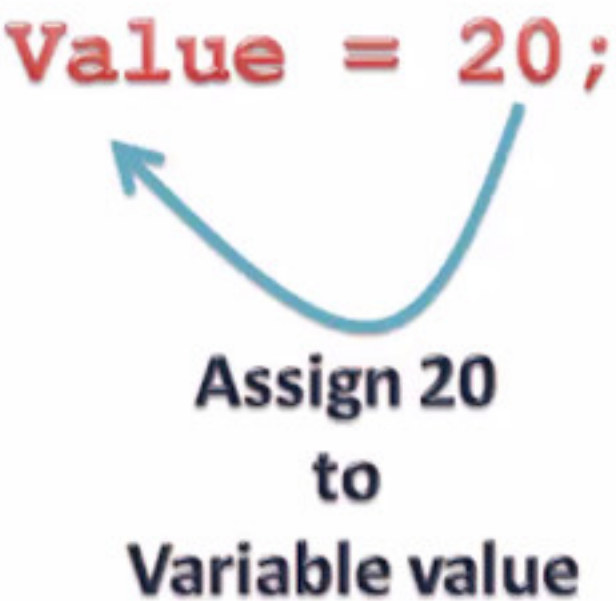
Costruttore di copia standard nelle classi derivate


```
class Z {
public:
  Z() {cout << "Z0 ";}
  Z(const Z& x) {cout << "Zc ";}
};

class C {
private:
  Z w;
public:
  C() {cout << "C0 ";}
  C(const C& x): w(x.w) {cout << "Cc ";}
};

class D: public C {
private:
  Z z;
public:
  D() {cout << "D0 ";}
  // costruttore di copia standard
};

int main() {
  D d; cout << "UNO\n";
  D e = d; cout << "DUE"; // costruttore di copia standard
}
// stampa:
// Z0 C0 Z0 D0 UNO
// Zc Cc Zc DUE
```


```
class Z {
public:
  Z() {cout << "Z0 ";};
  Z(const Z& x) {cout << "Zc ";}
};

class C {
private:
  Z w;
public:
  C() {cout << "C0 ";}
  C(const C& x): w(x.w) {cout << "Cc ";}
};

class D: public C {
private:
  Z z;
public:
  D() {cout << "D0 ";};
  D(const D& x) {cout << "Dc ";} // ridefinzione costr.copia
};

int main() {
  D d; cout << "UNO\n";
  D e = d; cout << "DUE"; // costruttore di copia ridefinito
}
// stampa:
// Z0 C0 Z0 D0 UNO
// Z0 C0 Z0 Dc DUE       // ATTENZIONE!
```

Assegnazione standard nelle classi derivate



```
class Z {
public:
  Z() {cout << "Z0 ";}
  Z(const Z& x) {cout << "Zc ";}
  Z& operator=(const Z& x) {cout << "Z= "; return *this;}
};

class C {
protected:
  Z w;
public:
  C() {cout << "C0 ";}
  C(const C& x): w(x.w) {cout << "Cc ";}
  C& operator=(const C& x) {w=x.w; cout << "C= "; return *this;}
};

class D: public C {
private:
  Z z;
public:
  D() {cout << "D0 ";}
  D(const D& x) {cout << "Dc ";}
};
int main() {
  D d; cout << "UNO\n";
  D e; cout << "DUE\n";
  e=d; cout << "TRE"; // assegnazione standard
}
// stampa:
// Z0 C0 Z0 D0 UNO
// Z0 C0 Z0 D0 DUE
// Z= C= Z= TRE
```

EXAMPLE

```
class Z {
public:
  int x;
  Z(): x(0) {cout << "Z0 ";}
  Z(const Z& x) {cout << "Zc ";}
  Z& operator=(const Z& x) {cout << "Z= "; return *this;}
};
class C {
public:
  Z w;
  C() {cout << "C0 ";}
  C(const C& x): w(x.w) {cout << "Cc ";}
  C& operator=(const C& x) {w=x.w; cout << "C= "; return *this;}
};
class D: public C {
public:
  Z z;
  D() {cout << "D0 ";}
  D(const D& x) {cout << "Dc ";}
  D& operator=(const D& x) {z=x.z; cout << "D= "; return *this;}
  // assegnazione definita male: chi ci pensa ad assegnare il
  // campo dati w di C? e se w fosse private?
};
int main() {
  D d; d.w.x = 3; cout << "UNO\n";
  D e; e.w.x = 5; cout << "DUE\n";
  e=d; cout << "TRE\n";
  cout << e.w.x << ' ' << d.w.x << " QUATTRO";
}
// stampa:
// Z0 C0 Z0 D0 UNO
// Z0 C0 Z0 D0 DUE
// Z= D= TRE
// 5 3 QUATTRO
```

B& b= *this;
b=X;
proposta contorta ma funziona, dal tracciamento evidenzia che chiama prima il costruttore della base

assegnazione ben definita

x=w.x; z=x.z;

esempio
importante
per i compiti

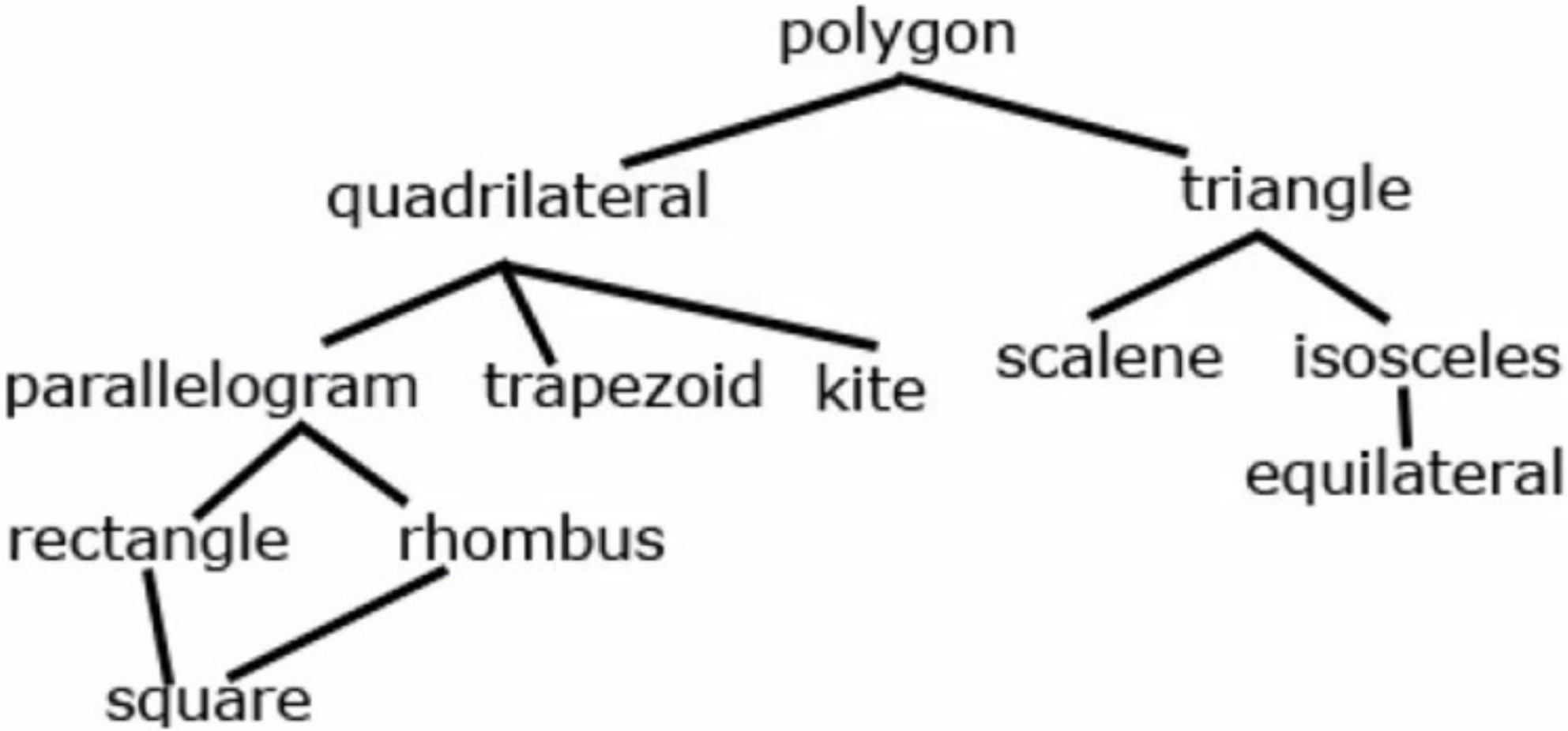
```
class B {
private:
  int x;
public:
  B(int k=1): x(k) {}
  B& operator=(const B& a) {x=a.x;}
  void print() const {cout << "x="<<x;}
};
class D: public B {
private:
  int z;
public:
  D(int k=2): B(k), z(k) {}
  // assegnazione con comportamento standard
  D& operator=(const D& x) {
    this->B::operator=(x); // assegnazione per sottooggetto
    z = x.z;
  }
  void print() const {B::print(); cout <<" z="<< z;}
};

int main() {
  D d1(4), d2(5);
  d1.print(); cout << endl; // x=4 z=4
  d2.print(); cout << endl; // x=5 z=5
  d1=d2;
  d1.print(); // x=5 z=5
}
```

Distruttore standard nelle classi derivate

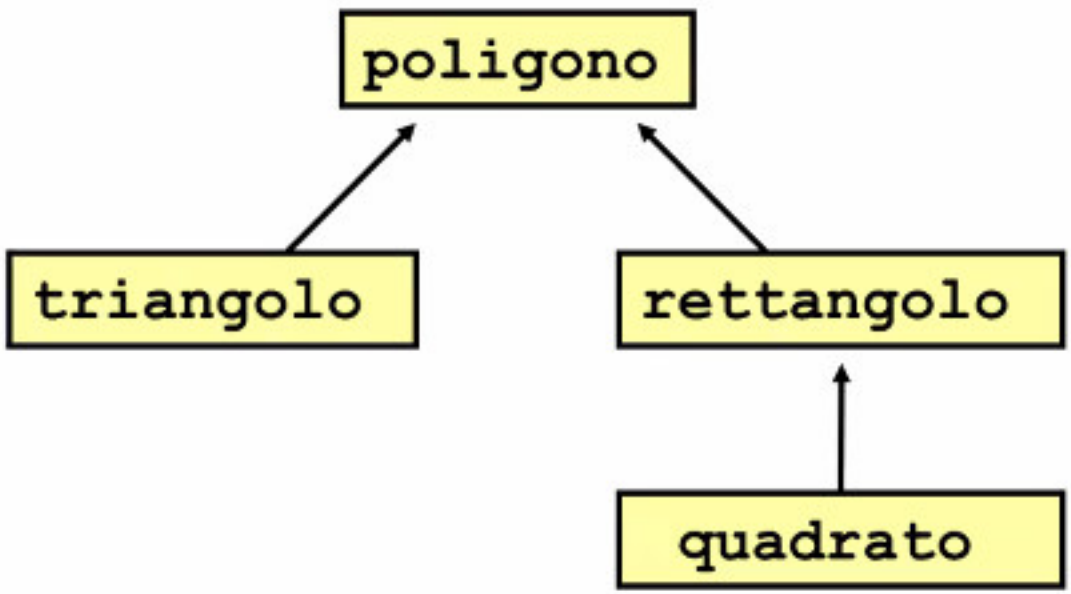

```
class Z {
public:
  Z() {cout << "Z0 ";}
  ~Z() {cout <<"~Z ";}
};

class C {
private:
  Z w;
public:
  C() {cout << "C0 ";}
  ~C() {cout << "~C ";}
};

class D: public C {
private:
  Z z;
public:
  D() {cout << "D0 ";}
};

int main() {
  D* p = new D; cout << "UNO\n";
  delete p; cout << "DUE"; // distruttore standard
}
// Z0 C0 Z0 D0 UNO
// ~Z ~C ~Z DUE
```


```
class Z {
public:
  Z() {cout << "Z0 ";}
  ~Z() {cout <<"~Z ";}
};

class C {
private:
  Z w;
public:
  C() {cout << "C0 ";}
  ~C() {cout << "~C ";}
};

class D: public C {
private:
  Z z;
public:
  D() {cout << "D0 ";}
  ~D() {cout <<"~D ";}
};

int main() {
  D* p = new D; cout << "UNO\n";
  delete p; cout << "DUE"; // distruttore ridefinito
}
// stampa:
// Z0 C0 Z0 D0 UNO
// ~D ~Z ~C ~Z DUE
```

Esempio di derivazione




EXERCISE


poligono
triangolo
rettangolo
quadrato

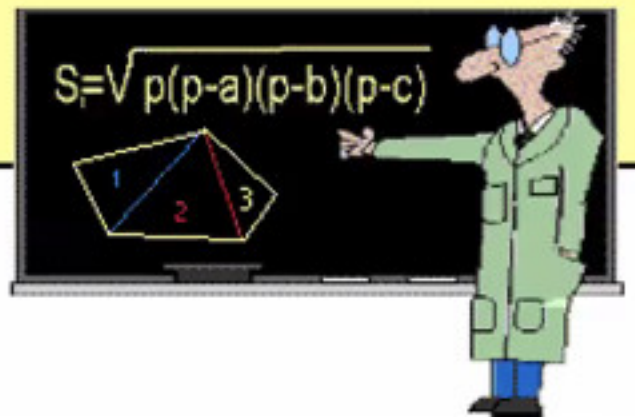
```
// file pol.h
#ifndef POL_H
#define POL_H

class punto {
 private:
  double x, y;
 public:
  punto(double a=0, double b=0): x(a), y(b) {}
  // metodo statico che calcola la distanza tra due punti
  static double lung(const punto& p1, const punto& p2);
};

class poligono {
 protected:
  unsigned int nvertici;
  punto* pp; // array dinamico di punti, nessun controllo di consistenza
 public:
   // si assume v array ordinato degli n vertici
  poligono(unsigned int n, const punto v[]);
  ~poligono();                                  // distruttore profondo
  poligono(const poligono&);                    // copia profonda
  poligono& operator=(const poligono&); // assegnazione profonda
  double perimetro() const; // ritorna il perimetro del poligono
};
#endif

// file pol.cpp
// HOMEWORK
```

```
// file ret.h
#ifndef RET_H
#define RET_H
#include "pol.h"
class rettangolo: public poligono {// rettangolo è un poligono specializzato
 public:
  rettangolo(const punto v[]); // nvertici == 4
  double perimetro() const;    // ridefinizione
  double area() const;         // metodo proprio di rettangolo
};
#endif

// file ret.cpp
#include "ret.h"

// NB: nessun controllo che i punti di v formino un rettangolo
rettangolo::rettangolo(const punto v[]) : poligono(4, v) {}

// specializzazione della funzionalità di calcolo del perimetro
double rettangolo::perimetro() const {
  double base = punto::lung(pp[1], pp[0]);
  double altezza = punto::lung(pp[2], pp[1]);
  return ((base + altezza)*2);
}

double rettangolo::area() const {
  double base = punto::lung(pp[1], pp[0]);
  double altezza = punto::lung(pp[2], pp[1]);
  return (base * altezza);
}
```

```
// file ret.h
#ifndef RET_H
#define RET_H
#include "pol.h"
class rettangolo: public poligono {// rettangolo è un poligono specializzato
 public:
  rettangolo(const punto v[]); // nvertici == 4
  double perimetro() const;    // ridefinizione
  double area() const;         // metodo proprio di rettangolo
};
#endif
```

```
// file qua.h
#ifndef QUA_H
#define QUA_H
#include "ret.h"
// quadrato è un rettangolo specializzato
class quadrato: public rettangolo {
 public:
  quadrato(const punto v[]); // invoca quello di rettangolo
  double perimetro() const;  // ridefinizione
  double area() const;       // ridefinizione
};
#endif

// file qua.cpp
#include "qua.h"
// nessun controllo che i punti di v formino un quadrato
quadrato::quadrato(const punto v[]) : rettangolo(v) {}

// specializzazione della funzionalità di calcolo del perimetro
double quadrato::perimetro() const {
  double lato = punto::lung(pp[1], pp[0]);
  return (lato * 4);
}

// specializzazione della funzionalità di calcolo dell'area
double quadrato::area() const {
  double lato = punto::lung(pp[1], pp[0]);
  return (lato * lato);
}
```

```
// file tri.h
#ifndef TRI_H
#define TRI_H
#include "pol.h"

class triangolo: public poligono {
 public:
  triangolo(const punto v[]); // 3 vertici
  double area() const;        // metodo proprio di triangolo
                              // perimetro() ereditato
};
#endif
// file tri.cpp
#include "tri.h"
#include <math.h>

triangolo::triangolo(const punto v[]) : poligono(3, v) {}

double triangolo::area() const { // usa la formula di Erone
  double p = perimetro()/2;
  double a=punto::lung(pp[1],pp[0]), b=punto::lung(pp[2],pp[1]),
         c=punto::lung(pp[0],pp[2]);
  return sqrt(p*(p-a)*(p-b)*(p-c));
}
```



```
// file main.cpp
#include <iostream>
#include "pol.h"
#include "tri.h"
#include "ret.h"
#include "qua.h"
using namespace std;

int main() {
  int i; punto v[4]; double x,y;
  cout << "Scrivi le coordinate di un triangolo" << endl;
  for (i = 0; i < 3; i++) { cin >> x >> y; v[i]=punto(x,y); }
  const triangolo tri(v);
  cout << "Scrivi le coordinate di un rettangolo" << endl;
  for (i = 0; i < 4; i++) { cin >> x >> y; v[i]=punto(x,y); }
  rettangolo ret1(v), ret2 = ret1;
  cout << "Scrivi le coordinate di un quadrato" << endl;
  for (i = 0; i < 4; i++) { cin >> x >> y; v[i]=punto(x,y); }
  quadrato qua1(v), qua2;
  qua2 = qua1;
  cout << "Triangolo:\n" << tri.perimetro()
       << '\t' << tri.area() << endl;
  cout << "Rettangolo:\n" << ret2.perimetro()
       << '\t' << ret2.area() << endl;
  cout << "Quadrato:\n" << qua2.perimetro()
       << '\t' << qua2.area() << endl;
}
```

ESERCIZIO. Definire una superclasse ContoBancario e due sue sottoclassi ContoCorrente e ContoDiRisparmio che soddisfano le seguenti specifiche:

1. Ogni ContoBancario è caratterizzato da un saldo e rende disponibili due funzionalità di deposito e prelievo: double deposita(double) e double preleva(double) che ritornano il saldo aggiornato dopo l'operazione di deposito/prelievo.
2. Ogni ContoCorrente è caratterizzato anche da una spesa fissa uguale per ogni ContoCorrente che deve essere detratta dal saldo ad ogni operazione di deposito e prelievo.
3. Ogni ContoDiRisparmio deve avere un saldo non negativo e pertanto non tutti i prelievi sono permessi; d'altra parte, le operazioni di deposito e prelievo non comportano costi aggiuntivi e restituiscono il saldo aggiornato.
4. Si definisca inoltre una classe ContoArancio derivata da ContoDiRisparmio. La classe ContoArancio deve avere un ContoCorrente di appoggio: quando si deposita una somma S su un ContoArancio, S viene prelevata dal ContoCorrente di appoggio; d'altra parte, i prelievi di una somma S da un ContoArancio vengono depositati nel ContoCorrente di appoggio.

CONTO ARANCIO

```
/*
ESERCIZIO. Definire una superclasse ContoBancario e due sue sottoclassi ContoCorrente e
ContoDiRisparmio che soddisfano le seguenti specifiche:

1) Ogni ContoBancario è caratterizzato da un saldo e rende disponibili due funzionalità di
deposito e prelievo: double deposita(double) e double preleva(double) che ritornano il saldo
aggiornato dopo l'operazione di deposito/prelievo.
2) Ogni ContoCorrente è caratterizzato anche da una spesa fissa uguale per ogni ContoCorrente che
deve essere detratta dal saldo ad ogni operazione di deposito e prelievo.
3) Ogni ContoDiRisparmio deve avere un saldo non negativo e pertanto non tutti i prelievi sono
permessi; d'altra parte, le operazioni di deposito e prelievo non comportano costi aggiuntivi e
restituiscono il saldo aggiornato.
4) Si definisca inoltre una classe ContoArancio derivata da ContoDiRisparmio. La classe
ContoArancio deve avere un ContoCorrente di appoggio: quando si deposita una somma S su un
ContoArancio, S viene prelevata dal ContoCorrente di appoggio; d'altra parte, i prelievi di una
somma S da un ContoArancio vengono depositati nel ContoCorrente di appoggio.
*/

class ContoBancario {
private:
  double saldo;
public:
  double deposita(double x) {
    return x>=0 ? saldo += x : saldo;
  }
  double preleva(double x){
    return x>=0 ? saldo -= x : saldo;
  }
  double getSaldo() const {return saldo;}
};

class ContoCorrente: public ContoBancario {
private:
  static double spesaFissa;
public:
  // se x<spesaFissa, non avviene il deposito
   double deposita(double x) {
     return ContoBancario::deposita(x-spesaFissa);
   }
  double preleva(double x){
    return ContoBancario::preleva(x+spesaFissa);
  }
};
double ContoCorrente::spesaFissa = 1.0;

class ContoDiRisparmio: public ContoBancario {
public:
  // Invariante: saldo >= 0
  double preleva(double x){
    return x<=getSaldo() ? ContoBancario::preleva(x) : getSaldo();
  }
  // ContoBancario::deposita() non necessita di ridefinizione
};

class ContoArancio: public ContoDiRisparmio {
  // conto di appoggio deve essere modificabile
};
```