Sommario

- Oggetti come attributi: lista di inizializzazione dei membri
- Parola chiave `const`:
  - Puntatori
  - Campi
  - Metodi
- Parola chiave `static`:
  - Variabili locali
  - Campi
  - Metodi

Oggetti come attributi

- Per inizializzare in modo appropriato un *dato membro oggetto* di una classe è necessario passare argomenti al costruttore dell'attributo.
- Per fare ciò è necessario utilizzare la lista di inizializzazione dei membri: se il membro è un'istanza di una classe, i valori iniziali sono passati al costruttore appropriato.

Oggetti come attributi: esempio

- Definiamo la classe `Point` che rappresenta i punti del piano in coordinate cartesiane.
- Definiamo la classe `Line` che rappresenta una linea che passa tra due punti del piano.
- Definiamo un numero minimo di metodi.

Oggetti come attributi: esempio

Point.h

```
#ifndef POINT_H
#define POINT_H

class Point{
   double x, y;
public:
   Point(double a=0, double b=0);
   ~Point();

   void SetX(double a);
   void SetY(double a);
   double GetX();
   double GetY();

   void Stampa();
};
#endif
```

Point.cpp

```
#include "Point.h"
#include <iostream>
using namespace std;
Point::Point(double a, double b){
   x = a;   y = b;
   cout<<"Point("<<x<<","<<y<<")"
   <<endl;
}
Point::~Point(){
   cout<<"~Point"<<endl;
}
void Point::SetX(double a){ x=a;}
void Point::SetY(double a){ y=a;}
double Point::GetX(){ return x;}
double Point::GetY(){ return y;}

void Point::Stampa(){
   cout<<"("<<x<<","<<y<<") ";
}
```

Oggetti come attributi: esempio

Line.h

```
#ifndef LINE_H
#define LINE_H

#include "point.h"

class Line{
   Point p1,p2;
public:
   Line(float x1, float y1,
        float x2, float y2);
   ~Line();

   void Stampa();
};
#endif
```

Oggetti come attributi

Line.cpp

```
#include "Point.h"
#include "Line.h"
#include <iostream>
using namespace std;
Line::Line(float x1, float y1,
           float x2, float y2) :
        p1(x1,y1), p2(x2,y2)
{
   cout<<"Line()"<<endl;
   //p1(x1,y1);//errore:
                //compile-time
}
Line::~Line(){
   cout<<"~Line()"<<endl;
}
void Line::Stampa(){
   cout<<"\tL: ";
   p1.Stampa(); p2.Stampa();
   cout<<endl;
}
```


Oggetti come attributi: esempio

Test.cpp

```
#include "Point.h"
#include "Line.h"
#include <iostream>
using namespace std;

void test1();

int main(){

   test1();

   return 0;
}

void test1(){
   Line l1(1,1,3,3);
   l1.Stampa();
}
```

Una possibile uscita

```
Point(1,1)
Point(3,3)
Line()
      L: (1,1) (3,3)
~Line()
~Point
~Point
```

Sono chiamati i costruttori dei dati membro e i relativi distruttori

Oggetti come attributi

- La differenza tra l'uso della *lista di inizializzazione* dei membri e l'*assegnamento* dei dati membro nel corpo del costruttore è che solo la prima fornisce un'inizializzazione, cioè l'invocazione del costruttore appropriato per il dato membro.
- Nel corpo del costruttore possono avvenire solo assegnazioni, questo può essere fonte di errori e *inefficienza*.
- Prima dell'assegnamento nel corpo del costruttore è implicitamente chiamato il *costruttore di default* del membro istanza di una classe.

Oggetti come attributi

```
Line::Line(float x1, float y1, float x2, float y2){
      cout<<"Line()"<<endl;
      //p1.x=x1;//errore: compile-time
      p1.SetX(x1); p1.SetY(y1);
      p2.SetX(x2); p2.SetY(y2);
   }
```

Sostituendo il costruttore definito precedentemente con questo privo della lista di inizializzazione dei membri, si ottiene un'uscita diversa durante il test.

Una possibile uscita

```
Point(0,0)
Point(0,0)
Line()
      L: (1,1) (3,3)
~Line()
~Point
~Point
```

Puntatori e `const`

- Per poter utilizzare un puntatore con un oggetto costante, si deve definire un puntatore che *punta ad un oggetto costante*:

```
const int bufsize = 512;
//int *pt = & bufsize; //errore:
                       //compile-time
const int *pt = & bufsize;

//*pt = 55; //errore: compile-time

int i =0;
pt = &i;
```

Il puntatore non è costante

Puntatori e `const`

- È possibile definire un *puntatore costante* (esso stesso costante) sia ad un oggetto costante sia ad un oggetto non constante:

Notare la posizione di `const`

```
int a=1, b=2;
int *const pt = &a;
*pt=33; // a vale 33

//pt= &b; // errore: compile-time
```


Puntatori e `const`

- Nei programmi reali si usa molto spesso un puntatore ad un oggetto costante come parametro formale di una funzione, per indicare che l'oggetto effettivo passato alla funzione non è modificato al suo interno:

```
int str_dim(const char *p);
```


Puntatori e `const`

```
int str_dim(const char *p){
  int i=0;
  //*p='a';//errore:compile-time
  while(*p++)  i++;
  return i;
}
```

Una possibile uscita

```
prova
La stringa e`lunga 5 caratteri
```

```
void test2(){
  string str;
  cin>>str;
  int n=str_dim(str.c_str());
  cout<<"La stringa e`lunga "<<n<<" caratteri"<<endl;
}
```

Campi e `const`

- I dati membro `const` di una classe devono essere sempre inizializzati nella lista di inizializzazione dei membri:

```
class Prova{
        //const int size=100;//errore: compile-time
        const int size;
public:
        Prova(int i);
        ~Prova();
         //...
};
```

```
Prova::Prova(int i): size(i){
        //size = i; //errore: compile-time
        cout<<"Prova("<<size<<")"<<endl;
}
Prova::~Prova(){cout<<"~Prova("<<size<<")"<<endl;}
//...
```

Sono mostrati solo gli aspetti essenziali

Metodi e `const`

- Per modificare lo stato di un oggetto è necessario invocare una funzione membro pubblica: per rispettare la caratteristica `const` di un oggetto, è necessario distinguere tra i metodi che possono modificare lo stato dell'oggetto e quelli che non lo modificano.
- Chi progetta la classe indica i metodi che non modificano lo stato come `const`: la parola chiave `const` è inserita tra la lista dei parametri e il corpo della funzione.
- Modifichiamo di conseguenza la classe `Point`.

Metodi e `const` : esempio

Point.h

```
#ifndef POINT_H
#define POINT_H
class Point{
   double x, y;
public:
   Point(double a=0, double b=0);
   ~Point();
   void SetX(double a);
   //void SetX(double a)const;
   void SetY(double a) ;
   double GetX()const;
   double GetY()const;
   void Stampa()const;
};
#endif
```

Errore: compile-time

Point.cpp

```
#include "Point.h"
#include <iostream>
using namespace std;
Point::Point(double a, double b){
   x = a;   y = b;
   cout<<"Point("<<x<<","<<y<<")"<
   <endl;
}
Point::~Point(){cout<<"~Point"<<
   endl;}
void Point::SetX(double a){ x=a;}
void Point::SetY(double a){ y=a;}
double Point::GetX()const { return
   x;}
double Point::GetY()const { return
   y;}
void Point::Stampa()const {
   cout<<"("<<x<<","<<y<<") ";
}
```


Metodi e `const` : esempio

```
void test1(){
   Point p1;
   p1.SetX(33);
   cout<<p1.GetX()<<endl;

   const Point p2(-2,5);
   //p2.SetX(11); //errore:compile-time
   cout<<p2.GetX()<<endl;
}
```

Una possibile uscita

```
Point(0,0)
33
Point(-2,5)
-2
~Point
~Point
```

Variabili locali e `static`

- All'interno di una funzione è possibile dichiarare un oggetto locale che duri per tutta la durata del programma. Tale comportamento è ottenuto dichiarando l'oggetto `static`.
- Un oggetto locale statico è inizializzato la prima volta che l'esecuzione del programma passa attraverso la dichiarazione dell'oggetto.
- L'esempio mostra due funzioni che incrementano di un'unita` il valore della coordinata x di un oggetto punto locale.

Variabili locali e `static`

```
#include <iostream>
#include "Point.h"
using namespace std;
void somma_s(){
   static Point p1(2,2);
   p1.SetX(p1.GetX()+1);
   cout<<p1.GetX()<<" ";
}
void somma_n(){
   Point p1(2,2);
   p1.SetX(p1.GetX()+1);
   cout<<p1.GetX()<<" ";
}
int main(){
   for(int i=0; i<5;i++)
        somma_s();
   cout<<endl;
   cout<<"---------------"<<endl;
   for(int i=0; i<5;i++)
        somma_n();
   cout<<"---------------"<<endl;
}
```

Una possibile uscita

```
Point(2,2)
3 4 5 6 7
---------------
Point(2,2)
3 ~Point
Point(2,2)
3 ~Point
Point(2,2)
3 ~Point
Point(2,2)
3 ~Point
Point(2,2)
3 ~Point
---------------
~Point
```


Campi, metodi e `static`

- Talvolta è utile che tutti gli oggetti di una particolare classe accedano ad un *dato comune* (per esempio per tener conto di quanti oggetti sono stati creati). Esiste una sola copia di un *dato membro statico* per tutta la classe, al contrario degli altri dati membro, di cui ogni oggetto ha la propria copia.
- Un dato membro è reso statico inserendo nella dichiarazione la parola chiave `static`.
- Il valore di un dato membro statico può cambiare nel tempo.

Campi, metodi e `static`

- In generale, un dato membro statico è *inizializzato all'esterno* della definizione della classe e poiché si può fornire una *sola definizione*, non deve essere inserito nel file header.

- I metodi che accedono solo a dati membro statici dovrebbero essere dichiarati `static`, in tal modo possono non essere riferiti ad una istanza particolare della classe.
- Un metodo statico non ha il puntatore `this`, pertanto qualsiasi riferimento *implicito* (per esempio accedere ad un attributo non statico) od *esplicito* a tale puntatore provoca un errore di compilazione.

Campi, metodi e `static` : esempio

Point.h

```
#ifndef POINT_H
#define POINT_H
class Point{
   double x, y;
   static int n;
public:
   Point(double a=0, double b=0);
   ~Point();
   void SetX(double a);
   void SetY(double a) ;
   double GetX()const;
   double GetY()const;

   void Stampa()const;

   static int GetN();
};
#endif
```

Si modifica la classe Point in modo tale da tener conto del numero di oggetti creati.

Campo statico

Metodo statico

Point.cpp Campi, metodi e `static` : esempio

```
#include "Point.h"
#include <iostream>
using namespace std;
int Point::n=0;
Point::Point(double a, double b){
   x = a;    y = b;
   n++;
   cout<<"Point("<<x<<","<<y<<")"<<endl;
}
Point::~Point(){
   n--;
   cout<<"~Point"<<endl;
}
void Point::SetX(double a){ x=a;}
void Point::SetY(double a){ y=a;}
double Point::GetX()const { return x;}
double Point::GetY()const { return y;}
void Point::Stampa()const {
   cout<<"("<<x<<","<<y<<") ";
}
int Point::GetN(){
   return n;
}
```

Definizione del campo statico

Il costruttore incrementa il contatore degli oggetti creati

Il distruttore è necessario per decrementare il contatore degli oggetti creati

Campi, metodi e `static` : esempio

```
#include <iostream>
#include "Point.h"
using namespace std;
void test1();
void test2();

int main(){
   test1();
   cout<<"============"<<endl;
   cout<<Point::GetN()<<endl;
   return 0;
}
void test1(){
   cout<<Point::GetN()<<endl;
   cout<<"-------------"<<endl;
   Point a;
   Point *p=new Point(3,3);
   cout<<a.GetN()<<endl;
   cout<<"++++++++++++"<<endl;
   test2();
   cout<<"++++++++++++"<<endl;
   cout<<Point::GetN()<<endl;
   cout<<"-------------"<<endl;
   delete p;
}
void test2(){
   Point v[3];
   cout<<v[0].GetN()<<endl;
}
```

Una possibile uscita

```
0
------------
Point(0,0)
Point(3,3)
2
++++++++++++
Point(0,0)
Point(0,0)
Point(0,0)
5
~Point
~Point
~Point
++++++++++++
2
------------
~Point
~Point
============
0
```