C++

Luca Lista

INFN, Sezione di Napoli

Sintassi base del C++

In C/ C++ non è necessario un particolare formato il codice

```
int main() {
  // esempio di programma

  ...

  return 0; // fine
}
```

Commenti con // o /* …*/

Istruzioni separate da “;”

Il C/ C++ è case sensitive

```
int i;
long j;
float x;
double d;
```

Tipi predefiniti in C++

	16 bit	32 bit	64 bit
char[1]	8	8	8
int[1]	16	32	32
bool	8	8	8
short[1]	16	16	16
long[1]	32	32	64
float	32	32	32
double	64	64	64
long double	64	128	128

[1] Può essere **unsigned**

Sintassi C++

- Controllo di flusso del programma

```
for (i = 1; i <= 10; i++)
{
  . . .
}
if (i == 10 && j > 4 || x)
{
   . . .
}
while( x != 5 )
{
   . . .
}
do
{
   . . .
} while( x != 5 )
```

Hello, world!

- Esempio di programma semplice con I/O

```
#include <iostream>

int main()
{
  cout << "Hello, world !" << endl;

  return 0;
}
```

direttiva al preprocessore

end of line

- Come compilare:

```
c++ program.cc -o prova
```

Funzioni matematiche

- In **C++** non esistono funzioni predefinite

```
#include <iostream>
#include <cmath>

int main()
{
  double r, theta, phi;

  cin >> r >> theta >> phi ;

  double x = r * sin( theta ) * sin( phi );
  double y = r * sin( theta ) * cos( phi );
  double z = r * cos( theta );

  cout << x << ", " << y << ", "
       << z << endl;

  return 0;
}
```

`cmath.h` definisce `sin`, `cos`, …

Niente `2**4`. Usare `pow(2, 4)`

Variabili locali e *scope*

- Le dichiarazioni possono essere fatte ovunque nel codice
- Le dichiarazioni di variabili sono valide solo entro un certo "*scope*"

```
#include <iostream>
const float pi = 3.1415396;

int main()
{
  int j = 0;
  if ( j == 0 )
  {
    int k = 5;
  }

  j = k; // errore: non compila !
  return 0;
}

void myRoutine( float x )
{
  int y = x * 2 * pi / 180;
  cout << "f(x) = "
       << y << endl;
}
```

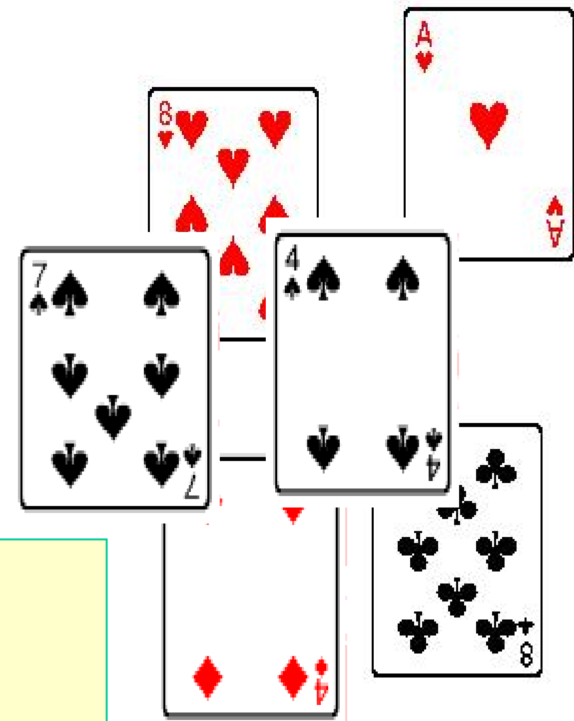
pi globale

j locale

Enumeratori

- In **C++** sono supportati tipi definiti dall'utente

```
enum Color
{
  red, green, blue
};

Color screenColor = blue;
Color windowColor = red;

int   n = blue; // valido
Color c = 1;    // errore
```



```
enum Seme
{
  cuori, picche, quadri, fiori
};
```

Array

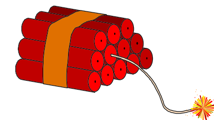
- In **C++** sono supportati gli array di dimensione fissa

```
int main()
{
  int x[10];

  for ( int i = 0; i < 10, i++ )
    x[i] = 0;

  double m[5][5];

  for ( int i = 0; i < 5; i++ )
    for ( int j = 0; j < 5; j++ )
      m[i][j] = i * j;

  return 0;
}
```

- Inizializzazione:

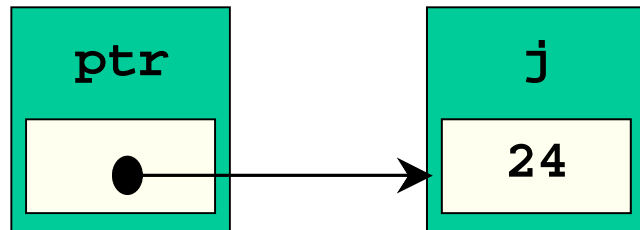
```
int x[] = { 1, 2, 3, 4 };

char t[] =
  { 'C', 'i', 'a', 'o', '\0' };

char s[] = "Ciao";

int m[2][2] =
  { {11, 12}, {21, 22} };
```

- L'indice va da **0** a **n-1**.
  Usare un indice maggiore di **n-1** può causare un *crash*.



Puntatori

- Riferimento ad una locazione di memoria



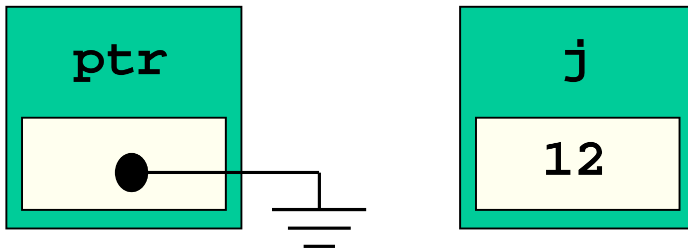

```
#include <iostream>

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

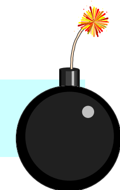
Puntatori

- Puntatore nullo



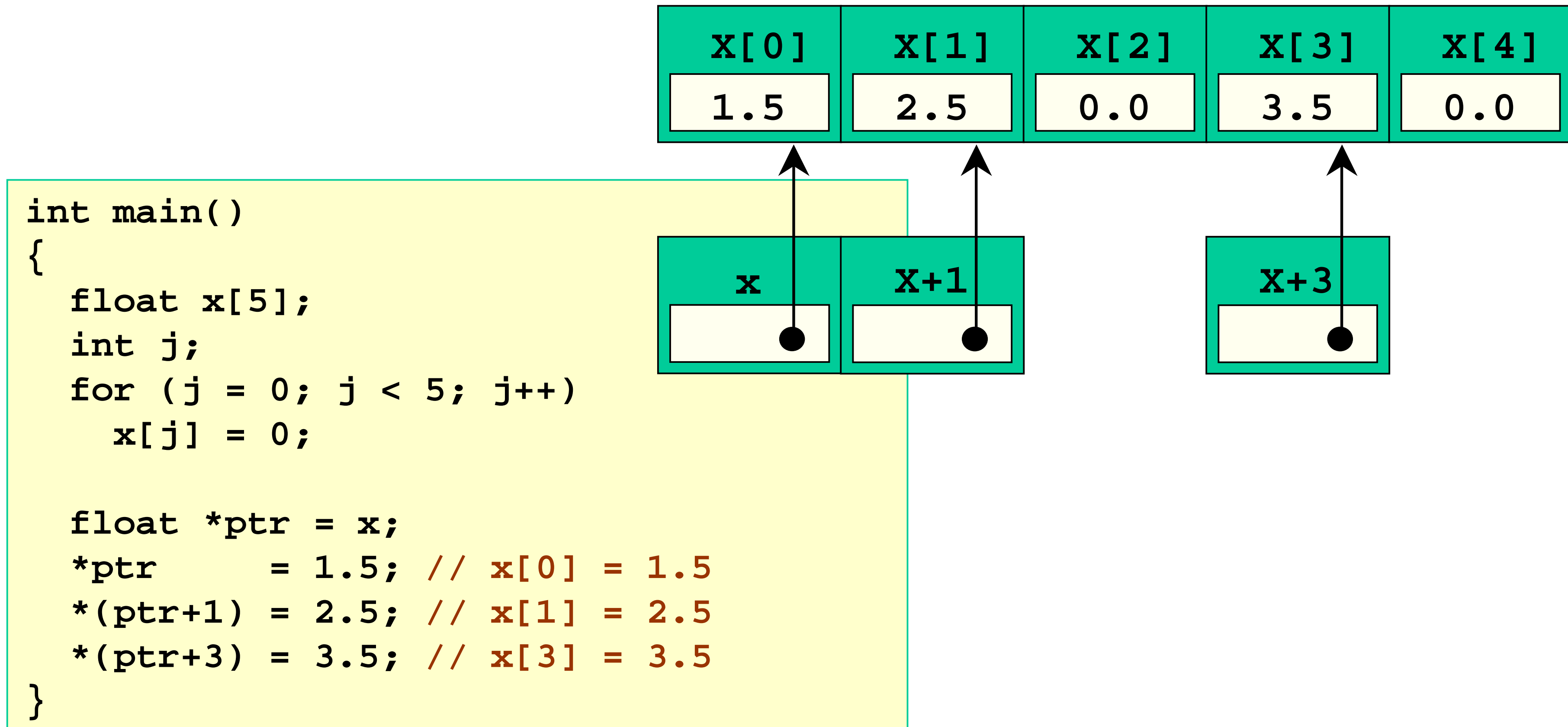

```
#include <iostream>

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

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

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



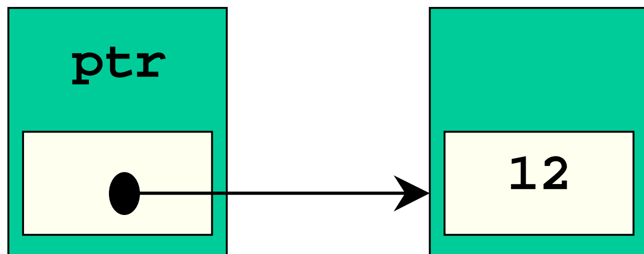
Puntatori e array

- In **C** gli array sono trattati come puntatori



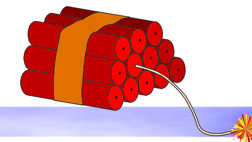
Puntatori: allocazione dinamica

- Riferimento ad una locazione di memoria




```
#include <iostream>

int main()
{
   int *ptr = new int;
   *ptr = 12;
   cout << *ptr << endl;

   delete ptr;
   return 0;
}
```

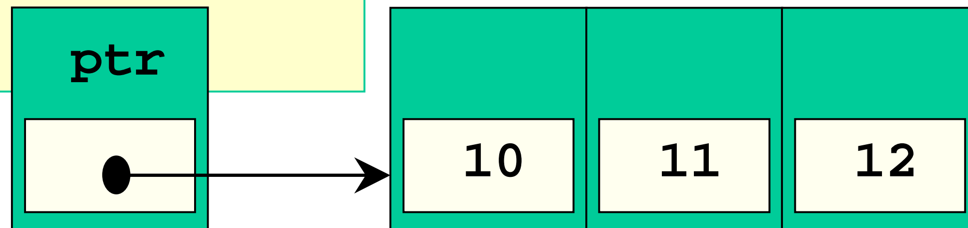
- Attenzione:
  - Non usare **`delete`** fa accumulare locazioni di memoria inutilizzate (*memory leak*)
  - Utilizzare puntatori prima del **`new`** o dopo il **`delete`** causa il *crash* del programma



Puntatori: allocazione dinamica

- Riferimento a più 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;
}
```



Classe Vector

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

Vector.h

```
class Vector
{
 public:
   Vector(double x, double y, double

   double x();
   double y();
   double z();
   double r();
   double phi();
   double theta();
 private:
   double x_, y_, z_;
};
```

Vector.cc

```
#include “Vector.h”

Vector::Vector(double x, double y, double z)
  : x_(x), y_(y), z_(z)
{ }

double Vector::x()
{
  return x_;
}

double Vector::r()
{
  return sqrt( x_*x_ + y_*y_ + z_*z_);
}
```

dati o attributi

Punto e virgola!

Interfaccia e implementazione

- Gli attributi privati non sono accessibili al di fuori della classe
- I metodi pubblici sono gli unici visibili

Vector.h

```
class Vector
{
public:
   Vector(double x, double y, double z);
   double x();
   double y();
   double z();
   double r();
   double phi();
   double theta();
private:
   double x_, y_, z_;
};
```

Vector.cc

```
#include "Vector.h"

Vector::Vector(double x,
               double y,
               double z) :
  x_(x), y_(y), z_(z)
{}

double Vector::x()
{
  return x_;
}

double Vector::r()
{
  return sqrt(x_*x_ + y_*y_ + z_*z_);
}
```

Classe Vector

- Come usare **Vector**:

main.cc

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

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

  cout << " v = ("
       << v.x() << ","
       << v.y() << ","
       << v.z() << ")" << endl;
  cout << " r = " << v.r();
  cout << " theta = " << v.theta() << endl;
  return 0;
}
```

Output:

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

Interfaccia e implementazione

- La struttura interna dei dati (**`x_`**, **`y_`**, **`z_`**) che rappresentano l'oggetto della classe **`Vector`** sono *nascosti* (**`private`**) ai client della classe.
- I client *non dipendono* dalla struttura interna dei dati (come lo erano i client dei common block Fortran)
- Se la struttura interna cambia (es.: **`r_`**, **`theta_`**, **`phi_`**), il codice che usa **`Vector`** non deve essere modificato.

Classe `Vector`

- Protezione dell'accesso ai dati:

main.cc

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

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

  cout << " V = ("
       << v.x_ << ","           //
       << v.y_ << ","           // non compila !
       << v.z_ << ")" << endl; //
  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

Vector.h

```
class Vector
{
public:
  Vector(double x, double y, double z);
  double x() const;
  double y() const;
  double z() const;
  double r() const;
  double phi() const;
  double theta() const;
  void scale(double s);
private:
  double x_, y_, z_;
};
```

Selettori (const)

modificatore

Vector.cc

```
#include "Vector.h"

void Vector::scale(double s)
{
  x_ *= s; y_ *= s; z_ *= s;
}
```

main.cc

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

Argomenti delle funzioni

- Un argomento può essere passato come valore o come riferimento
- Passare due oggetti della classe **Vector** come valore **(=copia)** è dispendioso

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

- Passare solo un riferimento **(= puntatore)** agli oggetti è più efficiente. Inoltre, è l'unico modo per poter modificare **v1** e **v2**.

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

Argomenti costanti

- Per evitare di modificare un argomento passato come riferimento, si può dichiararlo **const**

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

```
double normalize( Vector& v )
{
  v.scale( 1 / v.r() );
}
```

```
int main()
{
  const Vector v( 1, 1, 0 ), w( 1, 0, 1 );
  double vw = dotProduct( v, w );
  normalize( v ); // errore: non compila!
}
```

Operatori

- E' possibile ridefinire **+**, **-**, *****, **[ ]**, **++**, **==**, . . .

Vector.h

```
class Vector
{
public:
  Vector(double x, double y, double z);
  double x() const;
  double y() const;
  double z() const;
  double r() const;
  double phi() const;
  double theta() const;
private:
  double x_, y_, z_;
};

Vector operator+(const Vector& v1,
                 const Vector& v2);
Vector operator-(const Vector& v1,
                 const Vector& v2);
```

Vector.cc

```
Vector operator+(const Vector& v1,
                 const Vector& v2)
{
  return Vector(v1.x() + v2.x(),
                v1.y() + v2.y(),
                v1.z() + v2.z() );
}

Vector operator-(const Vector& v1,
                 const Vector& v2)
{
  return Vector(v1.x() - v2.x(),
                v1.y() - v2.y(),
                v1.z() - v2.z() );
}
```

Operatori

- Esempio:

main.cc

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

int main()
{
  Vector v1(1, 0, 0), v2(0, 1, 0);
  Vector v;

  v = v1 + v2;

  cout << " v = " << v << endl;
  cout << " r = " << v.r();
  cout << " theta = " << v.theta() << endl;
}
```

Sintassi alternativa *(!#@!?)* :

```
v.operator=( operator+(v1, v2) );
```

ridefinizione di **<<**

Output:

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

Classe **Array** a dimensione fissa

costruttore

distruttore

Array.h

```
class Array
{
public:
  Array( int size );
  Array(const Array & rh);
  ~Array();
   int size() const;
  int& operator[]( int i );
private:
  int* array_;
  int size_;
};
```

main.cc

```
int main()
{
  Array a(2);
  a[0] = 1;
  a[1] = 2;
}
```

Array.cc

```
#include <assert.h>
#include "Array.h"

Array::Array( int size ) :
  size_( size )
{
  array_ = new int[ size ];
}
Array::Array(const Array&rh) :size_(rh.size_)
{
  array_ = new int[size];
  // copiare il contenuto….
}
Array::~Array()
{
  delete [] array_;
}

int& Array::operator[]( int i )
{
  assert( i >=0 && i < size_ );
  return array_[i];
}

int Array::size() const { return size_; }
```

Operatori

- Esempio:

main.cc

```
#include <iostream>
#include <cmath>
#include "Vector.h"
#include "Matrix.h" // matrice 3x3

int main()
{
  Vector v1(1, 1, 0);

  double phi = M_PI/3;
  double c = cos(phi), s = sin(phi);

  Matrix m(1,  0, 0,
           0,  c, s,
           0, -s, c);

  Vector u = m * v;
}
```

Overloading di operatori

- possono esistere funzioni con lo stesso nome ma con argomenti diversi

Vector.hh

```
class Vector
{
public:
  // ...

private:
  double x_, y_, z_;
};

Vector operator*(const Vector &, double);
double operator*(const Vector&,
                 const Vector&);
```

Vector.cc

```
Vector operator*(const Vector&,
                 double s)
{
  return Vector( v.x() * s,
                 v.y() * s,
                 v.z() * s );
}

double
operator*(const Vector& v1,
          const Vector& v2)
{
  return ( v1.x() * v2.x() +
           v1.y() * v2.y() +
           v1.z() * v2.z() );
}
```

Operatori utilizzabili

2 argomenti

```
+     +=    <<=
-     -=    ==
*     *=    !=
/     /=    <=
%     %=    >=
^     ^=    &&
&     &=    ||
|     |=    ,
>     >>    ()
<     <<    []
=     >>=   ->*
```

1 argomento

```
+
-
*
&
->
~
!
++
--
```

```
Array v(3);
int n = v[0];

Complex c(2, 2);
Complex cConj = *c;

Vector v; Point p;
Vector u = v + p;

cout << v;
```

Argomenti di default

- E’ possibile specificare il default per gli argomenti delle funzioni

Vector.h

```
class Vector
{
public:
  Vector(double x = 0,
         double y = 0,
         double z = 0);
  . . .
};
```

Vector.cc

```
Vector::Vector(double x,
               double y,
               double z) :
  x_(x), y_(y), z_(z)
{ }

. . .
```

main.cc

```
#include <iostream>
#include “Vector.h”

int main()
{
  Vector v;
  cout << “v = “ << v << endl;
}
```

Output

```
v = (0, 0, 0)
```

Funzioni e dati statici

- Alcune funzioni e attributi possono essere comuni a tutta la classe

Book.h

```
class Book
{
public:
  Book(const string& title,
       const string& author);
  const string& title() const
    { return title_; }
  const string& author() const
    { return author_; }

  static int numberOfBooks();

private:
  string author_;
  string title_;
  static int n_;
};
```

Book.cc

```
#include "Book.h"

int Book::n_ = 0;

int Book::numberOfBook()
{
  return n_;
}

Book::Book(const string& title,
           const string& author) :
  title_(title), author_(author)
{
  n_ ++;
}
```

Ereditarietà

- Una classe derivata estende la classe base e ne eredita tutti i metodi e gli attributi

Book.h

```
class Book
{
public:
  Book(const string& title,
       const string& author);
  const string& title() const;
  const string& author() const;

protected:
  string title_, author_;
};
```

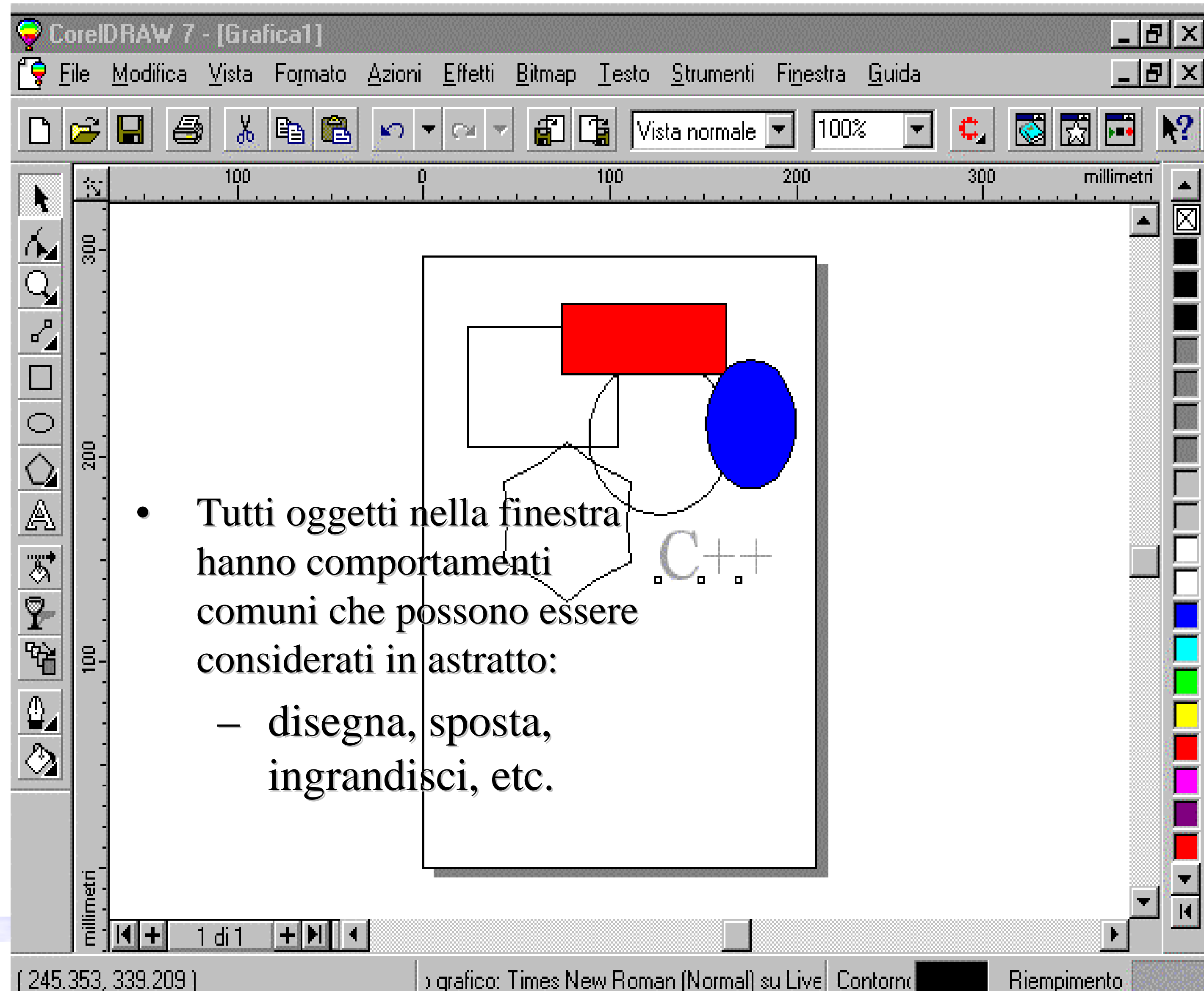
Volume.h

```
#include "Book.h"

class Volume : public Book
{
public:
  int volumeNumber() const
    { return number_; }

private:
  int number_;
};
```

- **Volume** è un **Book** che ha un attributo in più (**number_**) e un nuovo metodo (**volumeNumber()**)

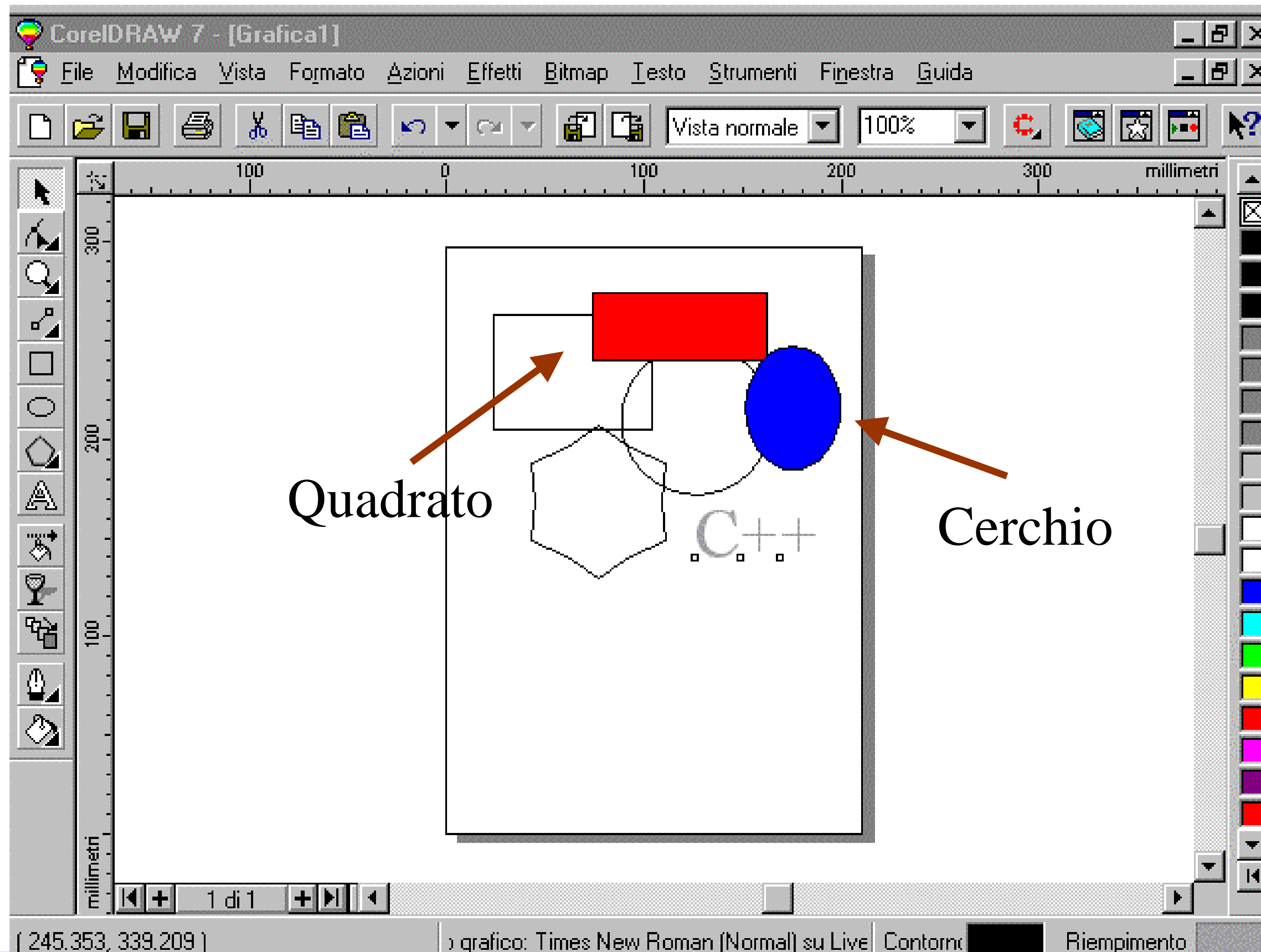
Classi astratte

- Esempio classico: `Shape`



- Tutti oggetti nella finestra hanno comportamenti comuni che possono essere considerati in astratto:
  - disegna, sposta, ingrandisci, etc.

Cerchi e quadrati



Cerchio

Circle.h

```
class Circle  {
public:
  Circle(Point2d center, double radius);
  ~Circle();

  void moveAt(const Point2d & p);
  void moveBy(const Vector2d & p);
  void scale(double s);
  void rotate(double phi);
  void draw() const;
  void cancel() const;
private:
  Point2d center_;
  double radius_;
};
```

Nome della classe

Costruttore

Distruttore

`Point2d`: classe che rappresenta un punto in 2 dimensioni.

Interfaccia Pubblica

Metodi: operazioni sugli oggetti

"Dati" privati (Attributi, membri)

Main.cc

```
#include "Circle.h"
int main()
{
  Circle c( Point2d(10, 10), 5 );

  c.draw();
  c.moveAt(Point2d(20, 30));

  return 0;
}
```

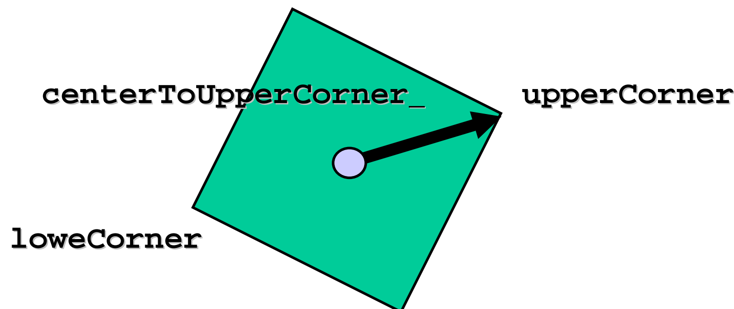
Circle.cc

```
#include "Circle.h"

void Circle::draw() const
{
  const int numberOfPoints = 100;
  float x[numberOfPoints], y[numberOfPoints];
  float phi = 0, deltaPhi = 2*M_PI/100;
  for ( int i = 0; i < numberOfPoints; ++i )
  {
     x[i] = center_.x() + radius_ * cos( phi );
     y[i] = center_.y() + radius_ * sin( phi );
     phi += dphi;
  }
  polyline_draw(x, y, numberOfPoints, color_, FILL);
}

void Circle::moveAt( const Point2d& p )
{
  cancel();  center_ = p;  draw();
}

void Circle::scale( double s )
{
  cancel();  radius_ *= s;  draw();
}

Circle::Circle( Point2d c, double r ) :
  center_( c ), radius_( r )
{  draw();  }

Circle::~Circle()
{  cancel();  }
```

Quadrato

Square.h

```
class Square  {
public:
  Square(const Point2d&, const Point2d&,
         Color color = TRASPARENT);
  ~Square();
  void moveAt( const Point2d& p );
  void moveBy( const Vector2d& p );
  void changeColor( Color color );
  void scale( double s );
  void rotate( double phi );
  void draw() const;
  void cancel() const;

private:
  Point2d center_;
  Vector2d centerToUpperCorner_;
  Color color_;
};
```



Square.cc

```
#include "Square.h"

void Square::draw() const
{
  float x[4], y[4];
  Vector2d delta( centerToUpperCorner_ );
  for ( int i = 0; i < 4; i++ )
  {
    Point2d corner = center_ + delta;
    x[i] = corner.x();
    y[i] = corner.y();
    delta.rotate( M_PI_2 );
  }
  polyline_draw(x, y, 4, color_, FILL);
}

void Square::rotate( double phi )
{
  cancel();
  centerToUpperCorner_.rotate( phi );
  draw();
}

Square::Square(const Point2d& lowerCorner,
               const Point2d& upperCorner,
               Color color) :
  center_( median(lowerCorner, upperCorner) ),
  centerToUpperCorner_( upperCorner - center_ ),
  color_( color )
{  draw();  }

void Square::scale( double s )
{ cancel();  centerToUpperCorner_ *= s;  draw(); }
```

Codice Applicativo (*Client*)

Main.cc

```
#include "Circle.h"
#include "Square.h"

int main()
{
  Circle c1( Point2d(2.,3.), 4.23 );
  Square r1( Point2d(2.,1.), Point2d(4.,3.) );

  Circle * circles[ 10 ];
  for ( int i = 0; i < 10; ++i )
  {
    circles[ i ] = new Circle( Point2d(i,i), 2. );
  }

  for ( int i = 0; i < 10; ++i )
    circles[ i ]->draw();

  return 0;
}
```

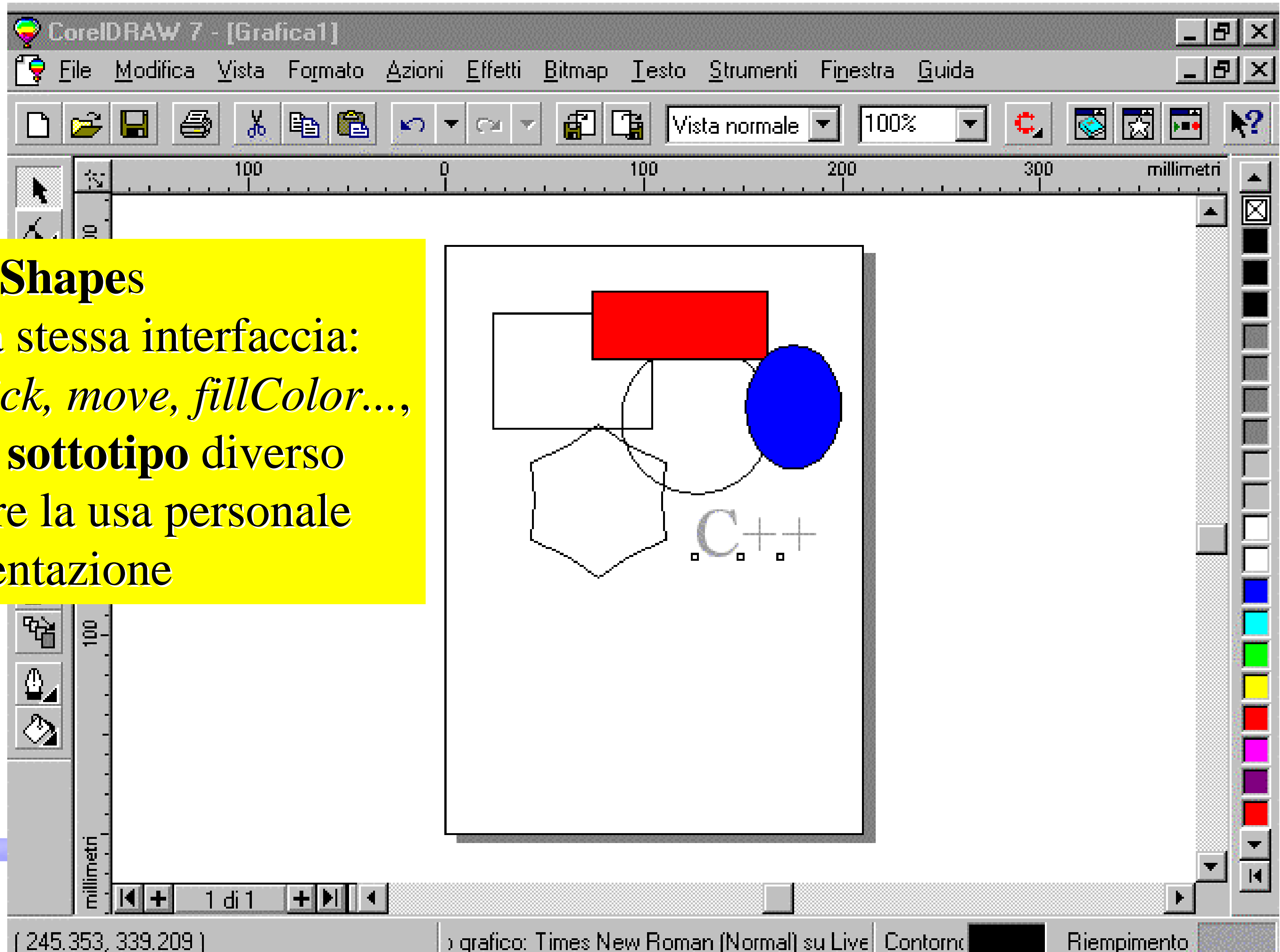
Costruisce un vettore di puntatori a cerchi, crea oggetti in memoria e salva i loro puntatori nel vettore.

Itera sul vettore e invoca `draw()` per ogni elemento

Come gestire cerchi e quadrati insieme?

Polimorfismo

Tutte le **Shapes**
hanno la stessa interfaccia:
draw, pick, move, fillColor...,
ma ogni **sottotipo** diverso
può avere la usa personale
implementazione



Interfaccia astratta

Shape.h

```
class Shape {
public:
  Shape() { }
  virtual ~Shape() { }

  virtual void moveAt(const Point2d& where) = 0
  virtual void changeColor(Color newColor) = 0;
  virtual void scale(double s) = 0;
  virtual void rotate(double phi) = 0;
  virtual void draw() const = 0;
  virtual void cancel() const = 0;
};
```

Square.h

```
#include "Shape.h"

class Square : public Shape
{
  // …. Il resto tutto uguale a prima
};
```

Main.cc

```
#include "Circle.h"
#include "Square.h"

int main()
{
  Shape * shapes[ 20 ];
  int index = 0;
  for ( int i = 0; i < 10; i++ )
  {
    Shape * s;
    s = new Circle( Point2d(i, i), 2.) );
    shapes[ index ++ ] = s;
    s = new Square( Point2d(i, i),
                    Point2d(i+1, i+2)) );
    shapes[ index ++ ] = s;
  }

  for ( int i = 0; i < 20; i++ )
    shapes[ i ]->draw();

  return 0;
}
```

Ereditarietà e riuso del codice

CenteredShape.h

```
Class CenteredShape: public Shape
{
public:
  CenteredShape(Point2d c,
    Color color = TRASPARENT)
  : center_(c), color_(color)
  { /*draw();*/ }
  ~Circle()
  { /*cancel();*/ }

  void moveAt( const Point2d& );
  void moveBy( const Vector2d& );
  void changeColor( Color );
  virtual void scale( double ) = 0;
  virtual void rotate( double ) = 0;
  virtual void draw() const = 0;
  virtual void cancel() const = 0;

protected:
  Point2d center_;
  Color color_;
};
```

Non si possono chiamare metodi virtuali in costruttori e distruttori (troppo presto, troppo tardi)

Square.h

```
#include "CenteredShape.hh"

class Square : public CenteredShape
{
public:
  Square( Point2d lowerCorner, Point2d upperCorner,
          Color col = TRASPARENT) :
    CenteredShape( median(lowerCorner, upperCorner), col),
    touc_(upperCorner - center_) { draw(); }
  ~Square() { cancel(); }

  virtual void scale( double s )
    { cancel(); centerToUpperCorner_ *= s; draw(); }
  virtual void rotate( double phi );
  virtual void draw() const;
  virtual void cancel() const;

private:
  Vector2d touc_;
};
```

Ereditarietà

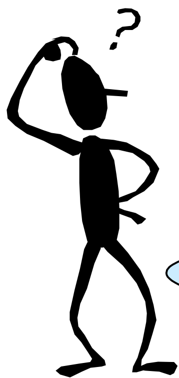
Rectangle.h

```
class Rectangle
{
public:
  Rectangle(double x0, double y0,
            double lx, double ly) :
    lx_(lx), ly_(ly), x0_(x0), y0_(y0) { }
  void scaleX(double s);
  void scaleY(double s);
protected:
  double x0_, y0_;
  double lx_, ly_;
};
```

- *Attenzione*: scegliere le relazioni di ereditarietà può essere non banale.
- Un quadrato è un rettangolo?

Square.h

```
class Square : public Rectangle
{
public:
  Square(double x0, double y0, double l) :
    Rectangle(x0, y0, l, l) { }
};
```



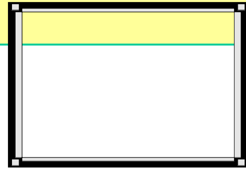
Avere **lx_** e **ly_** è ridondante per **Sqare**
Cosa succede se si invoca **scaleX** o **scaleY** ?

Ereditarietà multipla

- Una classe può ereditare da più classi

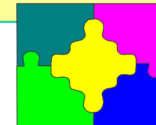
DrawableObj.h

```
class DrawableObj
{
public:
  virtual void draw() = 0;
};
```



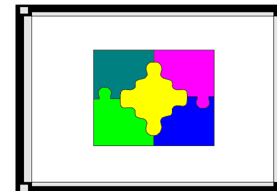
Shape.h

```
class Shape
{
public:
  virtual void scale(double s) = 0;
  virtual void moveAt( Vector2d& ) = 0;
};
```



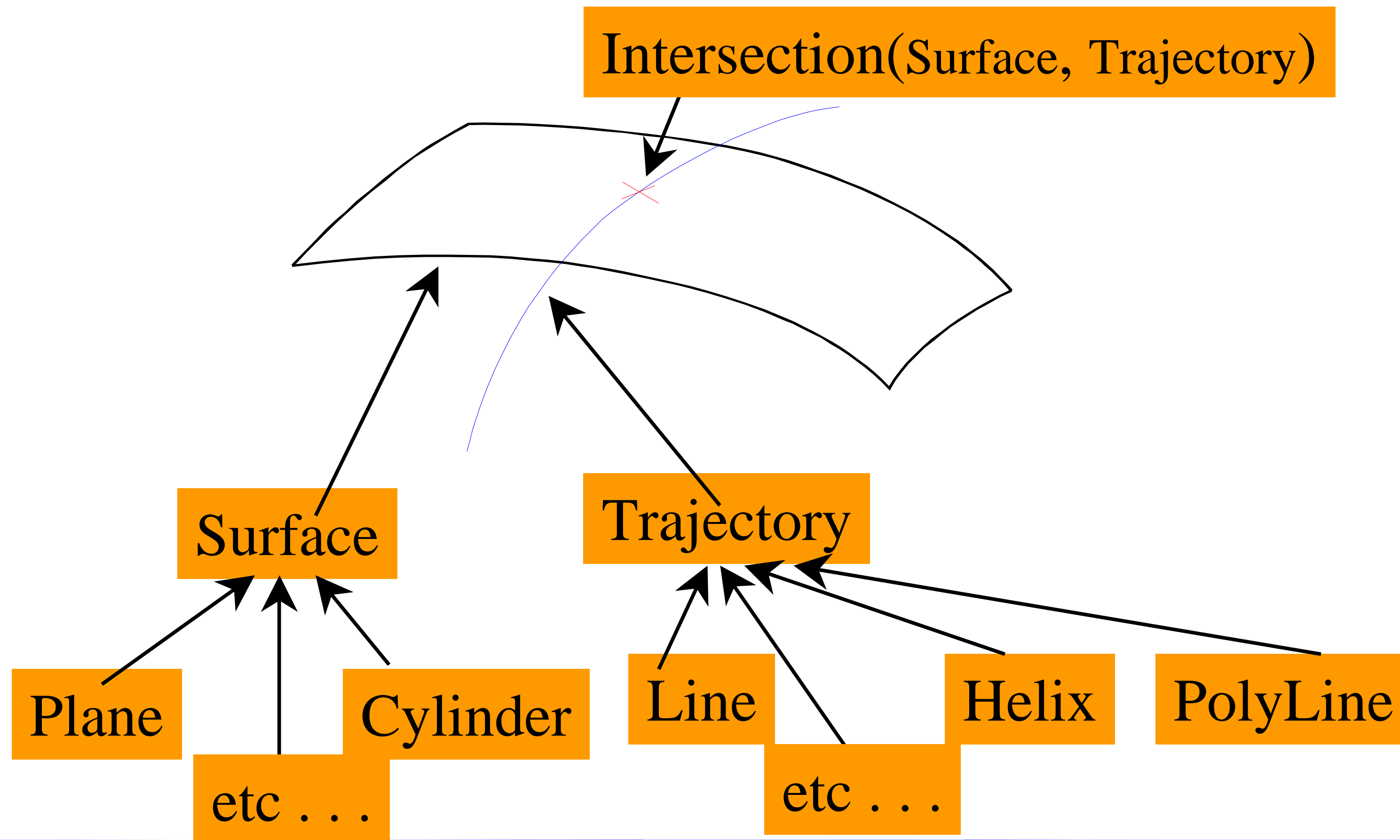
DrawableShape.h

```
class DrawableShape :
  public DrawableObj, public Shape
{
public:
  virtual void draw();
  virtual void scale(double s);
  virtual void moveAt( Vector2d& );
};
```



Superfici e traiettorie

- Nel tracking spesso è necessario calcolare intersezioni tra curve (tracce) e superfici (elementi di detector)



Superfici e traiettorie

- Interfaccia delle diverse **`Trajectory`**

Trajectory.h

```
class Trajectory
{
public:
 virtual Point position(double s) = 0;
 virtual Vector direction(double s) = 0;
};
```

Line.h

```
#include "Trajectory.h"

class Line : public Trajectory
{
public:
  virtual Point position(double s);
  virtual Vector direction(double s);
private:
  Point  origin_;
  Vector direction_;
};
```

Helix.h

```
#include "Trajectory.h"

class Helix : public Trajectory
{
public:
  virtual Point position(double s);
  virtual Vector direction(double s);
};
```

Superfici e traiettorie

- Implementazione

Trajectory.cc

```
#include "Trajectory.h"

// … vuoto ...
```

Line.cc

```
#include "Line.h"

Line::Line(const Point& o, constVector& d) :
  origin_( o ), direction_( d.unit() )
{
}

Point Line::position(double s)
{
  return ( origin_ + s * direction_ );
}
```

Helix.cc

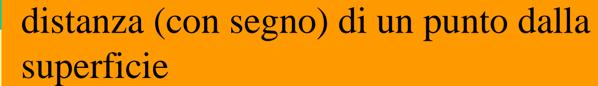
```
#include "Helix.h"

Helix::Helix()
{
}

Point Helix::position(double s)
{
 // implementazione
}
```

Superfici e traiettorie

- Interfaccia delle varie **Surface**

Surface.h

```
class Surface
{
public:
   virtual distance(const Point& p) = 0;
   virtual derDist(const Point& p, const Vector& r) = 0;
};
```



Plane.h

```
#include “Surface.h”

class Plane : public Surface
{
public:
   virtual distance(const Point& p);
   virtual derDist(const Point& p,
                   const Vector& r);
protected:
   Point origin_;
   Vector norm_;
   double dist_;
};
```

Cylinder.h

```
#include “Surface.h”

class Cylinder : public Surface
{
public:
   virtual distance(const Point& p);
   virtual derDist(const Point& p,
                   const Vector& r);
};
```

Superfici e traiettorie

- Surface è una classe astratta

Plane.cc

```
#include "Plane.h"

Plane::distance(const Point& p)
{
  return ( _dist - ( (p - origin_) * direction_) );
}

Plane::derDist(const Point& p,
               const Vector& r)
{
  return - r * _direction;
}
```

Surface.cc

```
#include "Surface.h"

 // vuoto
```

Cylinder.cc

```
#include "Cylinder.h"

Cylinder::distance(const Point& p)
{  /* . . . */  }

Cylinder::derDist(const Point& p,
                  const Vector& r)
{ /* . . .  */ }
```

Superfici e traiettorie

- Interfaccia di **`Intersection`**

Intersection.h

```
class Surface;
class Trajectory;

class Intersection
{
public:
  Intersection(Surface* s, Trajectory* t)
    surface_(s), trajectory_(t) {}
  Point intersect(double s1, double s2);

protected:
  double sIntersect(double s1, double s2);

  Surface* surface_;
  Trajectory* trajectory_;
};
```

Superfici e traiettorie

Intersection.cc

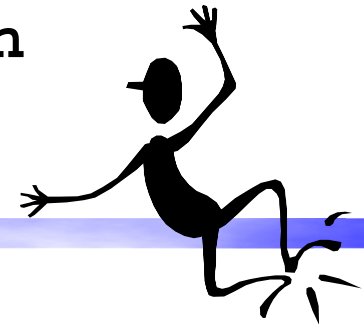
```
#include "Intersection.h"
#include <cmath>
#include "Surface.h"
#include "Trajectory.h"

const int maxIterations  20
const double sMax 1.e+6
const double accuracy1.e-3

double Intersection::sIntersect(double s1,
                                double s2)
{
  // algoritmo di Newton-Raphson
  double s = s1;
  double  maxS = max(s1, s2);
  double  minS = min(s1, s2);

  double d, delta;
  for( int j = 0; j < maxIterations; j++ )
  {
    Point p = _trajectory->position( s );
    d  = surface_->distance( p );
    delta = surface_->derDist( p,
              trajectory_->direction( s ) );
    double ds = - d / delta;
    double test =  s + ds;
```

- Implementazione dell'algoritmo

```
    // controlla che test è tra s1 e s2
    if( (s1 - test) * (test - s2) < 0.0 )
    {
      if ( s1 < s2 ) s += abs( d );
      else           s -= abs( d );
      if( s > maxS || s < minS )
        return sMax;
    }
    else s = test;

    if( abs(d) < accuracy )
      return s;
  }
  return sMax;
}

Point Intersection::intersect(double s1,
                              double s2)
{
  return
    trajectory_->position(sIntersect(s1, s2));
}
```

Superfici e traiettorie

- `Intersection` usa solo:
  - I metodi `position` e `direction` di un'oggetto `Trajectory`
  - I metodi `distance` e `derDist` di un oggetto `Surface`
- E' possibile aggiungere una nuova classe che modellizza una nuova `Trajectory` o una nuova `Surface` e `Intersection` continua a funzionare senza modificare una linea di codice!
- E' possibile rendere anche `Intersection` astratto...



Ritornando al primo esempio ...

- **`READKBD`** e **`READFL`** possono essere due implementazioni dello stesso metodo virtuale **`read`** di una classe **`InputDevice`**
- **`WRITEKBD`** e **`WRITEFL`** possono essere implementazioni dello stesso metodo virtuale **`write`** di una classe **`OutputDevice`**

```
void copy(InputDevice& in, OutputDevice& out)
{
  char c;
  while ( in.read(c) )
    out.write(c);
}
```