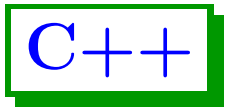

programmazione OO

L. Bellucci
Università di Firenze

23-24 Maggio

- Vantaggi della programmazione OO
- Programmare interfacce
- Ereditarietà e polimorfismo
- Entrare nell'azione

Procedurale vs Object Oriented

L'utilizzo di linguaggi *procedurali* comporta vari problemi:

- Evoluzione incontrollata del codice
- Dipendenza del codice dalle strutture dati

I linguaggi *object oriented* nascono con l'intento di superare queste limitazioni e rendere più semplice la manutenzione del software. In particolare:

- Riduzione della dipendenza del codice di alto livello da quello di basso livello
- Riutilizzo del codice di alto livello
- Dettagli delle implementazioni nascosto
- Supporto di tipi di dati astratti

Piccole note di sintassi

Tipi predefiniti

`bool`; `char`; `int`; `float`; `double`

Input Output

È necessario includere il file `iostream.h`

`cin` e `cout`

Esempio

```
#include <iostream.h>
int  main() {
  bool  b=true;
  char  c='a';
  int  i=1;
  float  f;
  cin>>f;
  cout<<b<<" "<<c<<" "<<i<<" "<<f<<endl;
}
```

Piccole note di sintassi

Array

```
float x[100];
```

Puntatori e reference

```
float* p; float& r;
```

Esempio

```
#include <iostream.h>
int  main() {
  float x[3]={1.1,2.2,3.3};
  cout<<x[0]<<x[1]<<x[2]<<endl;
  float  f=1.;
  float* p;
  p=&f;
  float& r;
  r=f;
  cout<<f<<p<<r<<endl;
}
```

Piccole note di sintassi

Controllo di flusso

`if; if...else; do...while; while;`

Allocazione dinamica della memoria

`new delete`

Esempio

```
#include <iostream.h>
int  main() {
  float* array;
  int n=0;   cin>>n;
  array=new float[n];
  for (int i=0; i<n; i++) {
    cin>>array[i];
  }
    .
    .
    .
  delete[] array;
}
```

Un esempio di classe

Vediamo come si dichiara e definisce una classe in *C++*.
Come linea generale diciamo che la dichiarazione avviene in un *header* file, e l'implementazione dei metodi in un file *libreria*.
Per convezione usiamo .h come estensione per gli *header* e .cc per le *librerie*.

Alcuni metodi sono obbligatori: tutte le classi devono possedere un costruttore ed un distruttore (generalmente `public`).

Vettore2D.h

```
#ifndef VETTORE2DH
#define VETTORE2DH
class Vettore2D {
  public:
    Vettore2D() {}
    Vettore2D(const float& x, const float& y);
    Vettore2D(const Vettore2D& punto);
    ~Vettore2D() {}
    float x() const;
    float y() const;
    float mod() const;
  private:
    float theX,theY;
};
#endif
```

Come vengono poi implementati i metodi dichiarati nell'*header* file?

Vettore2D.cc

```
#include <math.h>
#include "Vettore2D.h"
Vettore2D::Vettore2D(const float& x, const float& y):
                     theX(x), theY(y) {}
Vettore2D::Vettore2D(const Vettore2D& punto) {
  theX=Vettore2D.x();
  theY=Vettore2D.y();
}
float Vettore2D::x() const {
  return theX;
}
float Vettore2D::y() const {
  return theY;
}
float Vettore2D::mod() const {
  return sqrt(pow(theX,2)+pow(theY,2));
}
```

Esempio

```
#include <iostream.h>
#include "Vettore2D.h"
int  main() {
  Vettore2D unPunto(0.,0.);
  Vettore2D* unAltroPunto=new Vettore2D(1.,2.);
  cout<<unPunto.x()<<unPunto.y()<<endl;
  cout<<unAltroPunto->x()<<unAltroPunto->y()<<endl;
  delete unAltroPunto;
}
```

Esempio: un vettore

Avremo bisogno di vettori 3D per molti scopi:

- Identificare una posizione in un sistema di riferimento (cartesiano, cilindrico, sferico)
- Memorizzare l'impulso di una particella
- ...

Una struttura di questo tipo può essere utilizzata molte volte in un grande progetto software. Cosa succede, ad esempio, al codice di un utente se cambia la rappresentazione di un vettore (da coordinate cartesiane a cilindriche)?

Linguaggio procedurale

È necessario cambiare tutti i punti del codice dove vengono eseguite operazioni con questi vettori.

Linguaggio Object Oriented

Niente, e vedremo come...

Vettore.h

Questo è l'*header* file per un vettore 3D in coordinate cartesiane.

Vettore.h

```
#ifndef VETTOREH
#define VETTOREH
class Vettore {
  public:
    Vettore() {}
    Vettore(const float&, const float&, const float&);
    Vettore(const Vettore&);
    ~Vettore() {}
    float x() const;
    float y() const;
    float z() const;
    float r() const;
    float phi() const;
  private:
    float theX, theY, theZ;
};
#endif
```

Vettore.cc

Ed ecco l'implementazione dei metodi:

Vettore.cc

```
#include <math.h>
#include "Vettore.h"
Vettore::Vettore(const float& x, const float& y,
                 const float& z):
       theX(x), theY(y), theZ(z) {}
Vettore::Vettore(const Vettore& vettore) {
  theX=vettore.x();
  theY=vettore.y();
  theY=vettore.z();
}
float Vettore::x() const {
  return theX;
}
float Vettore::y() const {
  return theY;
}
float Vettore::z() const {
  return theZ;
}
float Vettore::r() const {
  return sqrt(pow(theX,2)+pow(theY,2));
}
float Vettore::phi() const {
  return atan(y/x);
}
```

Esempio

Supponiamo adesso di utilizzare una classe, che descrive ad esempio un punto materiale, che ha un metodo che restituisce un vettore (la posizione).

Esempio

```
#include <iostream.h>
#include "Vettore.h"
int  main() {
    .
    .
    .
  Vettore posizionePunto=unPunto.posizione();
  cout<<" R="<<posizionePunto.r()<<endl;
    .
    .
    .
}
```

Se la rappresentazione interna della classe vettore cambia (da coordinate cartesiane a cilindriche), il nostro codice non deve subire nessuna modifica perché noi utilizziamo l'interfaccia della classe, e non direttamente i suoi dati nascosti.

Overloading operatori

La classe Vettore non è al momento molto utile ...
Sarebbe utile, ad esempio, poter sommare due vettori. Una possibilità è definire una *funzione*.

VettoreUtility.h

```
#ifndef VETTOREUTILITYH
#define VETTOREUTILITYH
Vettore somma(const Vettore& v1, const Vettore& v2) {
  return Vettore(v1.x()+v2.x(),v1.y()+v2.y(),v1.z()+v2.z());
}
#endif
```

Esempio

```
#include "Vettore.h"
#include "VettoreUtility.h"
int  main() {
    :
  Vettore posizionePunto=Vettore(3.,2.,1.);
  Vettore traslazione=Vettore(5.,0.,0.);
  Vettore nuovaPosizionePunto(somma(posizionePunto,
                                    traslazione));
    :
}
```

Overloading operatori

Il *C++* ci mette a disposizione degli strumenti più eleganti e funzionali per ottenere lo stesso scopo: la possibilità di fare l'*overloading* degli operatori.

```
#ifndef VETTOREH
#define VETTOREH
class Vettore {
  public:
      :
    Vettore operator+(const Vettore& v) const;
      :
};
```

```
#include "Vettore.h"
   :
Vettore Vettore::operator+(const Vettore& v) const {
  return Vettore(theX+v.theX,theY+v.theY,theZ+v.theZ);
}
   :
```

Un passo in più : gli array

La classe Vettore ci permette di fare molte cose, ma naturalmente non ci accontentiamo: vorremmo, per esempio, una classe che ci permetta di decidere la lunghezza del vettore a *run-time*, in modo da essere più flessibile.

In particolare, vorremmo le seguenti caratteristiche:

- Dimensionamento a *run-time*
- Accesso all'i-esimo elemento
- Utilizzo automatico della memoria, senza bisogno di scrivere esplicitamente nel codice `new` o `delete`

Un passo in più : gli array

Vediamo come e' possibile realizzare le nostre richieste:

FloatArray.h

```
#ifndef FLOATARRAYH
#define FLOATARRAYH
class FloatArray {
  public:
    FloatArray();
    FloatArray(const int&);
    FloatArray(const FloatArray&);
    ~Array();
    float operator[](int i);
    int size() const;
    FloatArray& operator=(const FloatArray&);
  private:
    int theSize;
    float* theArray;
    void copy(const FloatArray&);
};
#endif
```

Un passo in più : gli array

FloatArray.cc

```
#include "FloatArray.h"
FloatArray::FloatArray() {
  theSize=0;
  theArray=0;
}
FloatArray::FloatArray(const int& n) {
  theSize=n;
  theArray=new float[n];
}
FloatArray::FloatArray(const FloatArray& arr) {
  theSize=arr.theSize;
  theArray=new float[theSize];
  copy(arr);
}
FloatArray::~FloatArray() {
  delete[] theArray;
}
float FloatArray::operator[](int i) {
  return theArray[i];
}
int FloatArray::size() {
  return theSize;
}
```

Un passo in più : gli array

```
FloatArray& FloatArray::operator=(const FloatArray& arr) {
  if (theArray!=arr.theArray) {
    if (theSize!=arr.theSize) {
      delete[] theArray;
      theSize=arr.theSize;
      theArray=new float[theSize];
    }
    copy(arr);
  }
  return *this;
}
void FloatArray::copy(const FloatArray& arr) {
  float* p=theArray+theSize;
  float* q=arr.theArray+arr.theSize;
  while (p>theArray) *--p=*--q;
}
```

Non siamo ancora soddisfatti...

... perché essere obbligati ad usare i `float`?
Possiamo superare il problema usando una classe `template`.

Array.h

```
#ifndef ARRAYH
#define ARRAYH
template<class T> class Array {
  public:
    Array();
    Array(const int&);
    Array(const Array<T>&);
    ~Array();
    T operator[](int i);
    int size() const;
    Array<T>& operator=(const Array<T>&);
  private:
    int theSize;
    T* theArray;
    void copy(const Array<T>&);
};
#endif
```

Non siamo ancora soddisfatti...

Array.cc

```
#include "Array.h"
  :
template<class T>
T Array<T>::operator[](int i) {
  return theArray[i];
}
  :
template<class T>
Array<T>::Array(const Array<T>& arr) {
  theSize=arr.theSize;
  theArray=new T[theSize];
  copy(arr);
}
  :
```

I template all'opera

La classe Array e' molto flessibile: nessuno ci impedisce di definire array non solo di tipi come `float`, ma anche di qualunque altra classe.

Esempio

```
#include <iostream.h>
#include "Array.h"
#include "Point.h"
int  main() {
    :
  Array<float> unArray(5);
  unArray=3.;
  cout<<unArray[2]<<endl;
    :
  Array<Point> unaGriglia(2);
  unaGriglia[0]=Point(0.,0.);
  unaGriglia[1]=Point(3.,7.);
  cout<<unaGriglia[1].x()<<endl;
    :
}
```

La nostra classe array inizia a essere soddisfacente. Non ci dispiacerebbe, però, avere una classe che, quando usiamo un array di tipi numerici (ad esempio `float`), ci fornisca gli opportuni operatori matematici.

D'altra parte vorremmo anche riutillizare il codice scritto finora, e non dover ri-implementare i metodi già presenti in Array nella classe ArrayWithMath.

A questo fino possiamo sfruttare il meccanismo delle *classi derivate*.

ArrayWithMath.h

```
#ifndef ARRAYWITHMATHH
#define ARRAYWITHMATHH
#include "Array.h"
template<class T> class ArrayWithMath:public Array {
  public:
    ArrayWithMath();
    ArrayWithMath(const int&);
    ArrayWithMath(const Array<T>&);
    ~ArrayWithMath();
    ArrayWithMath<T>& operator+(const ArrayWithMath<T>&);
    T operator*(const ArrayWithMath<T>&);
};
#endif
```

ArrayWithMath.cc

```
#include <iostream.h>
#include "ArrayWithMath.h"
  :
template<class T>
ArrayWithMath<T>::ArrayWithMath(const int& n):
                                  Array<T>(n) { }
template<class T>
T Array<T>::operator*(const ArrayWithMath<T>& arr) {
  if (theSize!=arr.theSize)
    cerr<<"different size!"<<endl;
  else{
    T tot=0;
    T* p=theArray+theSize;
    T* q=arr.theArray+arr.theSize;
    while (p>theArray) tot+=(*p)*(*q);
  }
}
  :
```

Ereditarietà

Quando creiamo un'istanza di una classe derivata, questa comunque eredita i metodi delle classi che la precedono nella gerarchia di derivazione.

Mmh. . . , si spiega meglio con un esempio, vediamo:

Esempio

```
#include <iostream.h>
#include "ArrayWithMath.h"
int  main() {
    .
    .
    .
  ArrayWithMath<float> unArray(2);
  unArray[0]=1.;  unArray[1]=1.;
  ArrayWithMath<float> unAltroArray(2);
  unAltroArray[0]=2.;  unAltroArray[1]=3.;
  unArray=unArray+unAltroArray;
  cout<<unArray[0]<<unArray[1]<<endl;
  cout<<unArray.size()<<endl;
    .
    .
    .
}
```

Classi astratte

Stiamo realizzando un programma per disegnare figure geometriche sullo schermo. Per il momento ci limiteremo a disegnare quadrati.

Square.h

```
#ifndef SQUAREH
#define SQUAREH
#include "Point.h"
class Square {
  public:
    Square(const Point& pos, const float& l);
    ~Square() {}
    void draw(); //implementato in Square.cc
  private:
    Point thePosition;
    float the;
};
#endif
```

Designer.h

```
#ifndef DESIGNERH
#define DESIGNERH
#include "Square.h"
class Designer {
  public:
    :
    void draw(const Square* square) {
      square->draw();
    }
    :
};
#endif
```

Classi astratte

Il programma principale si presenta come:

Esempio

```
#include "Designer.h"
#include "Square.h"
int  main() {
    :
  Designer thePainter();
  Square* aSquare=new Square(Point(0.,0.),3.);
  thePainter.draw(aSquare);
    :
}
```

Cosa succede se i quadrati ci vengono a noia e decidiamo di voler disegnare anche qualcos'altro, tipo i cerchi?
Ci serve una classe Circle e dobbiamo aggiungere un metodo a Designer per disegnare un Circle.

Ma è proprio necessario?

Classi astratte

Square, Circle ed in genere tutte le figure geometriche che ci può venire in mente di disegnare hanno alcuni aspetti in comune:

- un metodo che ci restituisce la posizione sullo schermo
- la possibilità di spostare la figura
- un metodo per disegnarla

Quello che possiamo fare è definire una *interfaccia astratta* da cui derivano tutte le figure geometriche che vogliamo poter disegnare.

Shape.h

```
#ifndef SHAPEH
#define SHAPEH
class Point; class Vettore; class Shape {
  public:
    virtual ~ArrayWithMath() = 0;
    virtual void draw() = 0;
    virtual Point& position() = 0;
    virtual void move(const Vettore&) = 0;
};
#endif
```

Classi astratte

Tutte le figure geometriche derivano da Shape.

Square.h

```
#ifndef SQUAREH
#define SQUAREH
#include "Shape.h"
#include "Point.h"
class Square:public Shape {
  public:
    :
    virtual void draw(); //implementato in Square.cc
    :
};
#endif
```

Circle.h

```
#ifndef CIRCLEH
#define CIRCLEH
#include "Shape.h"
#include "Point.h"
class Circle:public Shape {
  public:
    :
    virtual void draw(); //implementato in Circle.cc
    :
};
#endif
```

Designer continuerà ad essere una classe molto semplice e non dovremo avere un metodo per ogni figura:

Designer.h

```
#ifndef DESIGNERH
#define DESIGNERH
#include "Shape.h"
class Designer {
  public:
    :
    :
    void draw(const Shape* shape) {
      square->draw();     }
    :
    :
};
#endif
```

Questo è quello che significa programmare interfacce!
E nel programma principale cosa succede?

Classi astratte

Vediamo come è possibile avere un codice semplice e di facile lettura, ma anche totalmente funzionale, se le interfacce sono state programmate (e pensate con cura).

Esempio

```
#include "Designer.h"
#include "Square.h"
#include "Circle.h"
#include "Array.h"
int  main() {
    .
    .
    .
  Designer thePainter();
  Square* aSquare=new Square(Point(0.,0.),3.);
  thePainter.draw(aSquare);
    .
    .
    .
  Shape* aSquareShape=new Square(Point(0.,0.),3.);
  thePainter.draw(aSquareShape);
    .
    .
    .
  Array<Shape*> allFigures(2);
  allFigures[0]=new Square(Point(0.,0.),3.);
  allFigures[1]=new new Circle(Point(1.,2.),1.5.);
  for (int i=0; i<allFigures.size(); i++)
    thePainter.draw(*allFigures[i]);
    .
    .
    .
}
```