzione di un algoritmo è quello del calcolo numerico di una derivate.
  - ❑ Partiamo dalla cosa più semplice che possiamo pensare, scriviamo il rapporto incrementale con  $h$  "piccolo"

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h}$$

- ❑ Quanto piccolo  $h$  ? Stimiamo l'errore che commettiamo quando implementiamo questo calcolo nell'aritmetica finita del calcolatore : errore di troncamento ( $e_T$ ) ed errore di arrotondamento ( $e_r$ )
  - ❑ **Errore di troncamento** : scelto un valore per  $h$  possiamo stimare l'errore che commettiamo nell'assumere che il rapporto incrementale sia la derivata considerando lo sviluppo di Taylor

$$f(x_0 + h) - f(x_0) = f'(x_0)h + \frac{1}{2}h^2 f''(x_0) + \dots$$
$$f'(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + \frac{1}{2}hf''(x_0) + \dots$$


$$e_T \sim \frac{1}{2}hf''(x_0)$$

## Intermezzo : calcolo della derivata di una funzione

- **Errore di arrotondamento** : quando rappresento nella macchina il rapporto incrementale che uso come approssimazione della derivata. Di quanto mi sbaglio ?

$$\begin{aligned}\tilde{f}(x_0 + h) &= f(x_0 + h)(1 + \varepsilon_1) \\ \tilde{f}(x_0) &= f(x_0)(1 + \varepsilon_2)\end{aligned}$$

Assumendo  $h$  molto piccolo (quindi con errore trascurabile)

$$\frac{\tilde{f}(x_0 + h) - \tilde{f}(x_0)}{h} = \frac{f(x_0 + h) - f(x_0)}{h} + \frac{f(x_0 + h)\varepsilon_1 - f(x_0)\varepsilon_2}{h}$$

$\rightarrow \varepsilon_r$

assumendo  $f(x_0 + h) \sim f(x_0)$

$$\frac{f(x_0 + h)\varepsilon_1 - f(x_0)\varepsilon_2}{h} \approx \frac{f(x_0)(\varepsilon_1 - \varepsilon_2)}{h} < \frac{f(x_0)2\varepsilon_M}{h}$$

$$\varepsilon_r = \frac{f(x_0)2\varepsilon_M}{h}$$

## Intermezzo : calcolo della derivata di una funzione

Calcoliamo quindi l'errore totale che si commette nel calcolo della derivate :

$$e_{tot}(h) = e_r(h) + e_T(h) = \frac{1}{2}f''(x_0)h + 2\varepsilon_M f(x_0)\frac{1}{h}$$

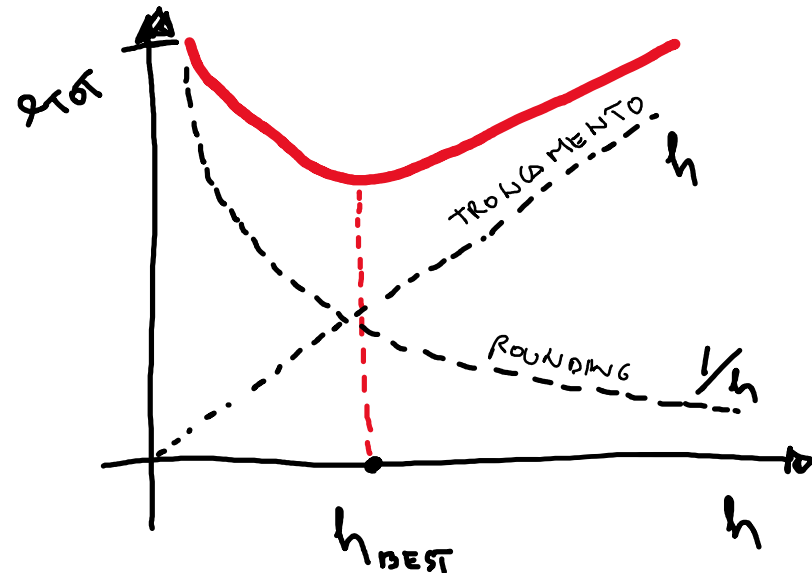
- Cerchiamo di capire se esiste un passo ottimale da utilizzare per minimizzare l'errore totale. Deriviamo l'espressione rispetto ad  $h$  e vediamo quando la derivata si annulla

$$\frac{de_{tot}}{dh} = \frac{1}{2}f''(x_0) - 2\varepsilon_M f(x_0)\frac{1}{h^2} = 0 \Rightarrow$$

$$\frac{1}{2}f''(x_0) = 2\varepsilon_M f(x_0)\frac{1}{h_{best}^2}$$

$$\Rightarrow h_{best} = \sqrt{\varepsilon_M \frac{4f(x_0)}{f''(x_0)}} \Rightarrow h_{best} = k\sqrt{\varepsilon_M}$$

$$e_{tot}^{best} = k'\sqrt{\varepsilon_M}$$



Attenzione quindi : il passo  $h$  non può essere scelto 'troppo' piccolo !

## Intermezzo : calcolo della derivata di una funzione

Possiamo fare di meglio? Costruire un algoritmo che permetta una precisione maggiore.

❑ Proviamo ad agire sull'errore di truncamento. Sviluppiamo la funzione

$$f(x_0 + h) = f(x_0) + hf^1(x_0) + \frac{1}{2}h^2f''(x_0) + \frac{1}{6}h^3f'''(x_0) + \frac{1}{24}h^4f^{IV}(x_0) + \frac{1}{120}h^5f^V(x_0) + \dots$$

$$f^1(x_0) = \frac{f(x_0 + h) - f(x_0)}{h} + \frac{1}{2}hf''(x_0) + o(h^2)$$

- ❑ Non sembra una grande idea: per migliorare il calcolo della derivata prima dobbiamo conoscere la derivata seconda.
- ❑ Cerchiamo un modo furbo di procedere : sviluppiamo la funzione da  $x_0$  in avanti ( $x_0+h$ ) e indietro ( $x_0-h$ )

$$\begin{aligned} f(x_0 + h) &= f(x_0) + hf^1(x_0) + \frac{1}{2}h^2f''(x_0) + \frac{1}{6}h^3f'''(x_0) + \frac{1}{24}h^4f^{IV}(x_0) + \frac{1}{120}h^5f^V(x_0) + \dots \\ f(x_0 - h) &= f(x_0) - hf^1(x_0) + \frac{1}{2}h^2f''(x_0) - \frac{1}{6}h^3f'''(x_0) + \frac{1}{24}h^4f^{IV}(x_0) - \frac{1}{120}h^5f^V(x_0) + \dots \end{aligned}$$

## Intermezzo : calcolo della derivata di una funzione

❑ Sottraiamo le due espressioni ottenute in questo modo

$$\begin{aligned} f(x_0 + h) &= f(x_0) + hf'(x_0) + \frac{1}{2}h^2f''(x_0) + \frac{1}{6}h^3f'''(x_0) + \frac{1}{24}h^4f^{IV}(x_0) + \frac{1}{120}h^5f^V(x_0) + \dots \\ f(x_0 - h) &= f(x_0) - hf'(x_0) + \frac{1}{2}h^2f''(x_0) - \frac{1}{6}h^3f'''(x_0) + \frac{1}{24}h^4f^{IV}(x_0) - \frac{1}{120}h^5f^V(x_0) + \dots \\ \hline f(x_0 + h) - f(x_0 - h) &= 2hf'(x_0) + \frac{2}{6}h^3f'''(x_0) + \frac{2}{120}h^5f^V(x_0) + \dots \end{aligned}$$

❑ Da questa differenza possiamo ricavare una nuova espressione per la derivata

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} - \frac{1}{3}h^2f'''(x_0) - \frac{1}{60}h^4f^V(x_0) + \dots$$

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} + o(h^2)$$

❑ ... che adesso ha un errore di ordine  $h^2$  ! (e non richiede di conoscere la derivata seconda 😊)

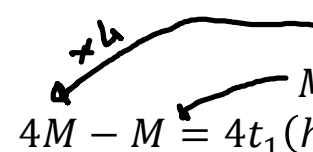
## Intermezzo : calcolo della derivata di una funzione

### Possiamo fare ancora meglio ? Estrapolazione di Richardson

□ Riprendiamo l'espressione della derivate estratta nella trasparenza precedente

$$f'(x_0) = \frac{f(x_0 + h) - f(x_0 - h)}{2h} - \frac{1}{3}h^2 f'''(x_0) - \frac{1}{60}h^4 f^{(5)}(x_0) + \dots$$

□ Ogni volta che incontriamo espressioni del genere possiamo valutare questa espressione in  $h$  e  $2h$  e poi sottrarre opportunamente le due per cancellare termini di errore crescenti.


$$\begin{aligned} M &= t_1(h) + k_1 h^2 + k_2 h^4 + o(h^6) \\ M &= t_1(2h) + k_1 (2h)^2 + k_2 (2h)^4 + o(h^6) \\ 4M - M &= 4t_1(h) - t_1(2h) + 4k_1 h^2 - 4k_1 h^2 + 4k_2 h^4 - 16k_2 h^4 + o(h^6) \\ M &= \frac{4t_1(h) - t_1(2h)}{3} - 4k_2 h^4 + o(h^6) \\ f'(x_0) &= \frac{1}{3} \left[ 4 \frac{f(x_0 + h) - f(x_0 - h)}{2h} - \frac{f(x_0 + 2h) - f(x_0 - 2h)}{4h} \right] + o(h^4) \end{aligned}$$

$$f'(x_0) = \left[ \frac{f(x_0 - 2h) - 8f(x_0 - h) + 8f(x_0 + h) - f(x_0 + 2h)}{12h} \right] + o(h^4)$$

□ Ora l'errore è un  $o(h^4)$  !



## Intermezzo : calcolo della derivata di una funzione

Vediamo a questo punto l'errore con nuovo metodo ( estrapolazione di Richardson )

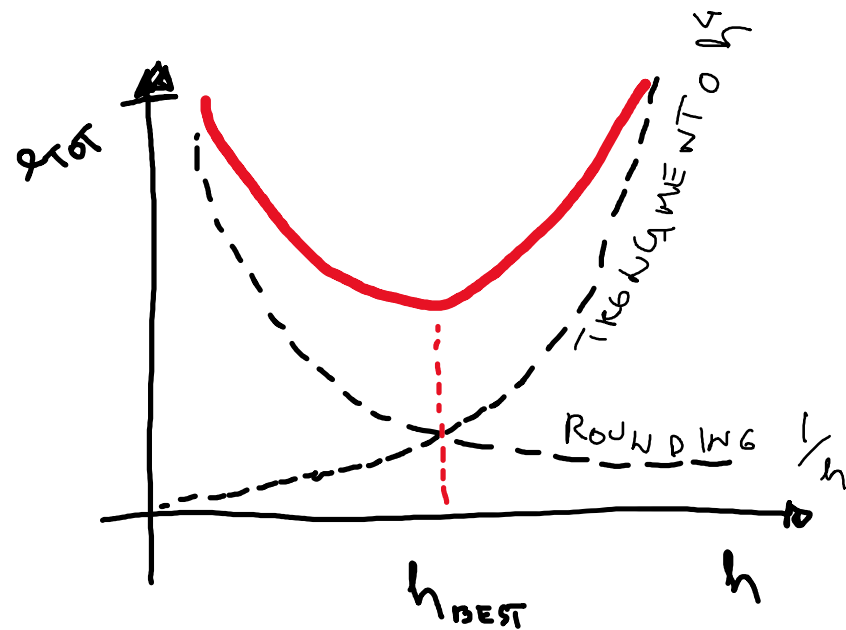
$$e_{tot} = e_r + e_T = 2\varepsilon_M f(x_0) \frac{1}{h} + k_1 h^4$$

$$\frac{de_{tot}}{dh} = -2\varepsilon_M f(x_0) \frac{1}{h^2} + 4k_1 h^3 = 0$$

$$\Rightarrow 2\varepsilon_M f(x_0) \frac{1}{h^2} = 4k_1 h^3$$

$$h_{best} = k_2 \sqrt[5]{\varepsilon_M}$$

$$\Rightarrow e_{best} = k \frac{\varepsilon_M}{\sqrt[5]{\varepsilon_M}} + k_3 \varepsilon_M^{4/5} \propto \varepsilon_M^{4/5}$$



---

## Codificare la ricerca di zeri di una funzione



## Usecase : ricerca zeri di una funzione

### Codifichiamo una funzione :

- ❑ `class Parabola`
- ❑ costruttori, metodi `set/get` + tutto ciò che si ritiene opportuno
- ❑ metodo dedicato `Eval(x)` che svolge la funzione di  $f(x)$
- ❑ Possiamo anche fare `overloading di operator()` se non vogliamo invocare il metodo `Eval()`

```
class Parabola {  
  
    public:  
  
        Parabola() { m_a = 0.; m_b = 0.; m_c = 1.; }  
        Parabola(double a, double b, double c) { m_a = a; m_b = b; m_c = c; }  
        double Eval(double x) const {return m_a*x*x+m_b*x+m_c;}  
  
        // ...  
  
    private:  
  
        double m_a, m_b, m_c;  
  
};
```

## Usecase : ricerca zeri di una funzione

```
class Bisezione {  
public:  
    Bisezione() ;  
    ~Bisezione();  
  
    // ...  
  
    double CercaZeri(double xmin,  
                     double xmax,  
                     const Parabola& f,  
                     double prec=0.001,  
                     unsigned int nmax=100);  
  
    // ...  
  
private:  
  
    double m_a, m_b;  
    double m_prec;  
    unsigned int m_nmax, m_niterations;  
};
```

Valore di default in caso venga omissso.

### Codifichiamo l'algoritmo:

- ❑ Class Bisezione : immagazzina gli estremi dell'intervallo, precision richiesta, numero (massimo) di iterazioni
- ❑ costruttori, metodi set/get (potete passare parametri al costruttore o modificarli dopo, come preferite )
- ❑ metodo dedicato CercaZeri() : questo calcolerà il valore dello zero della funzione
- ❑ m\_nmax (massimo numero di iterazioni permesse ) e m\_niterations ( numero di iterazioni effettuate ) possono essere data-membri della classe

## Usecase : ricerca zeri di una funzione

```
class Segno {  
public:  
    double Eval(double x) const {  
        return (x==0.?0.:(x>0?1.: -1));  
    }  
};
```

- ❑ ritorna 0 se  $x=0$ ,
- ❑ ritorna 1 se  $x>0$  e
- ❑ ritorna -1 se  $x<0$

```
double Bisezione::CercaZeri(double xmin,  
                             double xmax,  
                             const Parabola & f,  
                             double prec ,  
                             unsigned int nmax ) {  
  
    Segno sign;  
  
    m_niterations = 0;  
    m_prec = prec ;  
    m_nmax = nmax ;  
  
    if ( xmin<xmax ) {  
        m_a = xmin; m_b = xmax;  
    } else {  
        m_a = xmax; m_b = xmin;  
    }  
  
    double fa = f.Eval(m_a);  
    double fb = f.Eval(m_b);  
  
    while ( fabs(m_b-m_a) > m_prec ) {  
        double c = 0.5*(m_b+m_a);  
        double fc = f.Eval(c);  
        if ( m_niterations > m_nmax ) break;  
        m_niterations++ ;  
        if ( sign(fa)*sign(fc) <= 0 ) {  
            m_b=c; fb=fc;  
        } else if ( sign(fb)*sign(fc)<=0 ) {  
            m_a=c; fa=fc;  
        } else return 0.;  
    }  
    return 0.5*(m_b+m_a);  
};
```

Ciclo principale con arresto sulla precisione

Controllo ausiliario sul  
numero di iterazioni

Sostituiamo il controllo  
 $f(a) * f(b)$  con  $\text{sign}(f(a)) * \text{sign}(f(b))$ , più robusto  
contro errori di rounding

## Usecase : ricerca zeri di una funzione

```
#include "Funzioni.h"
#include "AlgoZeri.h"
#include <iostream>

using namespace std;

int main() {
    Bisezione bisettore ;
    Parabola f(3,5,-2) ;
    cout << bisettore.CercaZeri(0.9,1.1,f,0.001,100) << endl;
}
```

algoritmo

funzione

Ricerca dello zero con  
metodo bisezione

Codice funziona, ma non sembra granchè. Supponiamo di voler calcolare gli zeri di un'altra funzione (Coseno) : devo scrivere una nuova classe `Bisezione` ?

```
class Bisezione {
public :

    Bisezione();
    ~Bisezione();

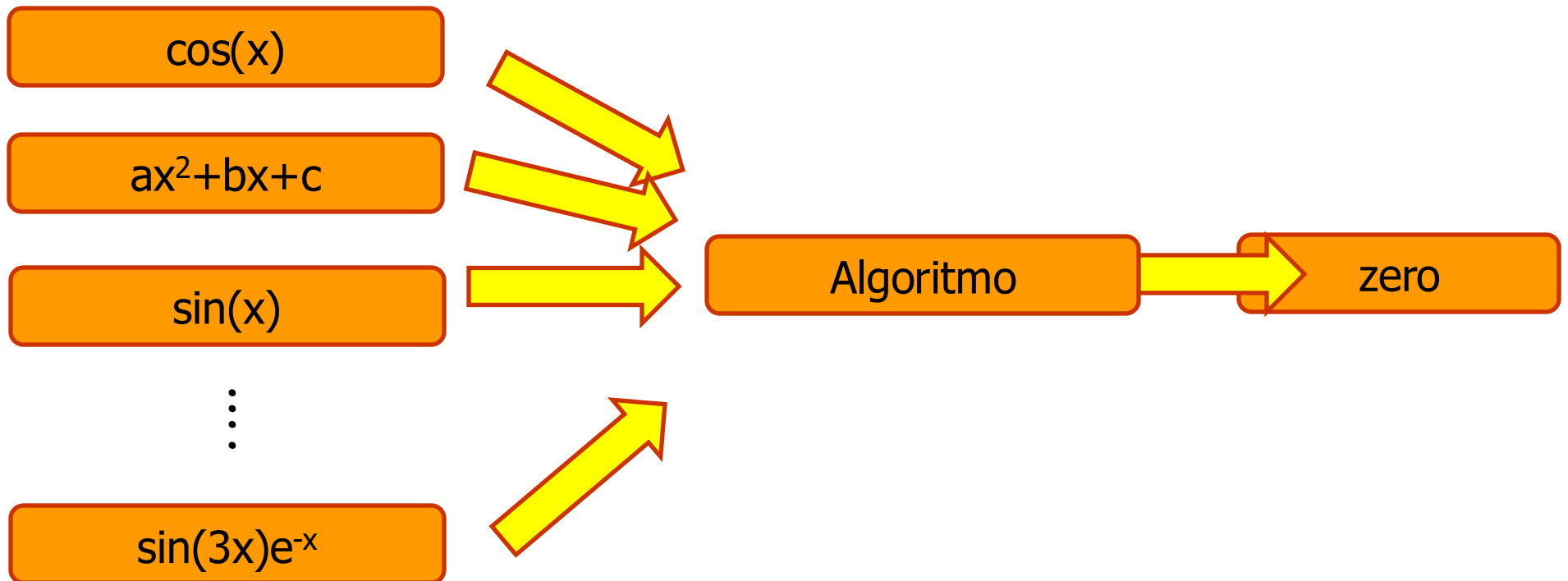
    ...

    double CercaZeri(double xmin, double xmax, const Parabola& f , double prec );
};
```

const Coseno& f

## Usecase : ricerca zeri di una funzione

Molte funzioni diverse ma un solo algoritmo !



## Polimorfismo : varie definizioni

1) La parola polimorfismo indica la capacità di qualcosa di assumere forme diverse. Il polimorfismo è una delle caratteristiche essenziali della programmazione orientata agli oggetti. In termini più tecnici, possiamo definire il polimorfismo come la caratteristica della programmazione orientata agli oggetti che consente all'oggetto della classe di comportarsi diversamente in circostanze diverse.

2) Una delle caratteristiche chiave dell'ereditarietà delle classi è che un puntatore a una classe derivata è type-compatibile con un puntatore alla sua classe base. Il polimorfismo è l'arte di sfruttare questa caratteristica semplice ma potente e versatile.

3) Il polimorfismo del C++ significa che una chiamata a un metodo della classe causerà l'esecuzione di una funzione diversa a seconda del tipo di oggetto che richiama la funzione.





## Polimorfismo : step I

Step I : una delle caratteristiche chiave dell'ereditarietà delle classi è che un puntatore a classe figlia è type-compatible con un puntatore alla sua classe madre.

```
class FunzioneBase {
public :
    double Eval(double x) const
    { return 0; };
};

class Parabola : public FunzioneBase {
public :
    double Eval(double x) const
    { return m_a*x*x + m_b*x + m_c ; };
};

class Coseno : public FunzioneBase {
public :
    double Eval(double x) const
    { return cos(x) ; };
};
```

Molto interessante da sapere ma ancora non se ne capisce l'utilità

```
int main () {
    FunzioneBase * myfbase = new FunzioneBase();
    Parabola * mypara = new Parabola();
    Coseno * mycos = new Coseno();

    FunzioneBase * myfpoly1 = new Parabola();
    FunzioneBase * myfpoly2 = new Coseno();

    cout << myfbase->Eval(10.) << endl;
    cout << mypara->Eval(10.) << endl;
    cout << mycos->Eval(10.) << endl;

    cout << myfpoly1->Eval(10.) << endl;
    cout << myfpoly2->Eval(10.) << endl;
}
```

## Polimorfismo : step II

Step II : un metodo virtuale è una funzione membro che può essere ridefinita in una classe derivata, preservandone le proprietà di chiamata tramite i riferimenti. Una classe che dichiara o eredita una funzione virtuale è chiamata classe polimorfica o classe virtuale

```
class FunzioneBase {  
    public :  
    virtual double Eval(double x) const  
    { return 0; };  
};  
  
class Parabola : public FunzioneBase {  
    public :  
    virtual double Eval(double x) const  
    { return m_a*x*x + m_b*x + m_c ; }  
};  
  
class Coseno : public FunzioneBase {  
    public :  
    virtual double Eval(double x) const  
    { return cos(x) ; }  
};
```

## Polimorfismo : step II

Step II : qual'è l'effetto di dichiarare un metodo come `virtual` ? Possiamo toccarne con mano l'effetto nel contesto del polimorfismo :

```
class FunzioneBase {  
public :  
    virtual double Eval(double x) const  
    { return 0; };  
};  
  
class Parabola : public FunzioneBase {  
public :  
    virtual double Eval(double x) const  
    { return m_a*x*x + m_b*x + m_c ; }  
};  
  
class Coseno : public FunzioneBase {  
public :  
    virtual double Eval(double x) const  
    { return cos(x) ; }  
};
```

Fuochino !

```
int main () {  
    FunzioneBase * myfbase = new FunzioneBase() ;  
    Parabola * mypara = new Parabola () ;  
    Coseno * mycos = new Coseno() ;  
  
    FunzioneBase * myfpoly1 = new Parabola() ;  
    FunzioneBase * myfpoly2 = new Coseno();  
  
    cout << myfbase->Eval(10.) << endl;  
    cout << mypara->Eval(10.) << endl;  
    cout << mycos->Eval(10.) << endl;  
  
    cout << myfpoly1->Eval(10.) << endl;  
    cout << myfpoly2->Eval(10.) << endl;  
}
```

## Polimorfismo : step II

L'input è di tipo `FunzioneBase*`  
( non `Parabola*` ) !

```
#include "Funzioni.h"
#include "AlgoZeri.h"
#include <iostream>

using namespace std;

int main() {

    Bisezione bisettore ;

    Parabola * f = new Parabola(3,5,-2) ;

    cout << bisettore.CercaZeri(0.9 ,1.1 ,f ,0.001 ) << endl;
}
```

- ❑ `CercaZeri` accetta in input `FunzioneBase*`. Lavora con una "generica funzione"
- ❑ Nel `main` passiamo un puntatore a parabola : e' possibile, i due tipi sono compatibili
- ❑ Le classi sono polimorfiche : in `CercaZeri` le chiamate a `f->Eval()` dipendono dal puntatore che viene passato
- ❑ `CercaZeri` è lo stesso per qualsiasi funzione che eredita da `FunzioneBase` !

```
double Bisezione::CercaZeri(double xmin,
                             double xmax,
                             const FunzioneBase * f,
                             double prec ,
                             unsigned int nmax ) {

    Segno sign;

    m_niterations = 0;
    m_prec = prec ;
    m_nmax = nmax ;

    if ( xmin<xmax ) {
        m_a = xmin; m_b = xmax;
    } else {
        m_a = xmax; m_b = xmin;
    }

    double fa = f->Eval(m_a);
    double fb = f->Eval(m_b);

    while ( (m_b-m_a) > m_prec ) {
        double c = 0.5*(m_b+m_a);
        double fc = f->Eval(c);
        if ( m_niterations > m_nmax ) break;
        m_niterations++ ;
        if ( sign.Eval(fa)*sign.Eval(fc) <= 0 ) {
            m_b=c; fb=fc;
        } else if ( sign.Eval(fb)*sign.Eval(fc)<=0 ) {
            m_a=c; fa=fc;
        } else return 0.;
    }

    return 0.5*(m_b+m_a);
}
```

## Polimorfismo : step II

```
#include "Funzioni.h"
#include "AlgoZeri.h"
#include <iostream>

using namespace std;

int main() {

    Bisezione bisettore ;

    Parabola f ;

    cout << bisettore.CercaZeri(0.9, 1.1, f, 0.001 ) << endl;

}
```

No need to use pointers!

- ❑ La funzione `CercaZeri` può anche essere riscritta in modo che accetti una reference come input ( di fatto cattura il puntatore )
- ❑ Quindi non c'è bisogno di creare esplicitamente un puntatore a funzione nel `main` ( safer ! )

L'input è di tipo `FunzioneBase&`  
( non `Parabola*` ) !

```
double Bisezione::CercaZeri(double xmin,
                             double xmax,
                             const FunzioneBase & f,
                             double prec ,
                             unsigned int nmax ) {

    Segno sign;

    m_niterations = 0;
    m_prec = prec ;
    m_nmax = nmax ;

    if ( xmin < xmax ) {
        m_a = xmin; m_b = xmax;
    } else {
        m_a = xmax; m_b = xmin;
    }

    double fa = f.Eval(m_a);
    double fb = f.Eval(m_b);

    while ( (m_b - m_a) > m_prec ) {
        double c = 0.5*(m_b + m_a);
        double fc = f.Eval(c);
        if ( m_niterations > m_nmax ) break;
        m_niterations++;
        if ( sign.Eval(fa)*sign.Eval(fc) < 0 ) {
            m_b = c; fb = fc;
        } else if ( sign.Eval(fa)*sign.Eval(fc) > 0 ) {
            m_a = c; fa = fc;
        }
    }

    return 0.5*(m_b + m_a);

}
```

## Polimorfismo : step III

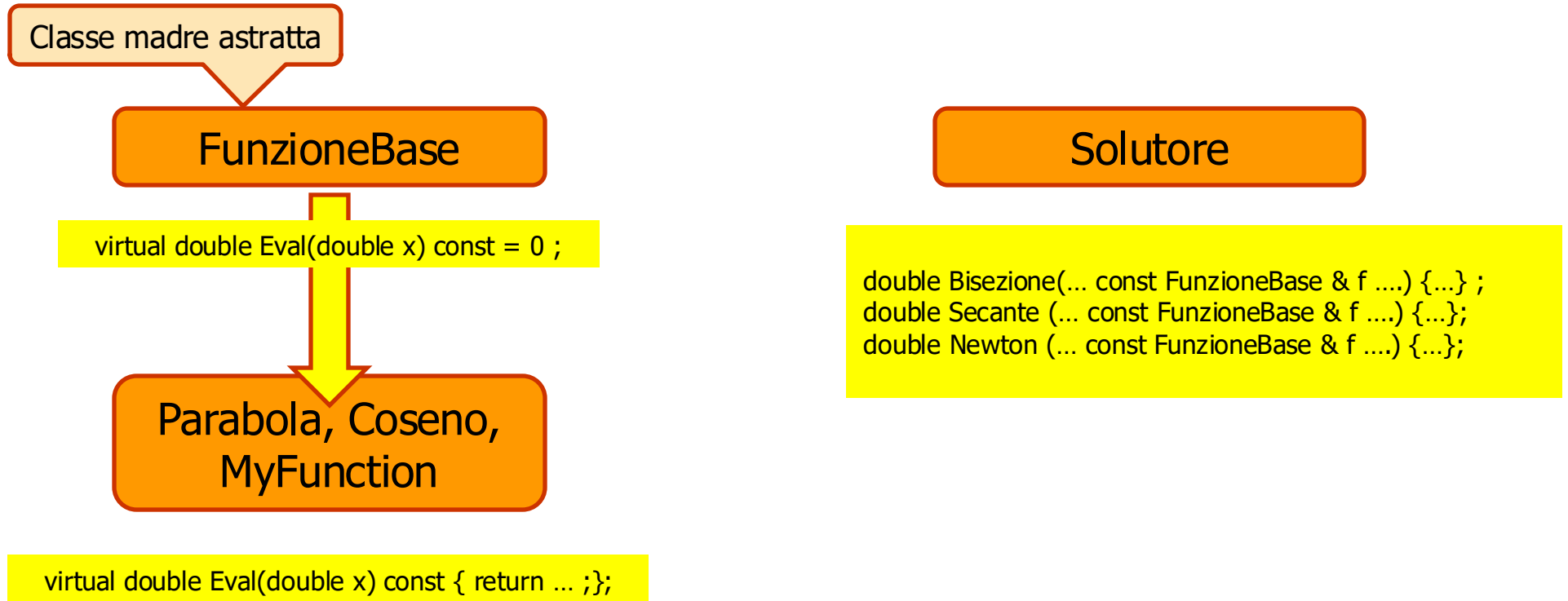
Step III: posso costruire una classe madre con metodi virtuali puri ovvero dichiarati ma non implementati (=0). Una classe di questo tipo si dice astratta. Le classi figlie dovranno obbligatoriamente implementare i metodi virtuali puri della classe base

```
class FunzioneBase {  
    public :  
    virtual double Eval(double x) const = 0 ;  
};  
  
class Parabola : public FunzioneBase {  
    public :  
    virtual double Eval(double x) const override  
    { return m_a*x*x + m_b*x + m_c ; }  
};  
  
class Coseno : public FunzioneBase {  
    public :  
    virtual double Eval(double x) const override  
    { return cos(x) ; }  
};
```

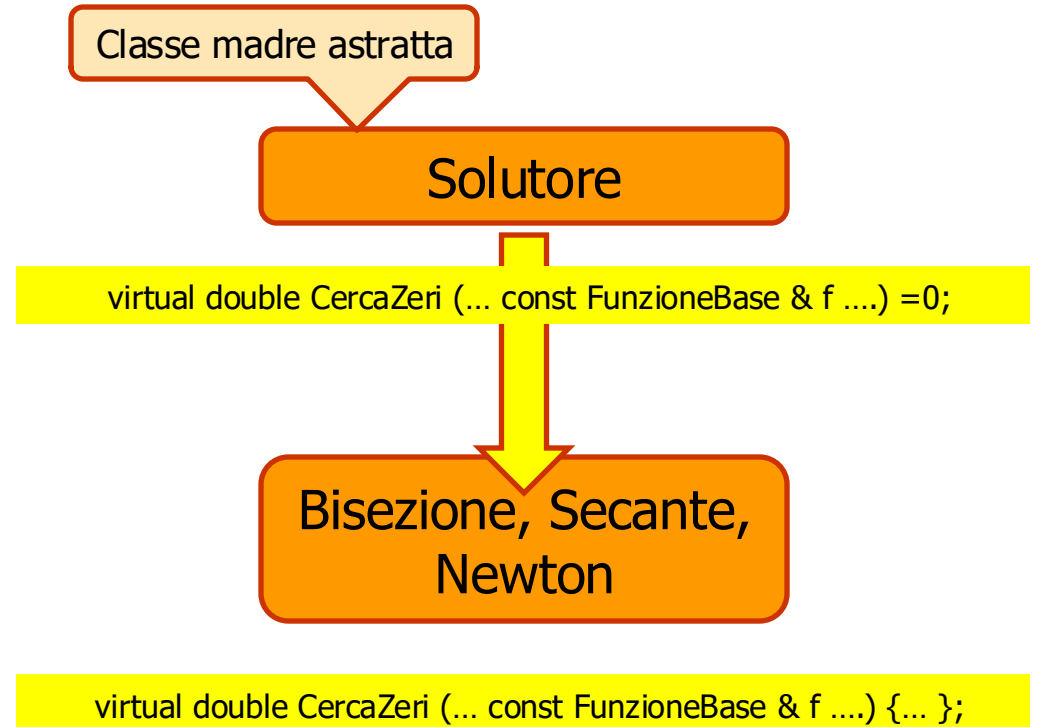
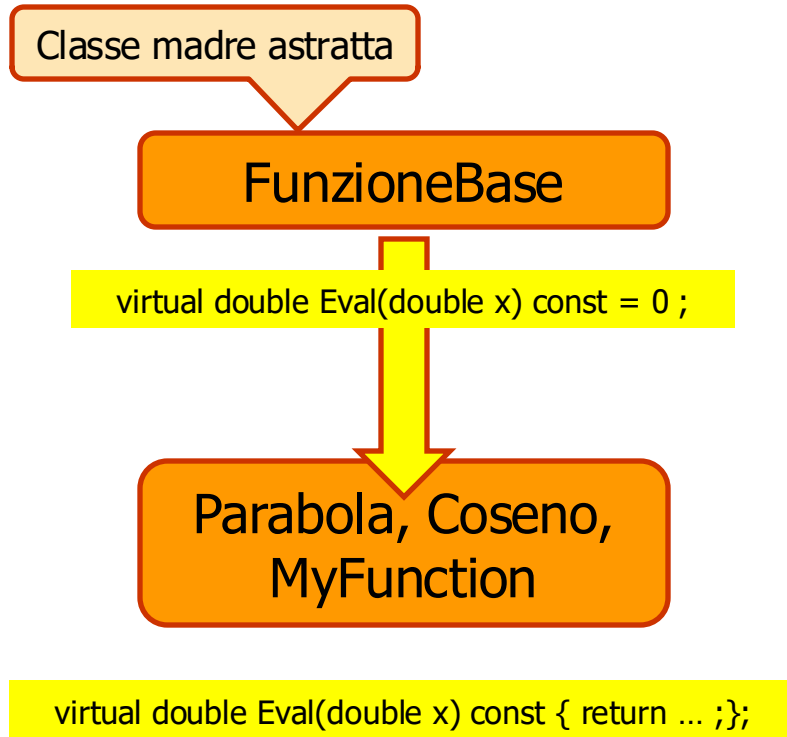
La keyword `override` fa sì che un errore venga segnalato se il metodo virtuale con lo stesso nome viene sovrascritto nella classe figlia con qualche parametro diverso

`virtual` nelle classi figlie può essere omissa

## Possibile struttura delle classi (I)



## Possibile struttura delle classi (II)





## Una nota sui distruttori di classi virtuali

Ogni volta che la classe ha almeno una funzione virtuale, anche il distruttore dovrebbe essere dichiarato virtuale

```
#include <iostream>

using namespace std;

class Base{

    virtual ~Base() { cout << "Base destructor called" << endl; };

    virtual void method() = 0 ;

};

class Derived : public Base {

    ~Base() { cout << "Base destructor called" << endl; };

    void method() = { cout << "Derived::method() called" << endl;};

};

int main() {

    Base * myclass = new Derived() ;

    myclass->method();

    delete myclass ;

}
```

Se il distruttore della classe base non è dichiarato virtuale, `delete myclass` chiamerà il distruttore della classe base e non quello derivato. Un distruttore virtuale è più sicuro poiché tutto ciò che è posseduto dalla classe derivata verrà deallocato correttamente

## Il polimorfismo non è l'unico modo (only for curious kids)

### ❑ La funzione può essere un puntatore a funzione, un funtore o una funzione lambda

```
// Puntatore a funzione parabola
double ParaFunction( double x ) {return 3*x*x - 5*x + 2 ;};

// Funtore che rappresenta la funzione parabola
class ParaFunctor {
public :

    ParaFunctor() { m_para1 = 1. ; m_para2 = 1.; m_para3 = 1. ; } ;
    ParaFunctor( double para1, double para2, double para3 ) {
        m_para1 = para1 ;
        m_para2 = para2 ;
        m_para3 = para3 ;
    } ;

    void SetParams( double para1, double para2, double para3 ) {
        m_para1 = para1 ;
        m_para2 = para2 ;
        m_para3 = para3 ;
    } ;

    double operator() ( double x ) {return m_para1 * x*x + m_para2 * x + m_para3 ;} ;

private :

    double m_para1, m_para2 , m_para3 ;
} ;

// lambda function ( no capture )
auto lambda = [](double x)->double { return 3*x*x - 5 * x + 2 ; };

// lambda function ( with parameter capture )
auto lambdaCapture [&](double x)->double {return para1 * x*x + para2 * x + para3 ;};
```

Puntatore a funzione (non è possibile passare parametri esterni)

Un funtore è fondamentalmente una classe e usiamo l'`operator()` per imitare una funzione. I parametri possono essere passati attraverso i costruttori o con il metodo `SetParams()`

Lambda function senza cattura di parametri

Lambda function con cattura dei parametri del `main`

## Il polimorfismo non è l'unico modo (only for curious kids)

- ❑ Se non piace il polimorfismo, esiste un approccio alternativo che può essere utilizzato per codificare l'algoritmo creando funzioni (metodi) che accettano puntatori a funzioni, funtori o funzioni lambda

```
// Funzione che accetta un puntatore a funzione
double BisezioneFunPointer(double xmin,
                           double xmax,
                           double (*f)(double),
                           double prec=0.001,
                           unsigned int nmax=100)
{
    unsigned int niterations = 0;
    double a = 0 ;
    double b = 0 ;
    if ( xmin<xmax ) {
        a = xmin;
        b = xmax;
    } else {
        a = xmax;
        b = xmin;
    }
    double fa = f(a);
    double fb = f(b);
    while ( fabs(b-a) > prec ) {
        double c = 0.5*(b+a);
        double fc = f(c);
        if ( niterations > nmax ) break;
        niterations++;
        if ( sign(fa)*sign(fc) <= 0 ) { b=c; fb=fc;}
        else if ( sign(fb)*sign(fc)<=0 ) { a=c; fa=fc;}
        else return 0.;
    }
    return 0.5*(b+a);
}
```

```
// Funzione template
template<typename Fun> double BisezioneTemplate(double xmin,
                                                double xmax,
                                                Fun f,
                                                double prec=0.001,
                                                unsigned int nmax=100)
{
    unsigned int niterations = 0;
    double a = 0 ;
    double b = 0 ;
    if ( xmin<xmax ) {
        a = xmin;
        b = xmax;
    } else {
        a = xmax;
        b = xmin;
    }
    double fa = f(a);
    double fb = f(b);
    while ( fabs(b-a) > prec ) {
        double c = 0.5*(b+a);
        double fc = f(c);
        if ( niterations > nmax ) break;
        niterations++;
        if ( sign(fa)*sign(fc) <= 0 ) { b=c; fb=fc;}
        else if ( sign(fb)*sign(fc)<=0 ) { a=c; fa=fc;}
        else return 0.;
    }
    return 0.5*(b+a);
}
```

```
// Funzione che accetta una std::function
double BisezioneStdFun(double xmin,
                       double xmax,
                       std::function<double (double)> f,
                       double prec=0.001,
                       unsigned int nmax=100 )
{
    unsigned int niterations = 0;
    double a = 0 ;
    double b = 0 ;
    if ( xmin<xmax ) {
        a = xmin;
        b = xmax;
    } else {
        a = xmax;
        b = xmin;
    }
    double fa = f(a);
    double fb = f(b);
    while ( fabs(b-a) > prec ) {
        double c = 0.5*(b+a);
        double fc = f(c);
        if ( niterations > nmax ) break;
        niterations++;
        if ( sign(fa)*sign(fc) <= 0 ) { b=c; fb=fc;}
        else if ( sign(fb)*sign(fc)<=0 ) { a=c; fa=fc;}
        else return 0.;
    }
    return 0.5*(b+a);
}
```

## Il polimorfismo non è l'unico modo (only for curious kids)

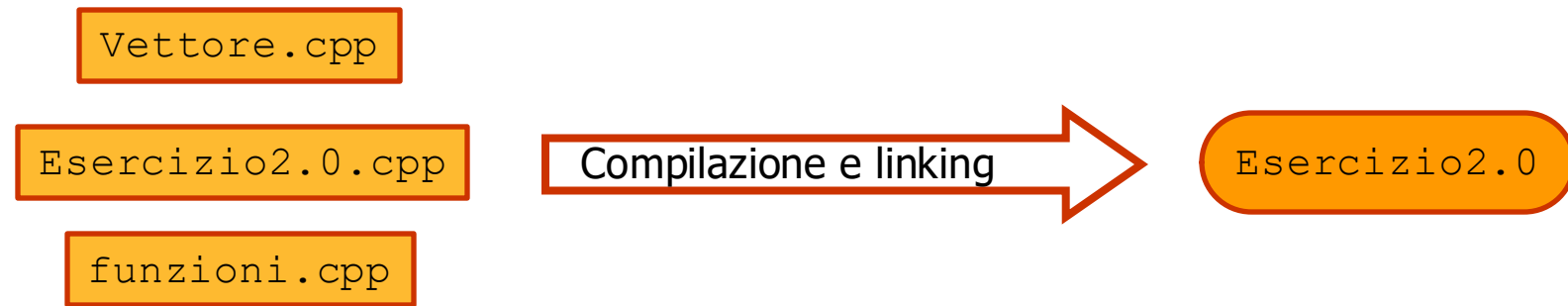
Esempio su come usare questi ingredient nel `main()` :

```
int main() {  
    // BisezioneFunPointer only works with function pointers  
    std::cout << BisezioneFunPointer(0.8,1.2,ParaFunction) << std::endl ;  
  
    // also with lambdas if there's no capture  
    auto lambda = [](double x)->double { return 3*x*x - 5 * x + 2 ; };  
    std::cout << BisezioneFunPointer(0.9,1.1, lambda ) << endl;  
  
    std::cout << BisezioneFunPointer(0.9,1.1, [](double x)->double { return 3*x*x - 5 * x + 2 ; } ) << endl;  
  
    // The other two methods work with function pointers, functors and lambdas  
    ParaFunctor myParaFunctor ( 3., -5., 2. );  
    std::cout << BisezioneStdFun(0.9,1.1,myParaFunctor ) << std::endl ;  
  
    std::cout << BisezioneTemplate(0.9,1.1, ParaFunction ) << endl;  
    std::cout << BisezioneTemplate(0.9,1.1, myParaFunctor ) << endl;  
    std::cout << BisezioneTemplate(0.9,1.1, [&](double x)->double { return 3*x*x - 5 * x + 2 ; } ) << endl;  
  
    std::cout << BisezioneStdFun(0.9,1.1, ParaFunction ) << endl;  
    std::cout << BisezioneStdFun(0.9,1.1, myParaFunctor ) << endl;  
    std::cout << BisezioneStdFun(0.9,1.1, [](double x)->double {return 3*x*x - 5 * x + 2 ; } ) << endl;  
  
    // check here, a lambda with capture  
    double para1 = 3;  
    double para2 = -5;  
    double para3 = 2;  
  
    std::cout << BisezioneTemplate(0.9,1.1, [&](double x)->double {return para1 * x*x + para2 * x + para3 ;} ) << endl;  
    std::cout << BisezioneStdFun(0.9,1.1, [&](double x)->double {return para1 * x*x + para2 * x + para3 ;} ) << endl;  
}
```

---

**Backup**

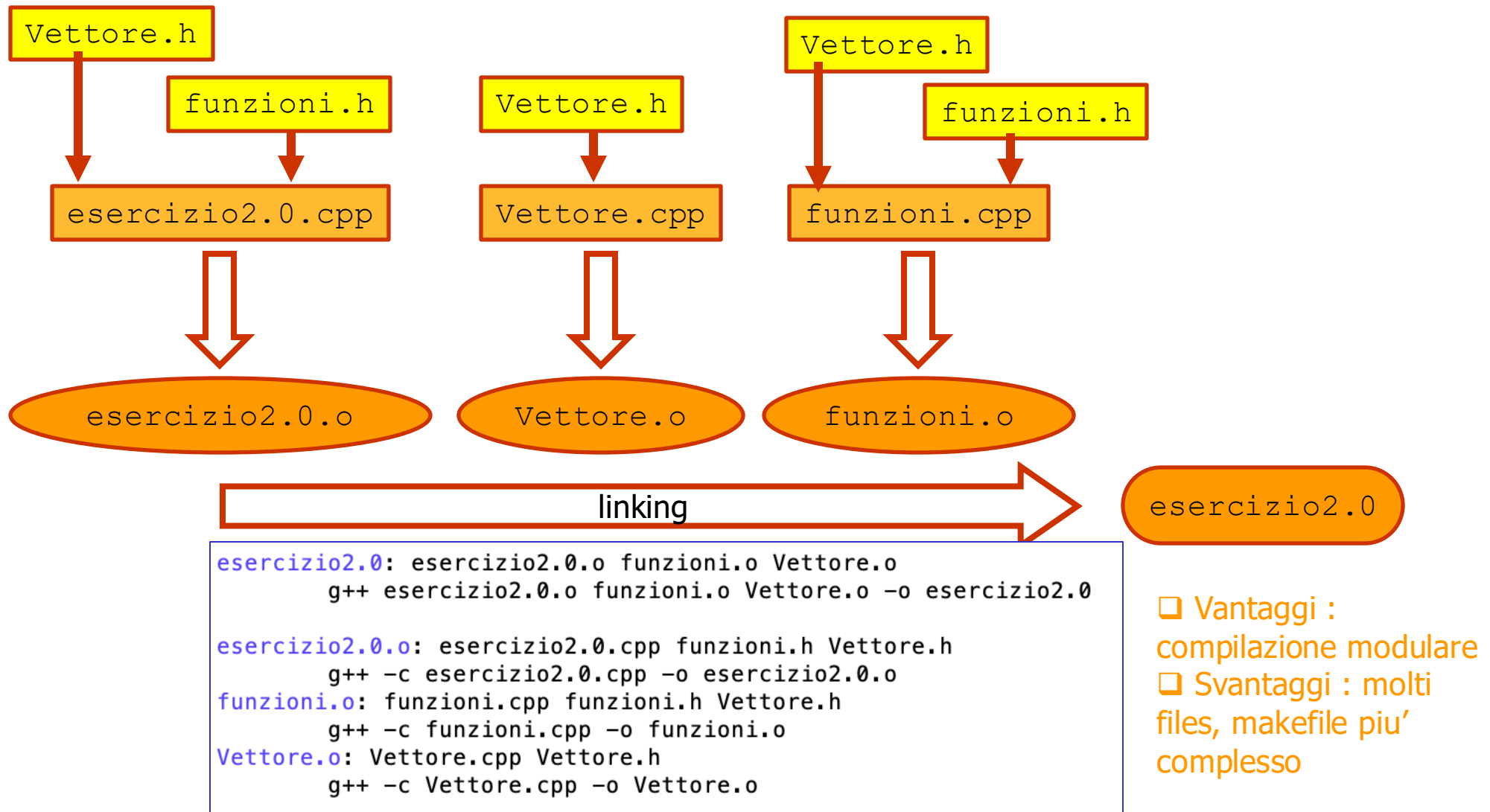
## Compilazione (I) : .cpp/.h separati, compilazione unica



```
esercizio2.0: esercizio2.0.cpp funzioni.cpp Vettore.cpp funzioni.h Vettore.h  
g++ esercizio2.0.cpp funzioni.cpp Vettore.cpp -o esercizio2.0
```

- ❑ Vantaggi : una sola istruzione di compilazione
- ❑ Svantaggi : una modifica ad una qualsiasi delle classi o funzioni causa una ricompilazione totale

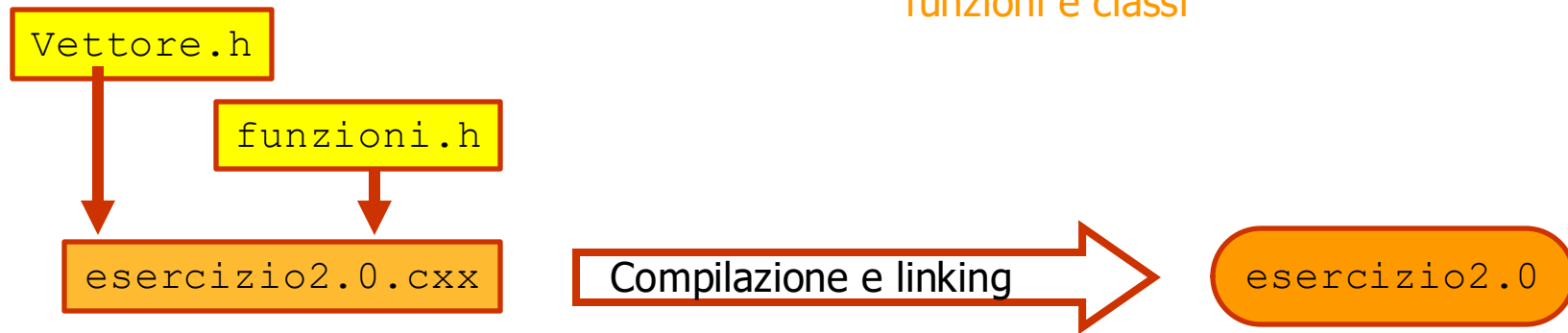
## Compilazione (II) : .cpp/.h separati, compilazione separata



- ❑ Vantaggi : compilazione modulare
- ❑ Svantaggi : molti files, makefile piu' complesso

## Compilazione (III) : solo .h, compilazione unica [classi template]

- ❑ Attraverso `#include` il main si incorpora contemporaneamente sia la dichiarazione sia l'implementazione di funzioni e classi



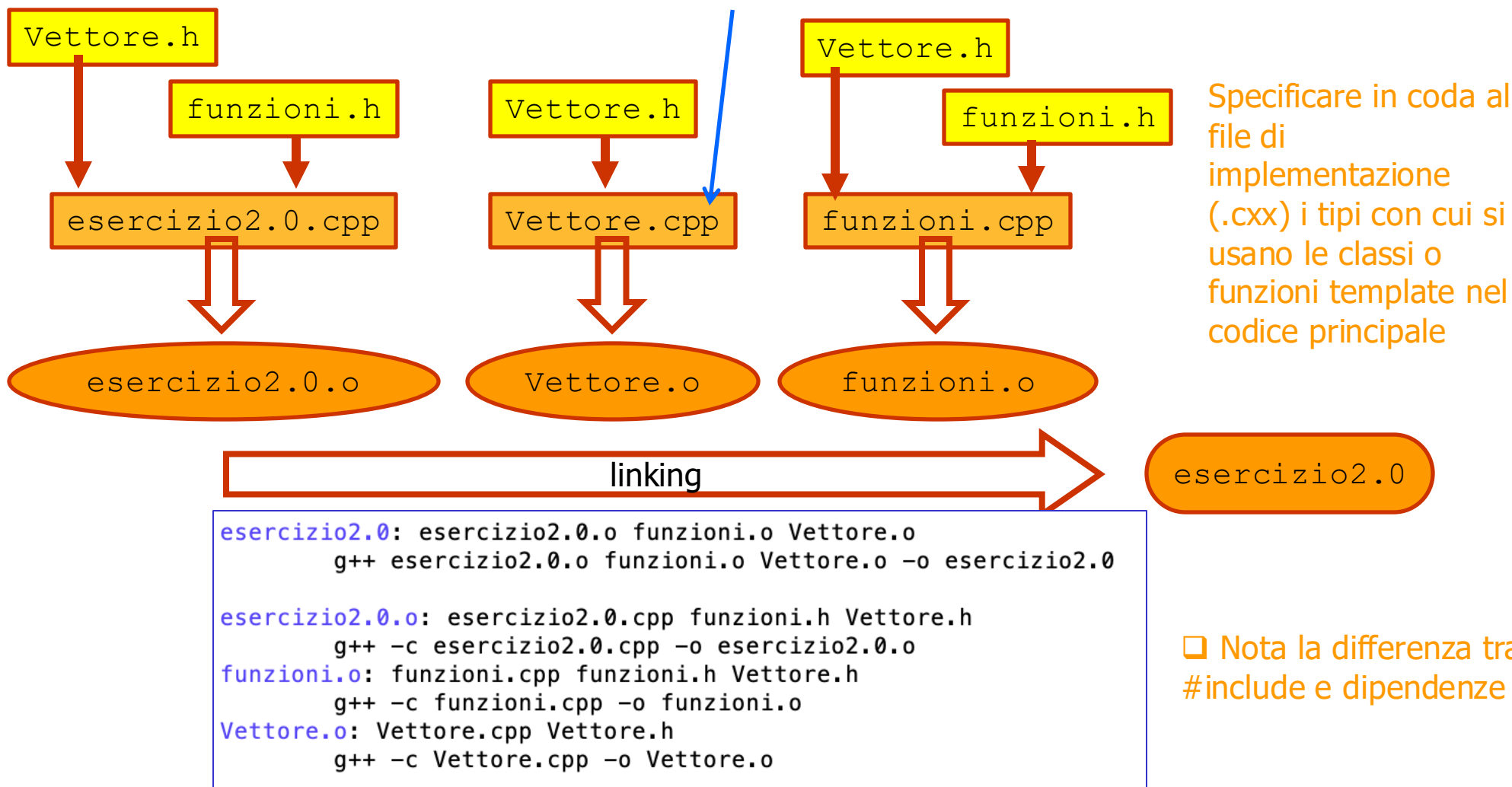
```
esercizio2.0: esercizio2.0.cpp funzioni.h Vettore.h  
g++ -o esercizio2.0 esercizio2.0.cpp
```

- ❑ Vantaggi : una sola istruzione di compilazione
- ❑ Svantaggi : una modifica ad una qualsiasi delle classi o funzioni causa una ricompilazione totale
- ❑ Unica possibilità per funzioni template



## Compilazione separata con classi template

```
template class Vettore<int>;  
template class Vettore<double>;
```



❑ Nota la differenza tra `#include` e dipendenze

## How to order a set of `Posizione` objects ?

**Option 2:** allows to define your own ordering criterium

```
template <class RandomAccessIterator, class Compare> void sort (RandomAccessIterator first,
                                                                RandomAccessIterator last,
                                                                Compare comp);
```

comp is a binary function that accepts two iterators and returns a value convertible to bool. We can use a function, functor or a lambda function ( see next slides )

<https://en.cppreference.com/w/cpp/algorithm/sort>

### Parameters

- first, last** - the range of elements to sort
  - policy** - the execution policy to use. See [execution policy](#) for details.
  - comp** - comparison function object (i.e. an object that satisfies the requirements of [Compare](#)) which returns `true` if the first argument is *less* than (i.e. is ordered *before*) the second.
- The signature of the comparison function should be equivalent to the following:

```
bool cmp(const Type1& a, const Type2& b);
```

## The STL in its full power: lambda functions

C++ 11 introduced lambda expression to allow us write an inline function which can be used for short snippets of code that are not going to be reuse and not worth naming. In its simplest form lambda expression can be defined as follows:

```
[ capture clause ] (parameters) -> return-type  
{  
    function code  
}
```

- ❑ Type : use auto ! Lambda functions are just syntactic sugar for anonymous functors. You never know the type of a lambda and you can't type it. Each lambda has it's own unique type
- ❑ Generally, return-type in lambda expression are evaluated by compiler itself and we don't need to specify that explicitly and -> return-type part can be ignored but in some complex case as in conditional statement, compiler can't make out the return type and we need to specify that.
- ❑ Syntax used for capturing variables :
  - ❑ [ ] can access only those variable which are local to it
  - ❑ [&] : capture all external variable by reference
  - ❑ [=] : capture all external variable by value
  - ❑ [a, &b] : capture a by value and b by reference

---

## Commenti sparsi :

1. Anche se non li dichiarate esplicitamente Il C++ mette a disposizione delle vostre classi un costruttore senza argomenti, un copy constructor, un copy assignment operator e un distruttore ( sotto certe condizioni <https://cplusplus.com/doc/tutorial/classes2/> )
  1. Se implementate anche un solo costruttore quello di default viene cancellato
  2. Se implementate un distruttore quello di default viene cancellato e verra' utilizzato il vostro
2. Non e' sempre necessario fare overloading del distruttore ( anzi raramente ): se non dobbiamo effettuare operazioni particolare possiamo sempre affidarci al distruttore di default.
  1. Nella classe `Vettore` ad esempio era fondamentale per de-allocare correttamente la memoria allocate dinamicamente. Per un oggetto di tipo `Posizione` possiamo fidarci del distruttore di default
  2. Se non ne avete bisogno potete non dichiarare affatto il distruttore. Se lo dichiarate dovete poi implementarlo !
3. Allo stesso modo non e' sempre necessario fare overloading del costruttore di copia ( o dell'assignment operator )
  1. Nella classe `Vettore` ad esempio era fondamentale per costruire correttamente due vettori effettivamente distinti. Per un oggetto di tipo `Posizione` possiamo fidarci degli operatori di default