Ereditarietà. Programmazione OO, metodi astratti, interfacce

Laboratorio di Metodi Computazionali e Statistici (2023/2024)

Fabrizio Parodi

Dipartimento di Fisica

Correzione esercizio riassuntivo

Si parte dalla classe `Vector3.h`, `Vector3.cpp`

- Estensione delle funzionalità: modulo e versore (entrambi sfruttando *this*) e opzionalmente prodotto scalare e vettore (non saranno sfruttate nell'esercitazione ma sono in generale utili per una classe Vector completa)
- Creazione di un programma di test che:
  - Lettura da file di un vector di Vector (da file `punti.dat`)
  - Calcolo del vettore somma (con range-based-for-loop)

Per partire da quanto fatto a lezione date il comando

```
git clone https://github.com/fabrizio-parodi/LabMCS-Es0.git Es0
```

Correzioni Esempi uso STL

- Elenco telefonico (associazione nome-numero) con ricerca di numero
- Elenco studenti ordinati per anno di nascita

Ricerca in un mini-elenco telefonico

```
1  #include <map>
2  int main(){
3    map<string,int> elenco;
4    elenco["Rossi"]   = 340352015;
5    elenco["Verdi"]   = 335289751;
6    elenco["Bianchi"] = 318103456;
7
8    auto it=elenco.begin();
9    while(it!=elenco.end()){
10     cout << it->first << " " << it->second << endl;
11     it++;
12   }
13
14   it=elenco.find("Rossi");
15   if (it!=elenco.end())
16     cout << ite->second << endl;
17   else
18     cout << "Not found" << endl;
19
20 }
```

Elenco studenti ordinati per anno di nascita

Supponiamo di voler ordinare per anno di nascita un'elenco di studenti.

- Creiamo intanto una classe contenente cognome ed anno di nascita per ogni studente.
- Definiamo il $<$ di quella classe in modo che lavori sull'anno
- Creiamo un vettore di oggetti
- Utilizziamo `sort` per ordinarli

Elenco studenti ordinati per anno di nascita

Creiamo intanto una classe contenente cognome ed anno di nascita per ogni studente.

```
1  class stud{
2  public :
3    stud(string nome="",int anno=0):m_nome(nome),m_anno(anno){}
4    bool operator<(const stud& s) const {return m_anno< s.m_anno;}
5    int    anno() const {return m_anno;}
6    string nome() const {return m_nome;}
7  private :
8    string m_nome;
9    int m_anno;
10 };
```

Elenco studenti ordinati per anno di nascita

```
1  int main(){
2
3    vector<stud> elenco;
4    stud p1("Rossi"    ,1989) ;
5    stud p2("Verdi"    ,1990) ;
6    stud p3("Bianchi"  ,1988) ;
7    elenco.push_back(p1);
8    elenco.push_back(p2);
9    elenco.push_back(p3);
10
11   sort (elenco.begin(),elenco.end ()) ;
12
13   for (auto s:elenco)
14     cout << s.nome() << " " << s.anno() << endl;
15
16   return 0;
17
18 }
```

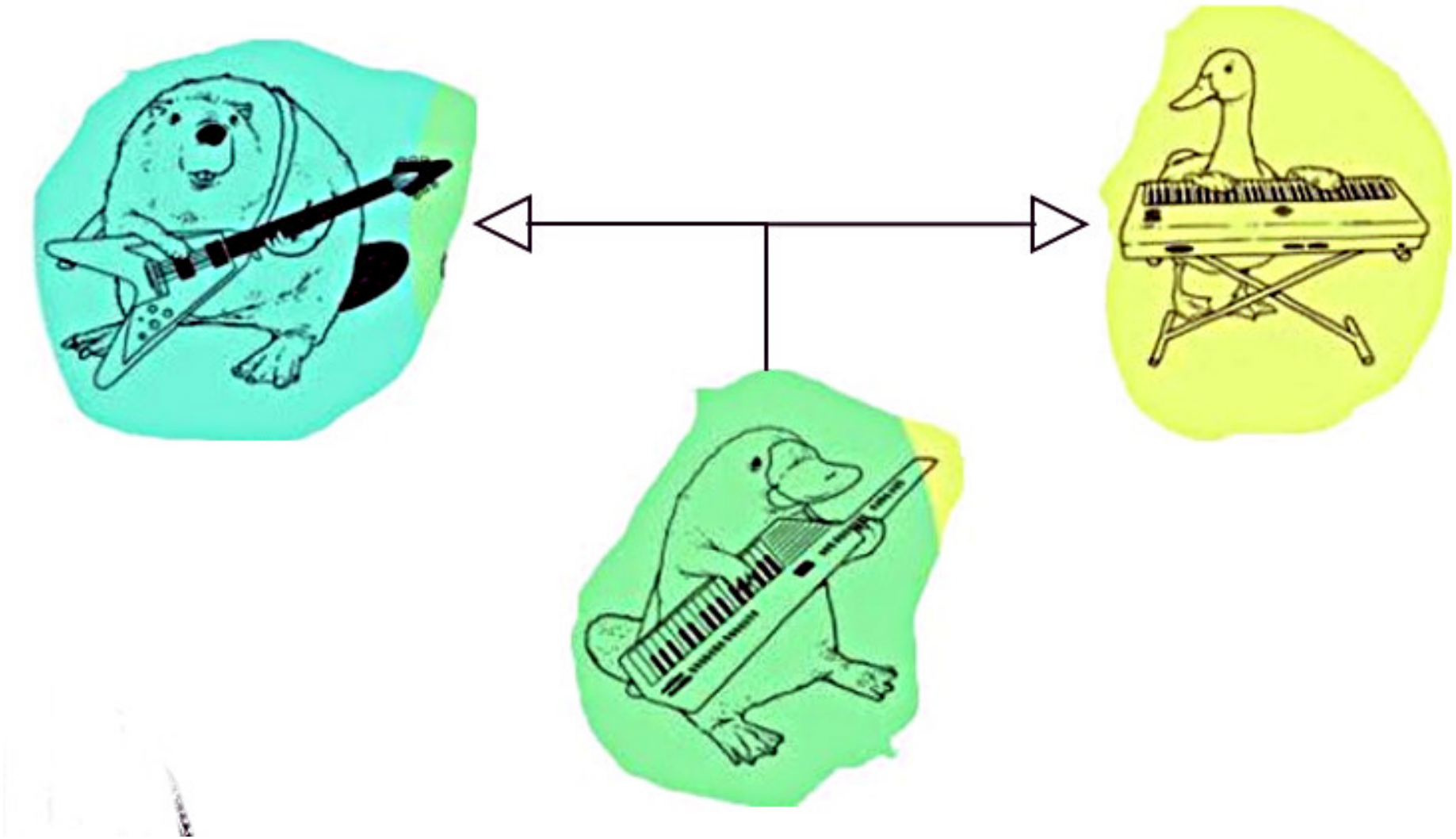
Elenco studenti ordinati per anno di nascita

Analogo ma con funzione di confronto esterna: non è necessario definire il minore nella classe `stud` (utile soprattutto se non si è il "possessore" della classe).

```
1  bool comp(const stud& a, const stud& b){
2    if (a.anno()<b.anno())
3      return true;
4    else
5      return false;
6  }
7
8  int main(){
9    vector<stud> el;
10   el.push_back(stud("Rossi",1989));    // simile a prima... solo
11   el.push_back(stud("Verdi",1990));    // p1, p2... non sono creati
12   el.push_back(stud("Bianchi",1988)); // ma vengono creati al volo
13                                       // gli stud da mettere in el
14   sort (el.begin(),el.end (),comp) ;
15
16   for (auto s:el){
17     cout << s.nome() << " " << s.anno() << endl;
18   }
19
20   return 0;
21
```

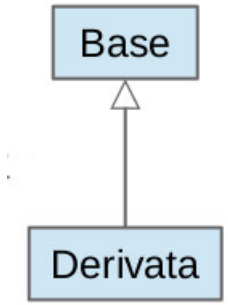
Ereditarietà

Ereditarietà



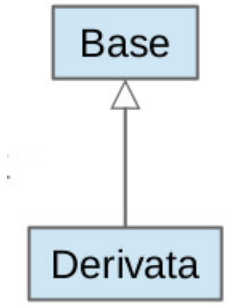
Ereditarietà

- È uno dei concetti base della programmazione OO: è una relazione di generalizzazione/specificazione tra classi
  - Una classe base definisce un concetto generale mentre la classe da essa derivata ne rappresenta una variante specifica.



Ereditarietà

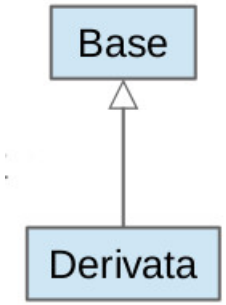
- È uno dei concetti base della programmazione OO: è una relazione di generalizzazione/specificazione tra classi
  - Una classe base definisce un concetto generale mentre la classe da essa derivata ne rappresenta una variante specifica.



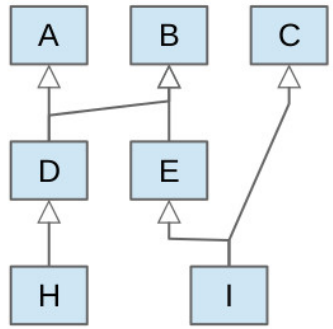
- Facilita il riutilizzo del codice:
  - Invece che re-implementare le caratteristiche comuni, la classe derivata eredita i dati e metodi della classe base
  - Eliminando le duplicazioni si riducono le dimensioni del codice e le possibilità di errore.

Ereditarietà

- È uno dei concetti base della programmazione OO: è una relazione di generalizzazione/specificazione tra classi
  - Una classe base definisce un concetto generale mentre la classe da essa derivata ne rappresenta una variante specifica.



- Facilita il riutilizzo del codice:
  - Invece che re-implementare le caratteristiche comuni, la classe derivata eredita i dati e metodi della classe base
  - Eliminando le duplicazioni si riducono le dimensioni del codice e le possibilità di errore.
- Permette di realizzare una gerarchia di classi. A seconda del linguaggio può essere:
  - singola: una classe eredita da una sola classe base.
  - multipla: una classe può derivare da più classi base (possibile in C++).

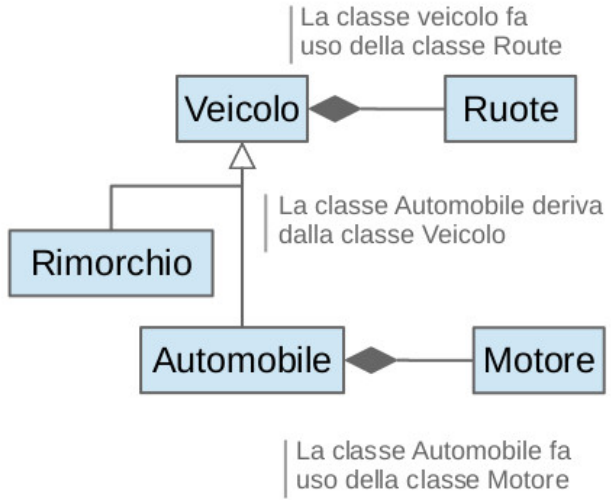


UML

- Uniform Modeling Language (UML) è un linguaggio per modellazione per la programmazione OO
  - fornisce una rappresentazione grafica delle classi e delle relazioni tra di esse

UML

- Uniform Modeling Language (UML) è un linguaggio per modellazione per la programmazione OO
  - fornisce una rappresentazione grafica delle classi e delle relazioni tra di esse
- Come funziona ?
  - Un rettangolo rappresenta una classe
  - Una freccia a triangolo indica una relazione di ereditarietà tra classi: la classe base è quella "puntata"
  - Un freccia a rombo indica invece una relazione di composizione: cioè una classe contiene oggetti di un'altra classe. Il rombo è disegnato dal lato della classe ospitante



Specificatore di accesso all'interno di una classe

```
1  class myclass{
2    public:
3      ...
4    protected:
5      ...
6    private:
7  };
```

Un membro o metodo dichiarato nel blocco:

- **public** è accessibile a tutti
- **private** è accessibile solo ai membri della classe
- **protected** (qualificatore di accesso specifico per l'ereditarietà) è accessibile dalle classi derivate (per loro è come se fosse public) ma è privato per il resto del programma.

Specificazione di accesso

- La sintassi per definire una classe derivata è

```
class Derivata : public Base1, public Base2, ... {
   // Interfaccia delle classe Derivata
};
```

- I nomi delle classi dopo : specificano da che classi base deriva la classe
- La keyword prima del nome delle classe basi indica il "tipo" di derivazione
- La classe derivata eredita tutti i dati membro e metodi delle sue classi base e delle eventuali classi base di queste ultime
  - la rappresentazione di un oggetto di una classe derivata è composto da tutti i dati membro (e metodi) presenti nelle classi del suo reticolo genealogico.

Specificazione di accesso

- All'esterno di una classe, protected equivale a private (i dati e metodi private/protected non sono accessibili)
- La classe derivata può accedere solo ai dati membro e ai metodi ereditati che non sono privati. L'accessibilità di un membro dipende sia dal suo indicatore di accesso che dal tipo di derivazione

`class Base`	`class Derivata : public Base`	`class Derivata : protected Base`	`class Derivata: private Base`
`public`: Accessibili dalle funzioni membro e dall'esterno	I membri public di Base sono public della Derivata	I membri public di Base sono protected della Derivata	I membri public di Base sono private della Derivata
`protected`: Accessibili solo dalle funzioni membro	I membri protected di Base sono protected della Derivata	I membri protected di Base sono protected della Derivata	I membri protected di Base sono private della Derivata
`private`: Accessibili solo dalle funzioni membro	I membri private di Base inaccessibili nella Derivata	I membri provate di Base sono inaccessibili nella Derivata	I membri private di Base sono inaccessibili nella Derivata

- Nella maggior parte dei casi si usa derivazione "publica".
- Tentazione di dichiarare protected le variabli che sarebbero naturalmente "private" nelle classi base $\rightarrow$ parziale violazione del principio di "encapsulamento": usare solo se necessario.

Costruttori nelle classi derivate

- Prima di C++11 la classe derivata doveva implementare esplicitamente i costruttori, in C++11 è possibile "ereditare" tutti i costruttori della classe base:

```
1    class Base{
2      public:
3        Base();
4      ...
5    }
6    class Derivata: public Base{
7      public:
8        using Base::Base;
9      ...
10   }
```

- Sempre in C++11 la costruzione per "delega" estende il meccanismo dei parametri default (anche i casi in cui non è banale definire un default)

```
1    Point3d(int x, int y, int z):m_x(x),m_y(y),m_z(z){}
2    Point3d(Point2d p):Point3d(p.x, p.y, 0){}
```

Metodi nelle classi derivate

- La classe derivata può fare l'overloading dei metodi non private che ha ereditato
  - Cioè può specializzare il metodo secondo necessità
  - Ovviamente, se non c'è overload, viene chiamato il metodo della classe base

Composizione o Ereditarietà ?

Ereditarietà (**is-a**):

```
1 class B {
2 };
3 class A : public B {
4 };
```

Composizione (**has-a**):

```
1 class B {
2 };
3 class A {
4   B b;
5 };
```

Composizione o Ereditarietà ?

Come scegliere:

- a senso (has-a o is-a ?)
- rispettare il criterio di Liskov:
  - *ogni metodo della classe base deve potere agire su un oggetto della classe derivata senza ulteriori informazioni.*

Possibili scelte di derivazione:

- specializzazione: creare versioni specilizzate delle classi.
- estensione: fornire ad una classe dati o funzionalità aggiuntive.

Accesso ai dati membri: composizione o ereditarietà

```
1  class B {
2    public:
3      double Var(){return m_var;}
4    protected:
5      double m_var;
6  };
7
8  class A : public B { // Ereditarieta'
9    public:
10     void Print(){
11       return m_var;
12     }
13 };
14
15 class A {              // Composizione
16   public:
17     void Print(){
18       return m_b.Var();
19     }
20   private:
21     B m_b;
22 };
```

Composizione o Ereditarietà ?

Proviamo ad immaginare come sfruttare composizione o ereditarietà per risolvere questi due problemi

- Classe triangolo (figura piana nel piano 2D)
  - Immaginiamo di partire da una classe punto (2d) e da una classe poligono
- Vettore generico n-dimensionale con somma e moltiplicazione per scalare
  - Partiamo dalla classe `vector`

Classe Poligono

Definiamo la classe poligono, composto da punti.

Classe Poligono

Definiamo la classe poligono, composto da punti.

```
1  class Punto{
2  public:
3    Punto(double x=0, double y=0):m_x(x),m_y(y){}
4    double X(){return m_x;}
5    double Y(){return m_y;}
6  private:
7    double m_x,m_y;
8  };
9  class Poligono{
10 public:
11   Poligono(const vector<punto>& punti):m_punti(punti){};
12   Poligono():Poligono(vector<punto>(0)){}
13   void print();
14 protected:
15   vector <punto> m_punti;
16 };
17 void Poligono::Print(){
18   for (auto p: m_punti)
19     cout << p.X() << " " << p.Y() << endl;
20 }
```

Classe triangolo

Creiamo una classe specializzata triangolo che abbia, ad esempio, anche il costruttore che prende due lati e l'angolo compreso tra di essi.

Classe triangolo

Creiamo una classe specializzata triangolo che abbia, ad esempio, anche il costruttore che prende due lati e l'angolo compreso tra di essi.

```
#include <vector>
#include <iostream>
#include <cmath>
using namespace std;
#include "Lez3_3.h"
class Triangolo: public Poligono{
public:
  using Poligono::Poligono;
  Triangolo(double a, double b, double theta):poligono(){
    m_punti.push_back(Punto(0,0));
    m_punti.push_back(Punto(a,0));
    m_punti.push_back(Punto(b*cos(theta),b*sin(theta)));
  }
};
int main(){
  Triangolo t(1,1,M_PI/2);
  t.Print();
  return 0;
}
```

Classe vettore n-dimensionale

Costruiamo una classe che erediti da `vector` aggiungendo il calcolo vettoriale

Classe vettore n-dimensionale

Costruiamo una classe che erediti da `vector` aggiungendo il calcolo vettoriale

```
1  #include <vector>
2  using namespace std;
3
4  template <class T>
5  class vectorn: public vector<T>{
6   public:
7    using vector<T>::vector;
8    vectorn<T> operator+(const vectorn&)const;
9    vectorn<T> operator*(double)        const;
10 };
```

Classe vettore n-dimensionale

```
1  #include "Lez3_4c.h"
2  #include <iostream>
3
4  template <class T>
5  vectorn<T> vectorn<T>::operator+(const vectorn<T>& b) const{
6    vectorn res(this->size());
7    for (int i=0;i<this->size();i++)
8      res.at(i) = this->at(i) + b.at(i);
9    return res;
10 }
11
12 template <class T>
13 vectorn<T> vectorn<T>::operator*(double f) const{
14   vectorn res(this->size());
15   for (int i=0;i<this->size();i++)
16     res.at(i) = this->at(i)*f;
17   return res;
18 }
19
20 int main(){
21   vectorn<double> v1(4,1);
22   vectorn<double> v2(4,3);
23   vectorn<double> v3(4);
```

Design

- Nello sviluppo di qualsiasi programma è sempre consigliabile partire da un design prestabilito che
  - consideri tutti gli aspetti del problema
  - tenga conto a priori delle possibili evoluzioni del programma: in futuro nuove requisiti potrebbero richiedere sostanziali modifiche del codice
- L'assenza di una fase di design porta al cosiddetto "spaghetti code"
  - la struttura del codice è talmente intrecciata da impedire ulteriori sviluppi
  - nuovi requisiti richiedono una completa riscrittura del programma
- Già nei linguaggi strutturati l'utilizzo delle funzioni comporta una fase di design e organizzazione del flusso del programma
  - Lo sforzo è ripagato dalla maggior leggibilità e gestibilità del codice
  - Ma rimane a discrezione dell'utente
- Nella programmazione OO il design diventa a tutti gli effetti obbligatoria
  - il programmatore è obbligato ad analizzare il problema in termini di classi e relazioni tra di esse; deve in pratica creare diagrammi UML.
  - Il design diventa la fase principale e richiede il 90% dello sforzo.

Progetto OO C++ per la prima esercitazione

L'obiettivo di questo progetto è descrivere in formalismo OO un punto materiale e poi utilizzarlo nella soluzione di equazioni del moto di un sistema gravitazionale.
Lista della spesa:

- Classe `Particle` (oggetto con massa e carica)
- Classe `MatPoint` (Particella con posizione e velocità)
- Aggiungere metodo `GravField` che calcola il campo gravitazionale prodotto da una particella (o corpo) massivo.

Per la rappresentazione dei vettori 3D usare la classe costruita nelle precedenti lezioni (`Vector3`).

Polimorfismo “run-time”

- Al contrario del polimorfismo “compile-time”, il metodo da chiamare viene deciso ”run-time”, quando l’oggetto è disponibile. È chiamato run-time, late o dynamic binding.
- Si realizza con la keyword `virtual`

Polimorfismo "run-time"

- I metodi di una classe base possono essere dichiarati virtual

```
virtual retType funcName(parType);
```

- Una classe che eredita una funzione virtual può ridefinirla (se non lo fa verrà usata la funzione della classe base)
- Tuttavia ridefinire un metodo virtual è diverso dalla pura ridefinizione del metodo (senza la keyword virtual). Se viene chiamato il metodo su puntatore della classe base che punta ad un oggetto della classe derivata, il metodo agisce su quest'ultima (e non della classe base come succede senza virtual).
- Si introduce così la possibilità di rinviare all'esecuzione la decisione di che metodo usare.

Polimorfismo "run-time"

```
1  #include <iostream>
2  class Animale{
3   public:
4    Animale(){}
5    virtual void parla(){ std :: cout << "???" << std :: endl ;}
6  };
7  class Cane: public Animale{
8   public:
9    using Animale::Animale;
10   void parla(){ std::cout << "Bau" << std::endl;}
11 };
12 class Gatto: public Animale{
13  public:
14   using Animale::Animale;
15   void parla (){ std :: cout << "Miao" << std :: endl ;}
16 };
17 class Uomo: public Animale{
18  public:
19   using Animale::Animale;
20   void parla (){ std :: cout << "Ciao" << std :: endl ;}
21 };
```

Polimorfismo "run-time"

```
1  #include <iostream>
2  #include <string>
3  #include "Lez3_5.h"
4  using namespace std;
5  int main(){
6
7    cout << "Inserisci il nome dell'animale" << endl;
8    string opt; cin >> opt;
9    Animale *p;
10   if (opt=="Cane")
11     p = new Cane;
12   else if (opt=="Gatto")
13     p = new Gatto;
14   else if (opt=="Uomo")
15     p = new Uomo;
16
17   // da qui in poi il codice e' implementato con
18   // il puntatore della classe base
19
20   p->parla();
21
22   return 0;
23 }
```

Metodi puramente virtuali

```
1 class Name {
2       public:
3               virtual returntype name(parameters) = 0;
4               ...
5 };
```

- Metodo puramente virtuale: pure virtual method.
- Se una classe ha un metodo puramente virtuale nessun oggetto può essere costruito con quella classe. La classe si dice **astratta**
- Il metodo è dichiarato ponendolo uguale a **0**
- Deve essere implementato dalle classi derivate
- L'utilità sta nel definire un'interfaccia che deve essere rispettata da tutti gli utenti della classe anche in assenza di implementazione esplicita (molto utile per sviluppo parallelo di codici).

Interfacce

```
1 class Poligono{
2  public:
3     ...
4     virtual double Area() = 0;
5     ...
6 }
```

- Costituisce un interfaccia perché obbliga tutte la classi derivate a definire il metodo `Area` con lo stesso prototipo.
- Nessuno poligono "generico" può essere creato.