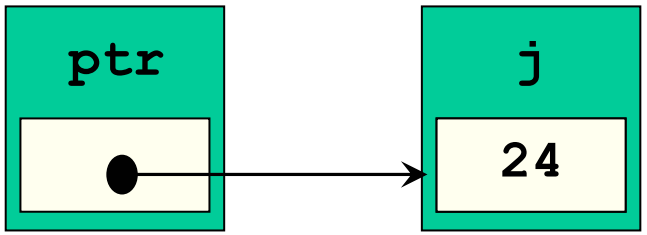
La programmazione *Object Oriented* e le sue applicazioni in Fisica.

Gabriella Cataldi, INFN Lecce

Daniele Martello, dip. Fisica e INFN Lecce

Puntatori

- Riferimento ad una locazione di memoria



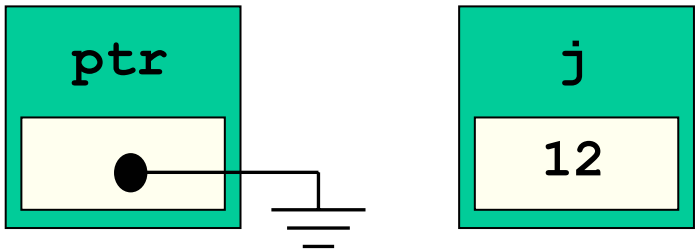

```
#include <iostream.h>

int main()
{
  int j = 12;
  int *ptr = &j;

  cout << *ptr << endl;
  j = 24;
  cout << *ptr << endl;
  cout << ptr << endl;

  return 0;
}
```

```
12
24
0x7b03a928
```

indirizzo di memoria

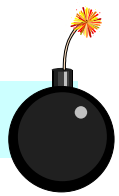
Puntatori

- Puntatore nullo




```
#include <iostream.h>

int main()
{
   int j = 12;
   int *ptr = 0;

   cout << *ptr << endl; // crash !
   return 0;
}
```

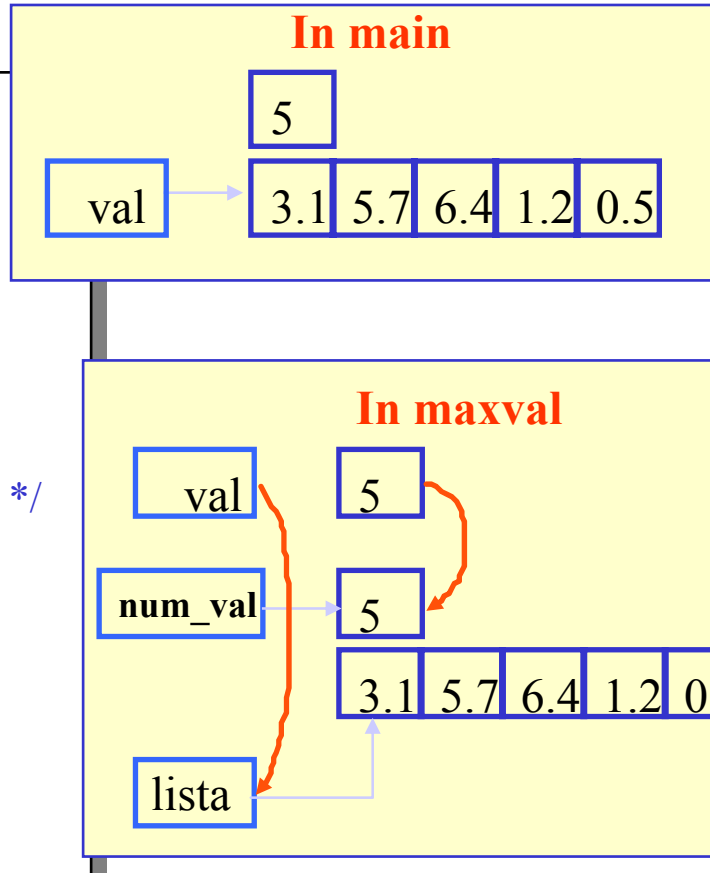
```
Segmentation violation (core dumped)
```



Puntatori e Array

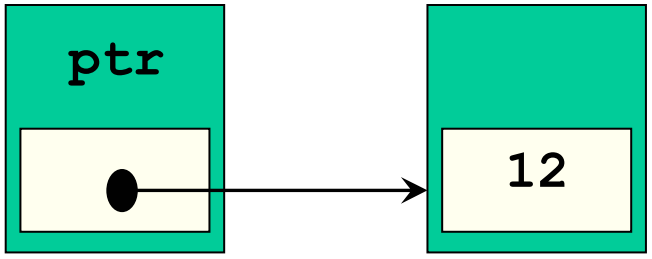
- In C gli *array* sono trattati come puntatori

```
#include <iostream>
/* Definisco la struttura della funzione*/
float maxval(int n, float *lista);
/* Corpo principale   */
void main () {
    float val[5]={ 3.1, 5.7, 6.4, 1.2, 0.5};
    cout<<"Il valore maggiore e'  "<< maxval(5,val))<<endl;
}
/* Cerca il max nel vettore val  costituito da num_val elementi */
float maxval (int num_val, float *lista) {
    int ind;
    float max=lista[0];
    for (ind=0; ind<num_val; ind++)
            if (lista[ind]>max) max=lista[ind];
    return max;
}
```



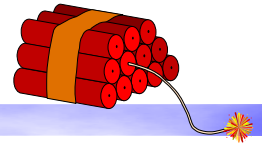
Puntatori: allocazione dinamica

- Riferimento ad una locazione di memoria




```
#include <iostream.h>

int main()
{
   int *ptr = new int;

   *ptr = 12;
   cout << *ptr << endl;

   delete ptr;
   return 0;
}
```

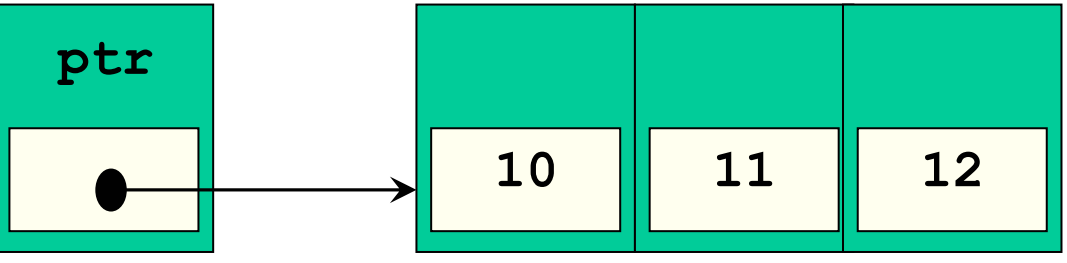
- Attenzione:
  - Non usare **`delete`** fa accumulare locazioni di memoria inutilizzate (*memory leak*)
  - Utilizzare puntatori prima del **`new`** o dopo il **`delete`** causa il *crash* del programma



Puntatori: allocazione dinamica

- Riferimento a piu' locazioni di memoria

```
#include <iostream>

int main()
{
  int *ptr = new int[3];

  ptr[0] = 10;

  ptr[1] = 11;

  ptr[2] = 12;

  delete [] ptr;
  return 0;
}
```



Lettura da file

```
#include <fstream>  //direttiva al preprocessore
#include <iostream>
using namespace std;
int Main()
{
  ifstream ifile("test.dat");
  int N;
  ifile >> N;
  cout<< "N= "<<N<<endl;
  double * x = new double[N];
  double * dx = new double[N];
  for (int i=0; i <N  ; i++ ){
    ifile >> x[i]>> dx[i];
    cout<<"misura "<<i+1<<" = "<<x[i]<<"+/-"<<dx[i]<<endl;
  }
  ifile.close();
}
```

Oggetto rappresentante il file in input

Programmazione procedurale

- Uno dei principali problemi del software è la sua evoluzione e la sua manutenzione
  - specialmente in grossi progetti o in progetti destinati ad evolversi nel tempo.

- Esempi con linguaggi procedurali (Fortran/C):
  - Cosa succede quando il codice viene modificato
  - Dipendenze all'interno del codice

Un esempio semplice (in Fortran ed in C)

- Un esempio apparentemente banale: copiare una sequenza di caratteri dalla tastiera alla stampante

```
      SUBROUTINE COPY
       EXTERNAL READKB
       LOGICAL  READKB
       EXTERNAL WRTPRN
       CHARACTER CH

       DO WHILE (READKB(CH))
         CALL WRTPRN(CH)
       ENDDO
      RETURN
```

```
      void copy() {

      char ch = 0;
      while((ch=readkb())!='x'){
          wrtprn(ch);
        }
        return;
      };
```

- Questo codice dovrà prima o poi essere modificato (aggiornamenti, estensioni, richieste dagli utenti, ecc.)

Modifiche dei requisiti

- Una prima modifica : scrivere anche su un file

```
      SUBROUTINE COPY(FLAG)
       EXTERNAL READKB
       LOGICAL  READKB
       EXTERNAL WRTPR,WRTFL
       CHARACTER CH
      INTEGER FLAG
       DO WHILE (READKB(CH))
         IF(FLAG .EQ. 1)THEN
          CALL WRTPRN(CH)
         ELSE
          CALL WRTFL(CH)
         ENDIF
       ENDDO
      RETURN
```

```
      void copy(int flag) {

       char ch=0;
         while ((ch=readkb())!='x') {
          if (flag==1) {
            wrtprn(ch);
          } else {
            wrtfl(ch);
          }
         }
         return;
      };
```

Evoluzione incontrollata

- Un'altra modifica per niente elegante: leggere anche da un file

```
      SUBROUTINE COPY(FLAG1, FLAG2)
      EXTERNAL READKB, READFL
      LOGICAL  READKB, READFL
      EXTERNAL WRTPR,WRTFL
      CHARACTER CH
      LOGICAL CONT
      INTEGER FLAG1, FLAG2

10    IF(FLAG1 .EQ. 1)THEN
        CONT = READKB(CH)
      ELSE
        CONT = READFL(CH)
      ENDIF
      IF (CONT) THEN
        IF(FLAG2 .EQ. 1)THEN
         CALL WRTPRN(CH)
        ELSE
         CALL WRTFL(CH)
        ENDIF
        GOTO 10
      ENDIF
      RETURN
```

```
void copy(int flag1, int flag2) {
    char ch = 0;
    while (ch != 'x') {
       if (flag1==1) {
         ch = readkb();
       } else {
         ch = readfl();
       }
       if (ch != 'x') {
         if (flag2 == 1) {
            wrtprn(ch);
         } else {
           wrtfl(ch);
         }
       }
    }
    return;
};
```

Descrizione dei dati

- Esempio: triangoli rettangoli

```
...
float c1;
float c2;
c1 = 1.;
c2 = 3.;
ipo = sqrt(c1*c1+c2*c2);
...
```

```
float ipotenusa(float cateto1, float cateto2) {
 return sqrt(cateto1*cateto1+cateto2*cateto2);
}
```

```
...
float c1;
float c2;
c1 = 1.;
c2 = 3.;
ipo = ipotenusa(c1, c2);
...
```

Evoluzione del codice

- Se cambia il formato dei dati?

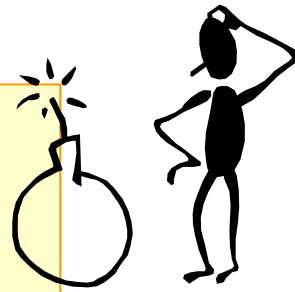
```
float → double
```

- Bisogna cambiare la funzione (gli argomenti sono diversi)

```
double ipotenusa(double cateto1, double cateto2)
```

- …e il codice!

```
...
double c1;
double c2;
c1 = 1.;
c2 = 3.;
ipo = ipotenusa(c1, c2);
...
```



Il concetto di dipendenza

- Nell'esempio precedente il codice di analisi ("*alto livello*") dipende dai dettagli della struttura dati ("*basso livello*").

```
double ipotenusa(double cateto1, double cateto2) {
   ...
}
```

IPOTENUSA dipende dalla struttura dei dati **CATETO1** e **CATETO2**

```
double c1;
double c2;
...
ipo = ipotenusa(c1, c2);
```

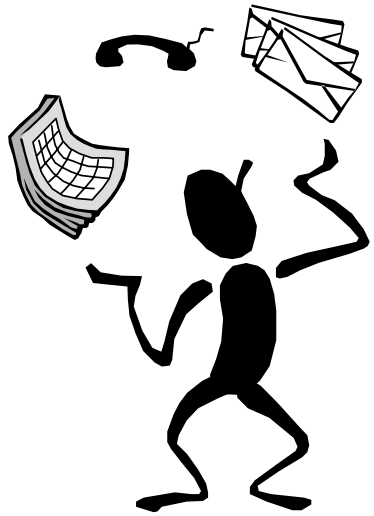
Il codice applicativo dipende dalla struttura dettagliata delle funzioni

C++/Object Oriented

- Riduce la dipendenza del codice di alto livello dalla rappresentazione dei dati
  Permette il riutilizzo del codice di alto livello
- Nasconde i dettagli di implementazione

Classi e oggetti

- Definizione di nuovi tipi (oltre a **`int`**, **`float`**, **`double`**) come:
  - numeri complessi,
  - vettori,
  - matrici, . . .
- ma anche:
  - tracce,
  - superfici,
  - elementi di rivelatori,
  - cluster, . . .
- Gli oggetti permettono di modellare una problema che rappresenti la realtà

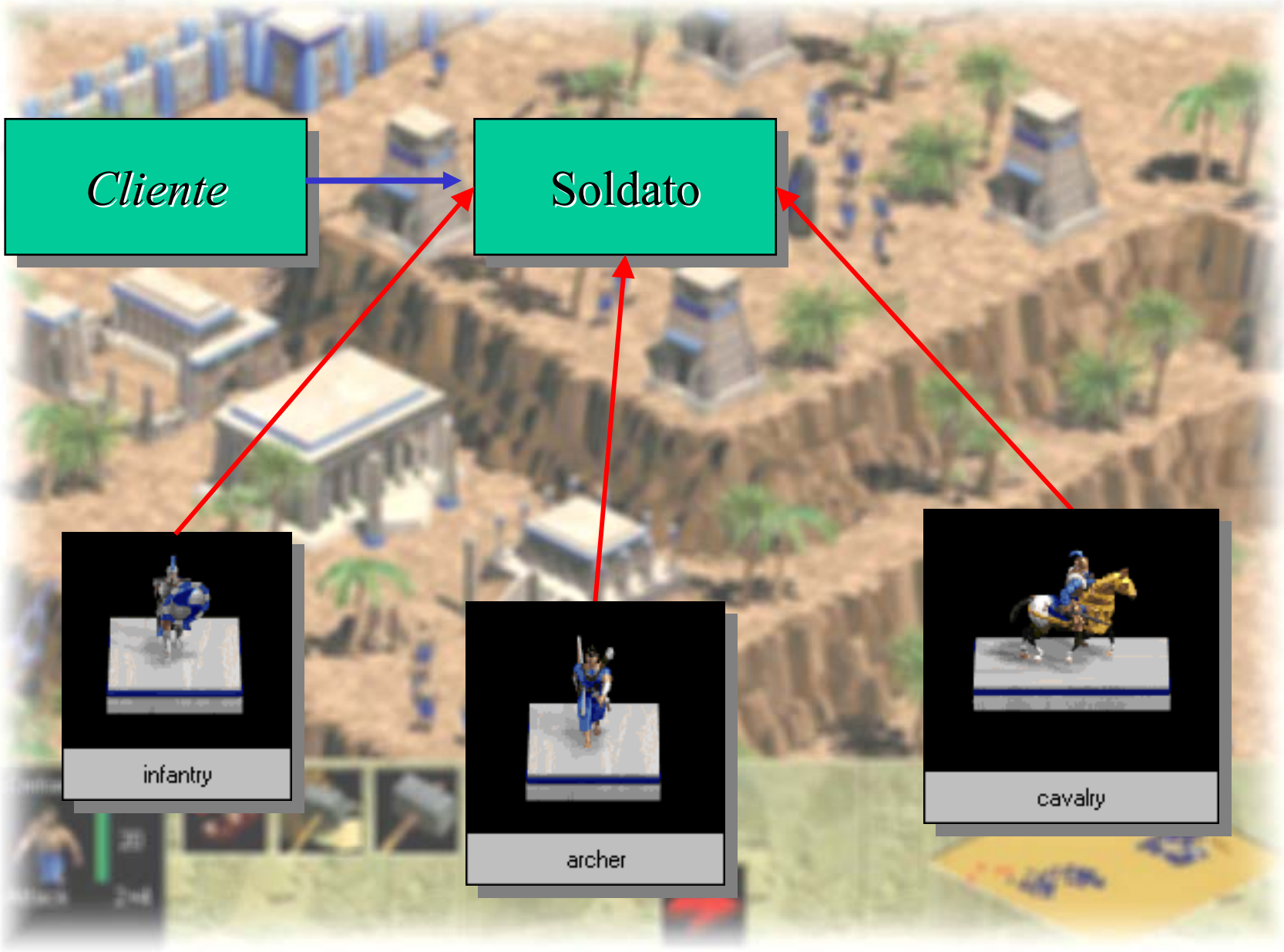


Concetti base dell'OO

- Classi ed oggetti
- Incapsulamento
- Relazione di ereditarietà
- Polimorfismo
- Programmazione Generica (**C++**)

Classi e Oggetti

- Un esempio di programma “orientato ad oggetti”: un videogioco


Cliente
Soldato
infantry
archer
cavalry

Classe `Vector3d`

- Un esempio: un vettore tridimensionale
- Una classe definisce oggetti che contengono insieme funzioni e dati

Vector3d.h

```
class Vector3d
{
public:
  Vector3d(double x, double y, double z);
  Vector3d(Vector3d& v);
  double x();
  double y();
  double z();
  double r();
  double phi();
  double theta();
private:
  double _x, _y, _z;
};
```

costruttore

funzioni o metodi

dati o attributi

Implementazione della classe

Vector3d.cc

```
#include "Vector3d.h"
Vector3d::Vector3d(double x, double y, double z) :
  _x(x), _y(y), _z(z) {}

Vector3d::Vector3d(Vector3d& v) :
  _x(v.x()), _y(v.y()), _z(v.z()) {}

double Vector3d::x()
{ return _x; }

double Vector3d::r()
{ return sqrt(_x * _x + _y * _y + _z * _z); }
```

costruttore copia

Interfaccia e implementazione

- Gli attributi privati non sono accessibili al di fuori della classe.
- I metodi pubblici sono gli unici visibili e costituiscono l'interfaccia di programmazione della classe.

Vector3d.h

```
class Vector3d
{
public:
  Vector3d(double x,double y,double z);
  double x();
  double y();
  double z();
  double r();
  double phi();
  double theta();
private:
  double _x, _y, _z;
};
```

Vector3d.cc

```
#include "Vector3d.h"

Vector3d::Vector3d(double x,
double y, double z) :
  _x(x), _y(y), _z(z)
{}

double Vector3d::x()
{ return _x; }
```

Classe `Vector3d`

- Come usare Vector3d:

main.cc

```
#include <iostream.h>
#include "Vector3d.h"

int main()
{
  Vector3d v(1, 1, 0);

  cout << " v = ("
       << v.x() << ","
       << v.y() << ","
       << v.z() << ")" << endl;
  cout << " r = " << v.r();
  cout << " theta = " << v.theta() << endl;
}
```

invoca il costruttore

Output:

```
v = (1, 1, 0)
r = 1.4141 theta = 1.5708
```

Interfaccia e implementazione

- Le strutture interne dei dati (**`_x`**, **`_y`**, **`_z`**) che reppresentano l'oggetto della classe **`Vector3d`** sono *nascoste* (**`private`**) ai *clients* della classe.
- I *clients* **`non dipendono`** dalla struttura dei dati (come lo erano i *clients* dei *common block* in FORTRAN)
- Se la struttura interna cambia (es.: **`_r, _theta, _phi`**), il codice che usa **`Vector3d`** non deve essere modificato.

Classe `Vector3d`

- Protezione dall'accesso ai dati:

main.cc

```
#include <iostream.h>
#include "Vector3d.h"

int main()
{
  Vector3d v(1, 1, 0);

  cout << " V = ("
       << v._x << ","              //
       << v._y << ","              // non
       << v._z << ")" << endl; // compila !
  cout << " r = " << v.r();
  cout << " theta = " << v.theta() << endl;
}
```

Selettori e modificatori

- I selettori const non modificano lo stato (= gli attributi) dell'oggetto
- I modificatori possono modificare lo stato dell'oggetto

Vector3d.cc

```
#include "Vector3d.h"

Vector3d::Vector3d(double x,
double y, double z) :
  _x(x), _y(y), _z(z)
{}

double Vector3d::x() const
{ return _x; }

void Vector3d::scale(double s)
{ _x *= s; _y *= s; _z *= s; }
```

Vector3d.h

```
class Vector3d
{
public:
  Vector3d(double x,double y,double z);
  double x() const;
  double y() const;
  double z() const;
  double r() const;
  double phi() const;
  void scale(double s);
private:
  double _x, _y, _z;
};
```

Selettori (const)

modificatore

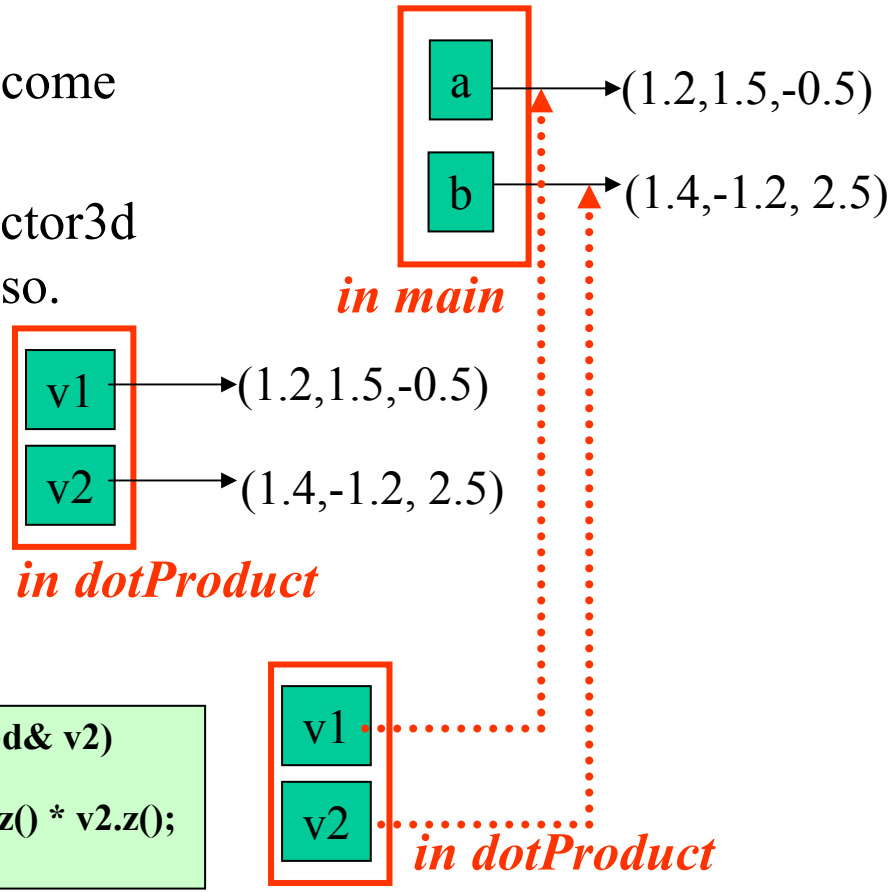
main.cc

```
int main ()
{
const Vector3d v(1,1,0);
double r = v.r(); //OK!
v.scale(1.1); // errore!!!
}
```

Argomenti delle funzioni

- E' possibile definire funzioni che utilizzano oggetti come argomenti.
- Un argomento puo' essere passato come valore o come riferimento
- Passare due oggetti della classe Vector3d come valore (= copia) e' dispendioso.

```
double dotProduct (Vector3d v1, Vector3d v2)
{
 return v1.x() * v2.x() + v1.y() * v2.y() + v1.z() * v2.z();
}
```



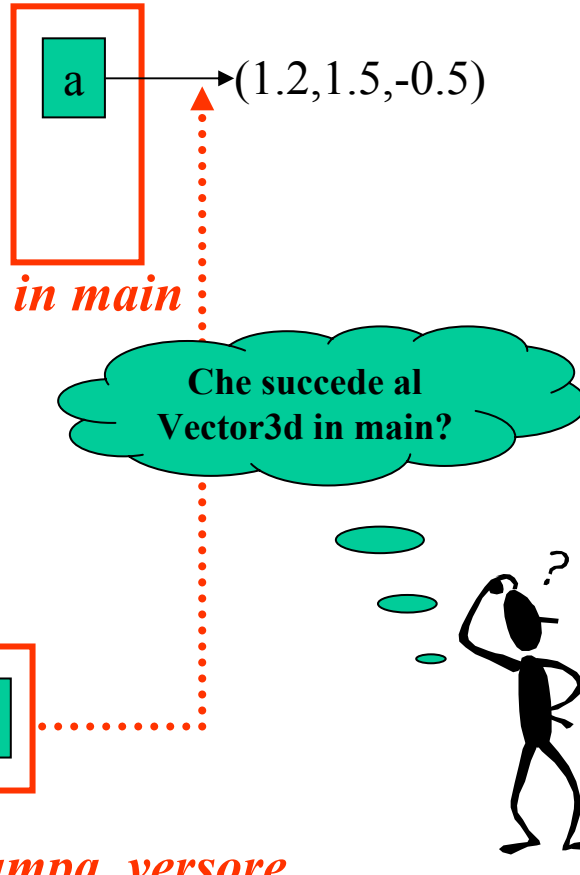
•Passare solo un riferimento (~puntatore) agli oggetti e' piu' efficiente.

```
double dotProduct (Vector3d& v1, Vector3d& v2)
{
 return v1.x() * v2.x() + v1.y() * v2.y() + v1.z() * v2.z();
}
```

Argomenti delle funzioni

Voglio implementare un metodo che stampa il versore di un vettore!

```
void stampa_versore(Vector3d&  v)
{
 v.scale ( 1 / v.r() ) ;
cout << " versore = ("
        << v.x() << ","
        << v.y()<< ","
        << v.z() << ")" << endl;
}
```



- Per evitare di modificare un oggetto passato come riferimento si puo' dichiararlo const

```
int main()
{
   const Vector3d a(1.2,1.5,-0.5), w(1,3,4);
  double aw = dotProduct(a,w);
  stampa_versore (a); //errore non compila
  return 0;
}
```

Esercizio1

```
#ifndef errorpoint_h
#define errorpoint_h
#include <cmath>
using namespace std;
class ErrorPoint {
private:
  double _value;
  double _error;
public:
// Default constructor
  ErrorPoint();
// a specialized  constructor
  ErrorPoint(double x,double dx=0.);
// Copy constructor
  ErrorPoint( const ErrorPoint & E);
// modifiers
  void value(double x);
  void error(double dx);
// selectors
  double value() const;
  double error() const;
};
#endif
```

ErrorPoint.h (presente)
ErrorPoint.cxx (da completare.

Esercizio1.cxx
Utilizza ErrorPoint ….

Esercizio2 & 3

```
..
/ una array di ErrorPoint
ErrorPoint dati[3];
...
 double x,dx;
..
    dati[k].value(x);
    dati[k].error(dx);
..
```

Esercizio2.cxx
Utilizzare una array di ErrorPoint e riempirla leggendo da tastiera

Esercizio3.cxx
Dimensionare dinamicamente una array di ErrorPoint e riempirla leggendo da tastiera

```
...
// una puntatore a  ErrorPoint
 ErrorPoint * dati;
....
  int N;
...
  cin>>N;
...
  dati = new ErrorPoint[N];
...
```