Overhead del late binding

Overhead del late binding

Article Talk

Virtual method table

From Wikipedia, the free encyclopedia

Efficiency [edit]

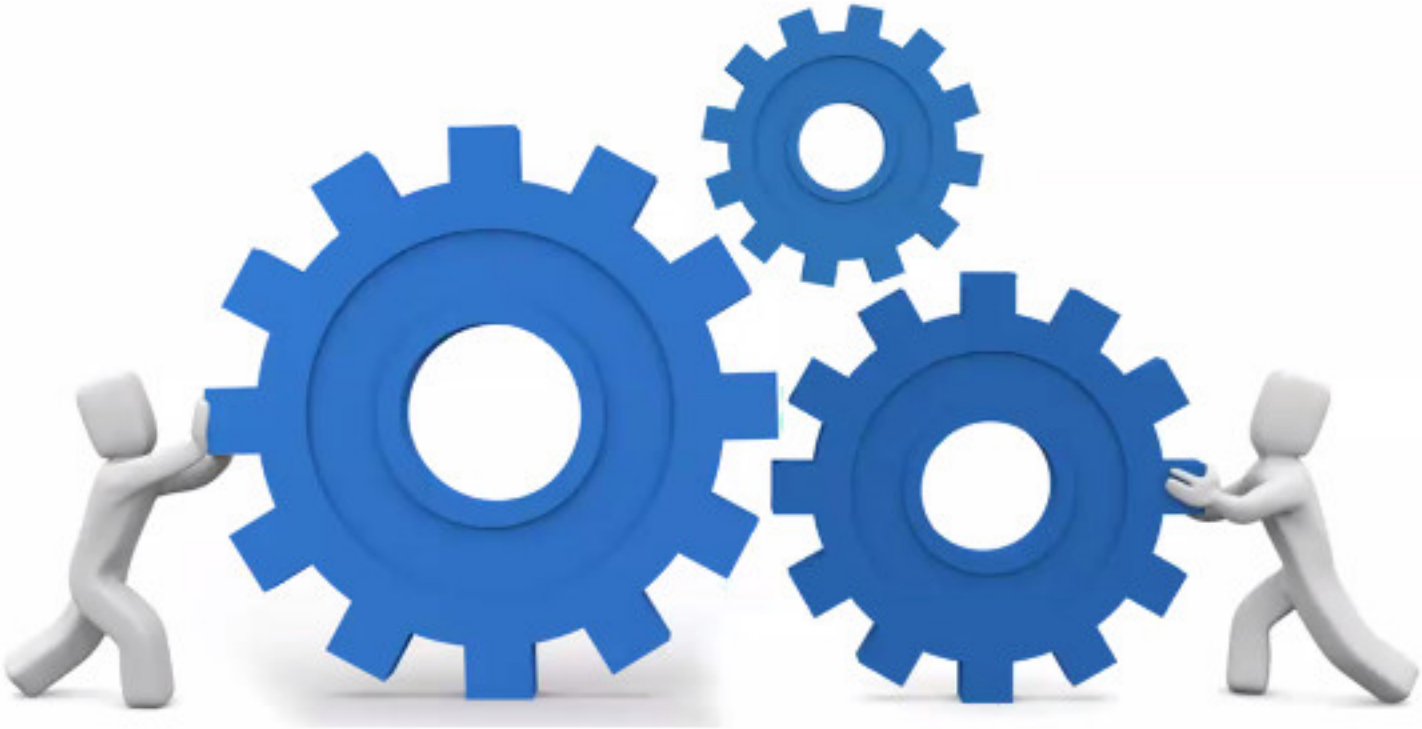
A virtual call requires at least an extra indexed dereference, and sometimes a "fixup" addition, compared to a non-virtual call, which is simply a jump to a compiled-in pointer. Therefore, calling virtual functions is inherently slower than calling non-virtual functions. An experiment done in 1996 indicates that approximately 6–13% of execution time is spent simply dispatching to the correct function, though the overhead can be as high as 50%.[4] The cost of virtual functions may not be so high on modern CPU architectures due to much larger caches and better branch prediction.

Esperimento

```
class B {
  public:
   virtual void f() {}
};

int main() {
  B* p = new B;
  long int i;
  // dynamic binding
  for (i=0; i<5000000000; ++i) p->f();      // 8.53 sec (~ +18%)
  // static binding
  for (i=0; i<5000000000; ++i) p->B::f();   // 6.99 sec
}

% time ./a.out  // MacOS2019, MacBookPro 2019, clang
```

Su MacOS2018, MacBookPro 2013, con clang eravamo a ~ +4%. Qualche anno fa con g++ eravamo sul ~ +15%

Implementazione del late binding

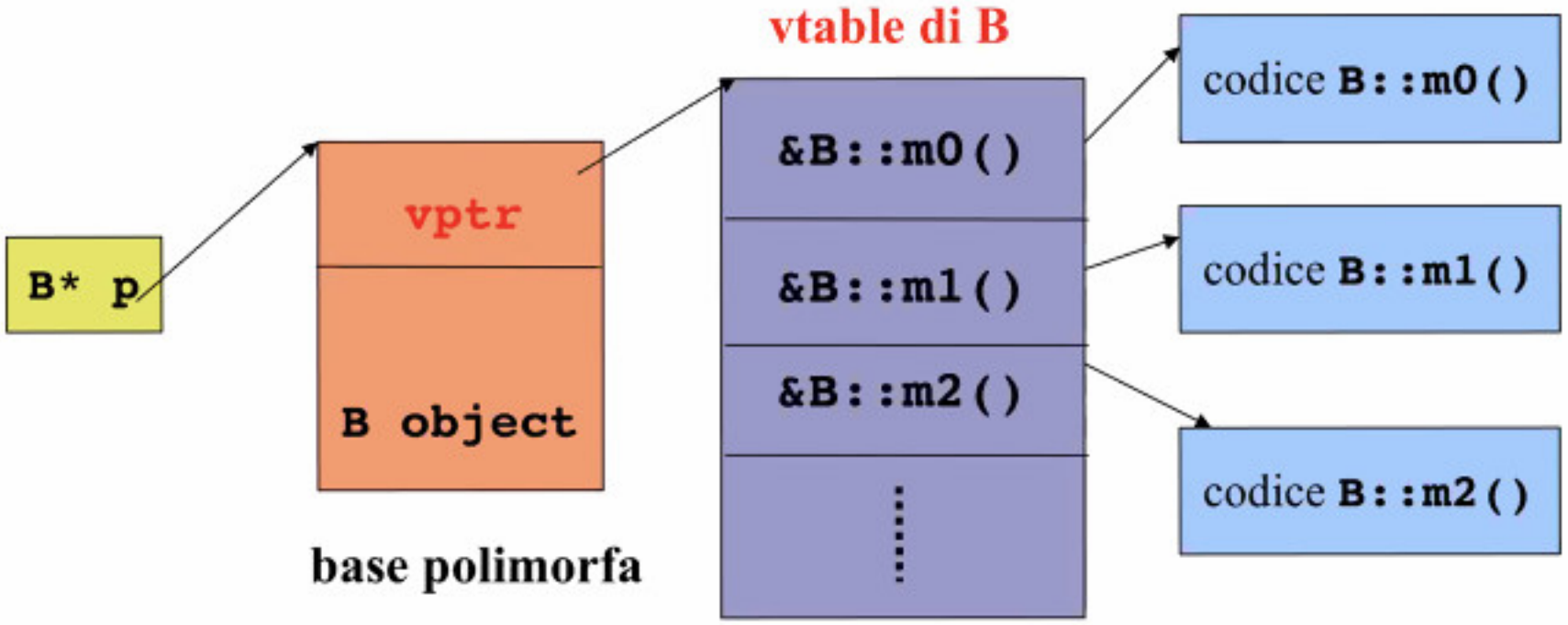


Virtual method table

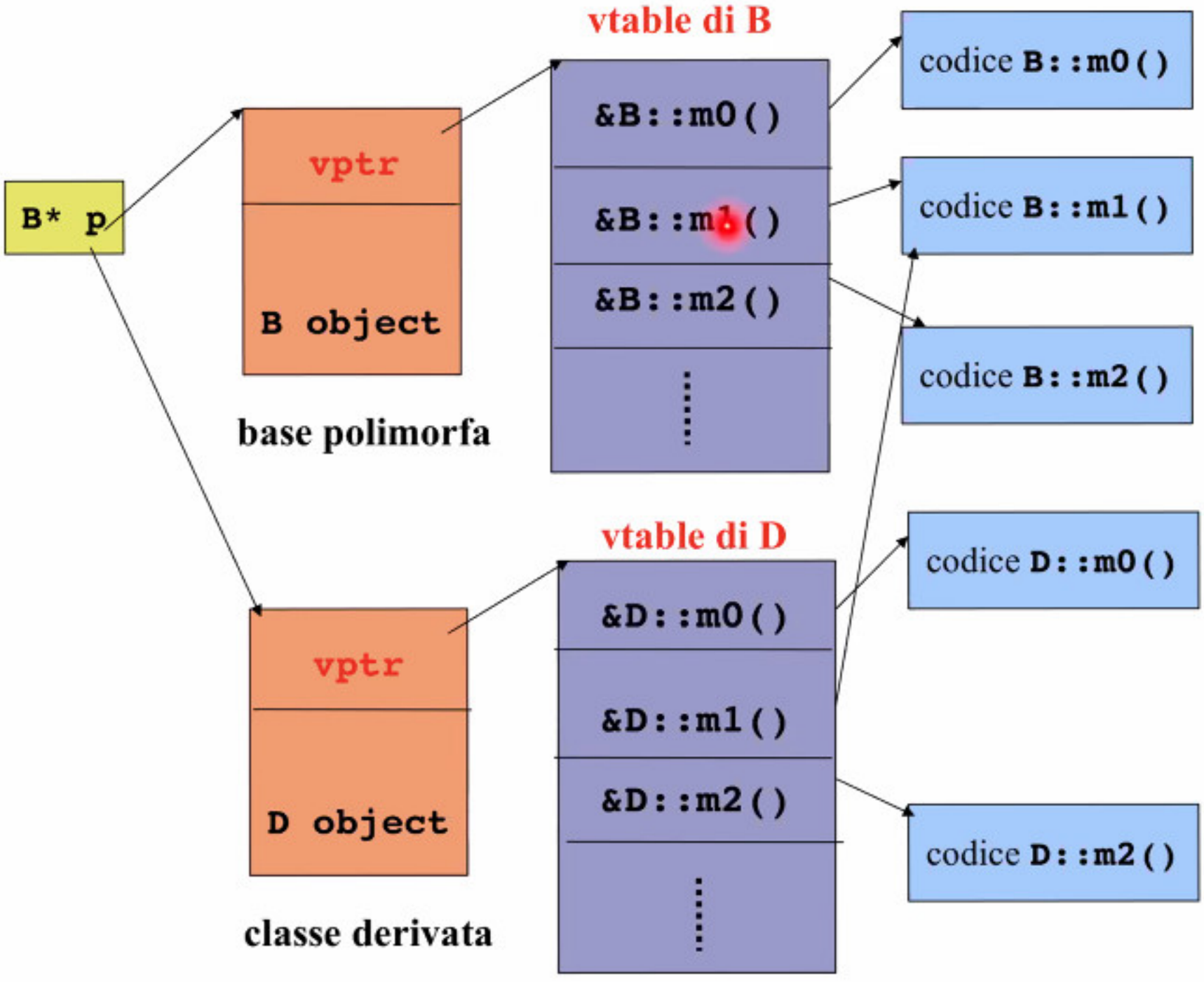
From Wikipedia, the free encyclopedia

A **virtual method table** (**VMT**), **virtual function table**, **virtual call table**, dispatch table, **vtable**, or **vftable** is a mechanism used in a programming language to support dynamic dispatch (or run-time method binding).

Whenever a class defines a virtual function (or method), most compilers add a hidden member variable to the class which points to an array of pointers to (virtual) functions called the virtual method table (VMT or Vtable). These pointers are used at runtime to invoke the appropriate function implementations, because at compile time it may not yet be known if the base function is to be called or a derived one implemented by a class that inherits from the base class.



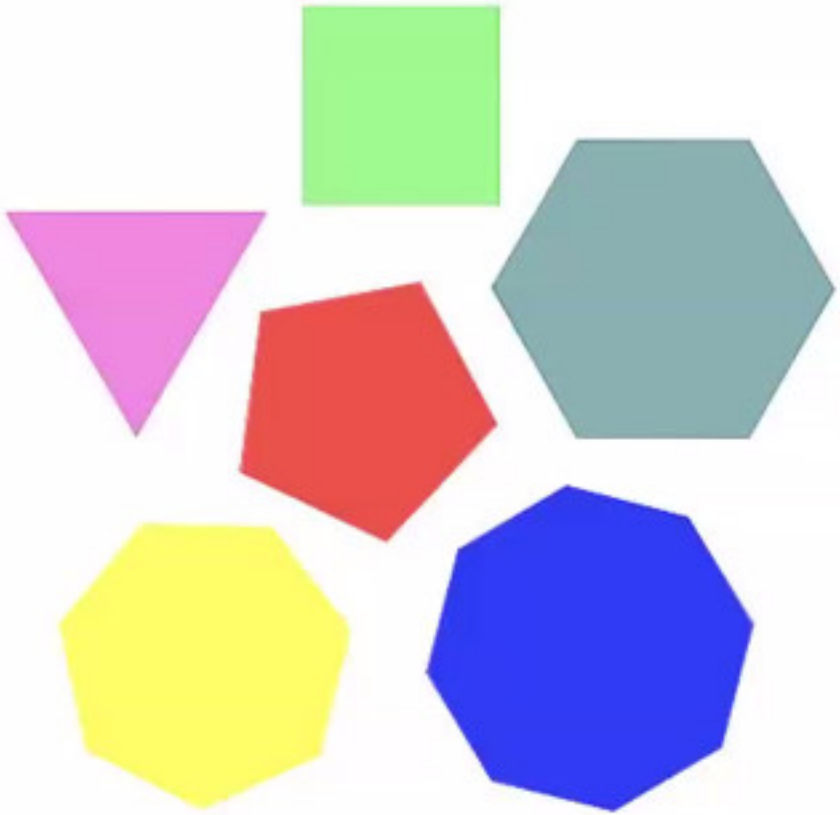
vptr *virtual pointer*


vtable di B
codice B::m0()
&B::m0()
B* p
vptr
codice B::m1()
&B::m1()
B object
&B::m2()
codice B::m2()
base polimorfa
vtable di D
codice D::m0()
&D::m0()
vptr
&D::m1()
&D::m2()
codice D::m2()
D object
classe derivata

```
class B {
public:
  FunctionPointer* vptr; // vpointer aggiunto dal compilatore
  virtual void m0() {}
  virtual void m1() {}
  virtual void m2() {}
};

class D: public B {
public:
  virtual void m0() {} // overriding
  virtual void m2() {} // overriding
};
```

```
B* p;

p=&b;
p->m2();
// chiama la funzione all'indirizzo *((p->vptr)+2)
// cioè chiama B::m2()

p=&d;
p->m2();
// chiama la funzione all'indirizzo *((p->vptr)+2)
// cioè chiama D::m2()
```

Back to poligono



```
// file polv.h
#ifndef POLV_H
#define POLV_H

class punto {
 private:
  double x, y;
 public:
  punto(double a=0, double b=0): x(a), y(b) {}
  static double lung(const punto& p1, const punto& p2);
};

class poligono { // classe polimorfa
 protected:
  int nvertici;
  punto* pp;
 public:
  poligono(int n, const punto v[]);
  ~poligono();
  poligono(const poligono& pol);
  poligono& operator=(const poligono& pol);
  // contratto: ritorno il perimetro del poligono di invocazione
  virtual double perimetro() const; // metodo virtuale
};
#endif
```

```
// file triv.h
#ifndef TRIV_H
#define TRIV_H
#include "polv.h"

class triangolo: public poligono {
 public:
  triangolo(const punto v[]);
  // eredita perimetro()
  // contratto: ritorno l'area del triangolo di invocazione
  virtual double area() const; // nuovo metodo virtuale
};
#endif

// file triv.cpp
#include <iostream>
#include <math.h>
#include "triv.h"

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

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

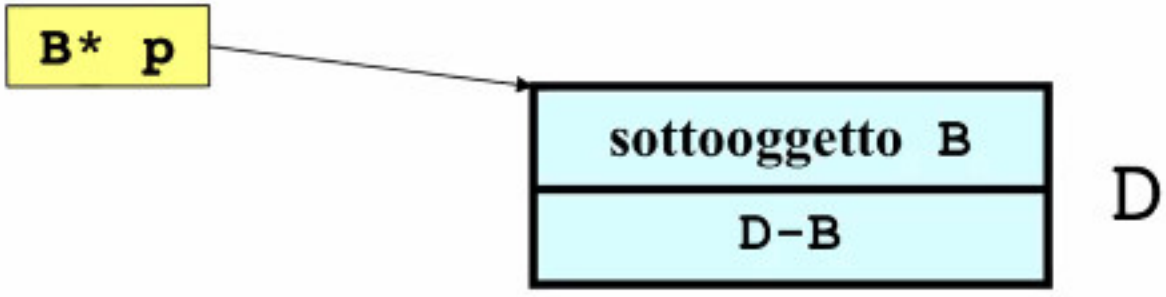
```
// Contratto: stampa il perimetro del poligono *p
void stampa_perimetro(poligono* p) {
  cout << "Il perimetro è " << p->perimetro() << endl; // chiamata polimorfa
}

// Contratto: stampa l'area del triangolo t
void stampa_area_triangolo(const triangolo& t) {
  cout << "L'area è " << t.area() << endl; // chiamata polimorfa
}

int main() {
  int i; punto vv[4]; double x,y;
  cout << "Scrivi le coordinate di 4 vertici:\n";
  for (i=0; i<4; i++) { cin >> x >> y; v[i]=punto(x,y); }
  poligono po(4, vv);
  cout << "Scrivi le coordinate di 3 vertici:\n";
  for (i=0; i<3; i++) { cin >> x >> y; v[i]=punto(x,y); }
  triangolo tr(vv) ;
  cout << "Scrivi le coordinate di 3 vertici, con angolo retto sul primo:\n";
  for (i=0; i<3; i++) { cin >> x >> y; v[i]=punto(x,y); }
  tri_rettangolo rr(vv) ;

  cout << "Poligono:\n";
  stampa_perimetro(&po);
  cout << "\nTriangolo:\n";
  stampa_perimetro(&tr); stampa_area_triangolo(tr);
  cout << "\nTriangolo rettangolo:\n";
  stampa_perimetro(&rr); stampa_area_triangolo(rr);
}
```

Sia `D` una classe derivata da `B`. Consideriamo la seguente situazione:

```
D* pd = new D;
B* pb = pd; // pb ha tipo dinamico D*
delete pb;
```

Sia `D` una classe derivata da `B`. Consideriamo la seguente situazione:

```
D* pd = new D;
B* pb = pd; // pb ha tipo dinamico D*
delete pb;
```

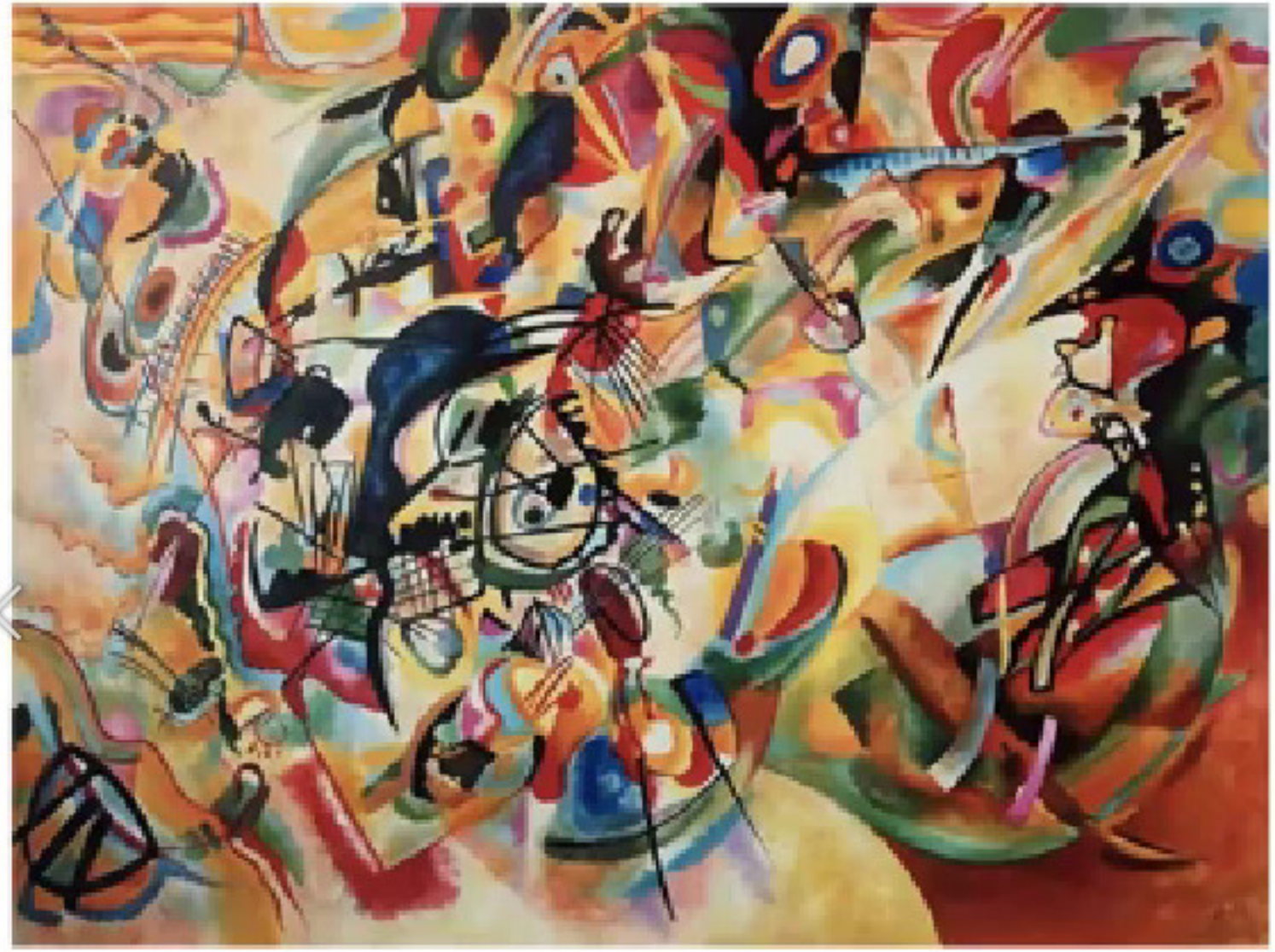


Distruttori virtuali

Esempio

```
class B {
private:
  int* p;
public:
  B(int n, int v) : p(new int[n]) {
    for(int i=0; i<n; i++) p[i]=v;
  };
  virtual ~B() { delete[] p; cout <<"~B() "; }; // distruttore virtuale
};

class C : public B {
private:
  int* q;
public:
  C(int sizeB, int sizeC, int v) : B(sizeB, v), q(new int[sizeC]) {
    for(int i=0; i<sizeC; i++) q[i]=v;
  };
  virtual ~C() {delete[] q; cout <<"~C() ";};
};

int main() {
  C* q = new C(4,2,18);
  B* p=q; // puntatore polimorfo
  delete p; // distruzione virtuale: invoca ~C()
}
// stampa: ~C() ~B()
// se ~B() non fosse virtuale verrebbe invocato solamente ~B() !
```

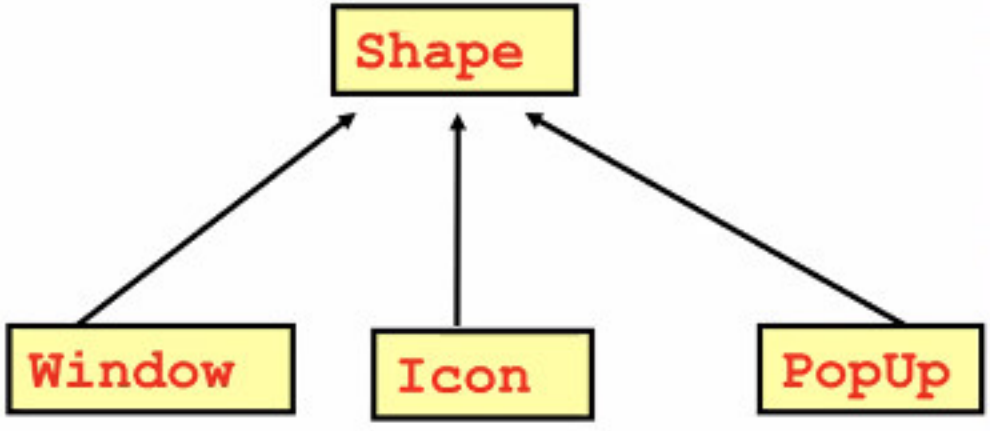
KANDINSKY



Classi base astratte

```
class Shape {
   ...
   virtual void draw(Position){...}
};
class Window: public Shape {
   ...
   void draw(Position) {...}
};
class Icon: public Shape {
   ...
   void draw(Position) {...}
};
class PopUp: public Shape {
   ...
   void draw(Position) {...}
};

class DesktopManager {
   ...
   void show(const Shape& s) {
      ...
      s.draw(computePosition());
      ...
   }
};
```

Quale implementazione?



Abstract and concrete [edit]

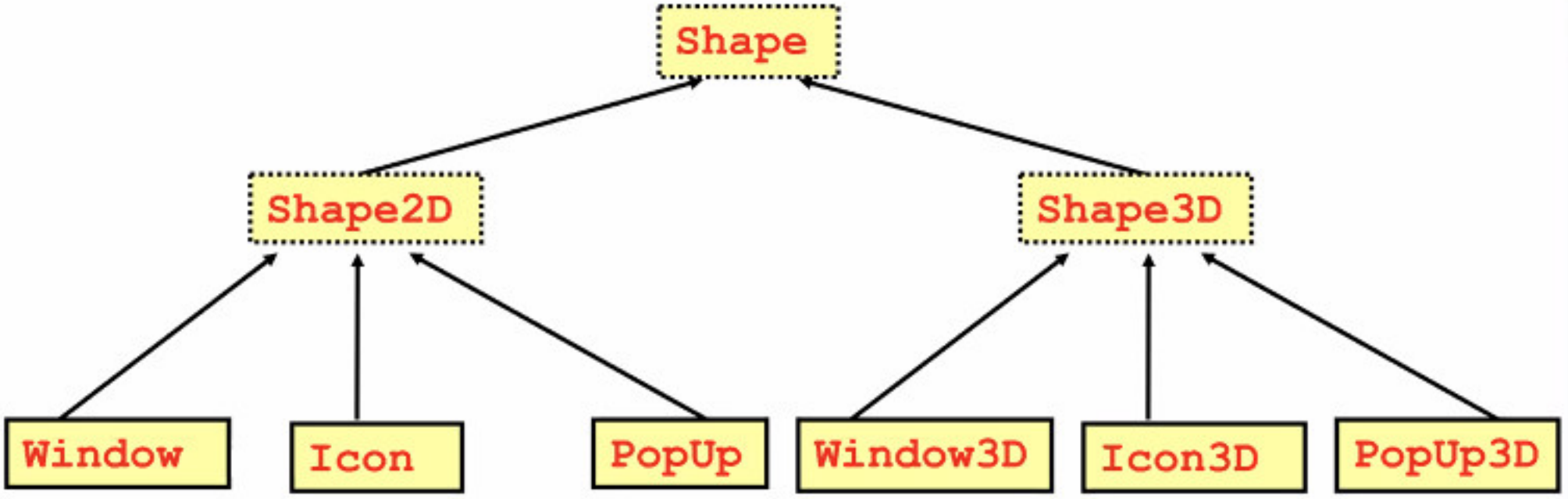
Main article: Abstract type

In a language that supports inheritance, an **abstract class**, or *abstract base class* (ABC), is a class that cannot be instantiated because it is either labeled as abstract or it simply specifies abstract methods (or *virtual methods*). An abstract class may provide implementations of some methods, and may also specify virtual methods via signatures that are to be implemented by direct or indirect descendants of the abstract class. Before a class derived from an abstract class can be instantiated, all abstract methods of its parent classes must be implemented by some class in the derivation chain.[25]

Most object-oriented programming languages allow the programmer to specify which classes are considered abstract and will not allow these to be instantiated. For example, in Java and PHP, the keyword *abstract* is used.[26][27] In C++, an abstract class is a class having at least one abstract method given by the appropriate syntax in that language (a pure virtual function in C++ parlance).[25]

A class consisting of only virtual methods is called a Pure Abstract Base Class (or *Pure ABC*) in C++ and is also known as an *interface* by users of the language.[13] Other languages, notably Java and C#, support a variant of abstract classes called an interface via a keyword in the language. In these languages, multiple inheritance is not allowed, but a class can implement multiple interfaces. Such a class can only contain abstract publicly accessible methods.[19][28][29]

A **concrete class** is a class that can be instantiated, as opposed to abstract classes, which cannot.



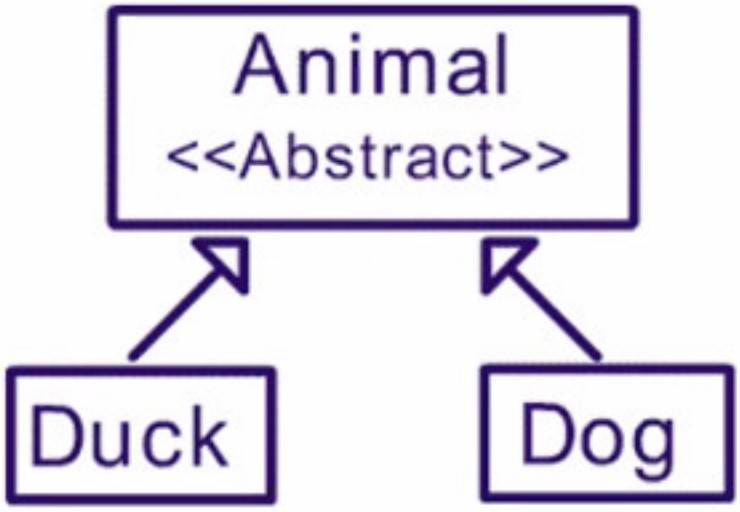
Classi base astratte

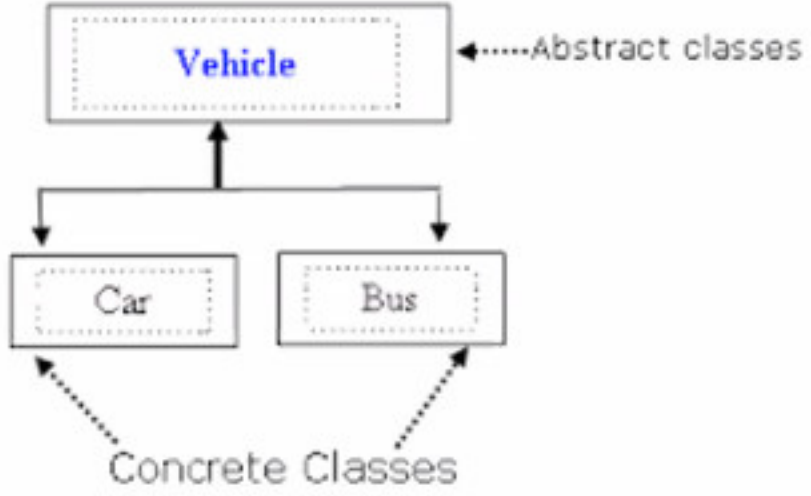
(1) almeno un metodo virtuale puro

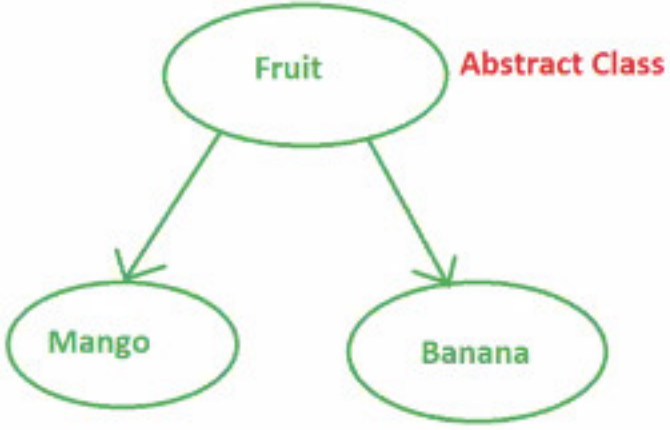
(2) non si possono costruire oggetti

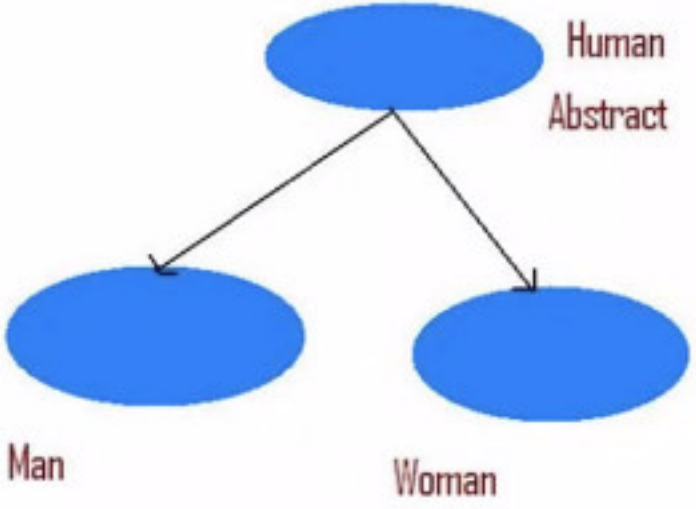
```
class B { // classe base astratta
public:
  virtual void f() = 0;
};

class C: public B { // sottoclasse astratta
public:
  void g() {cout << "C::g() ";}
};

class D: public B { // sottoclasse concreta
public:
  virtual void f() {cout << "D::f() ";}
};

int main() {
  // C c; // Illegale: "cannot declare c of type C ..."
  D d;     // OK, D è concreta
  B* p;    // OK, puntatore a classe astratta
  p = &d; // puntatore (super)polimorfo
  p->f(); // stampa: D::f()
}
```

Examples


Animal
<<Abstract>>
Duck
Dog


Vehicle
Abstract classes
Car
Bus
Concrete Classes


Fruit
Abstract Class
Mango
Banana

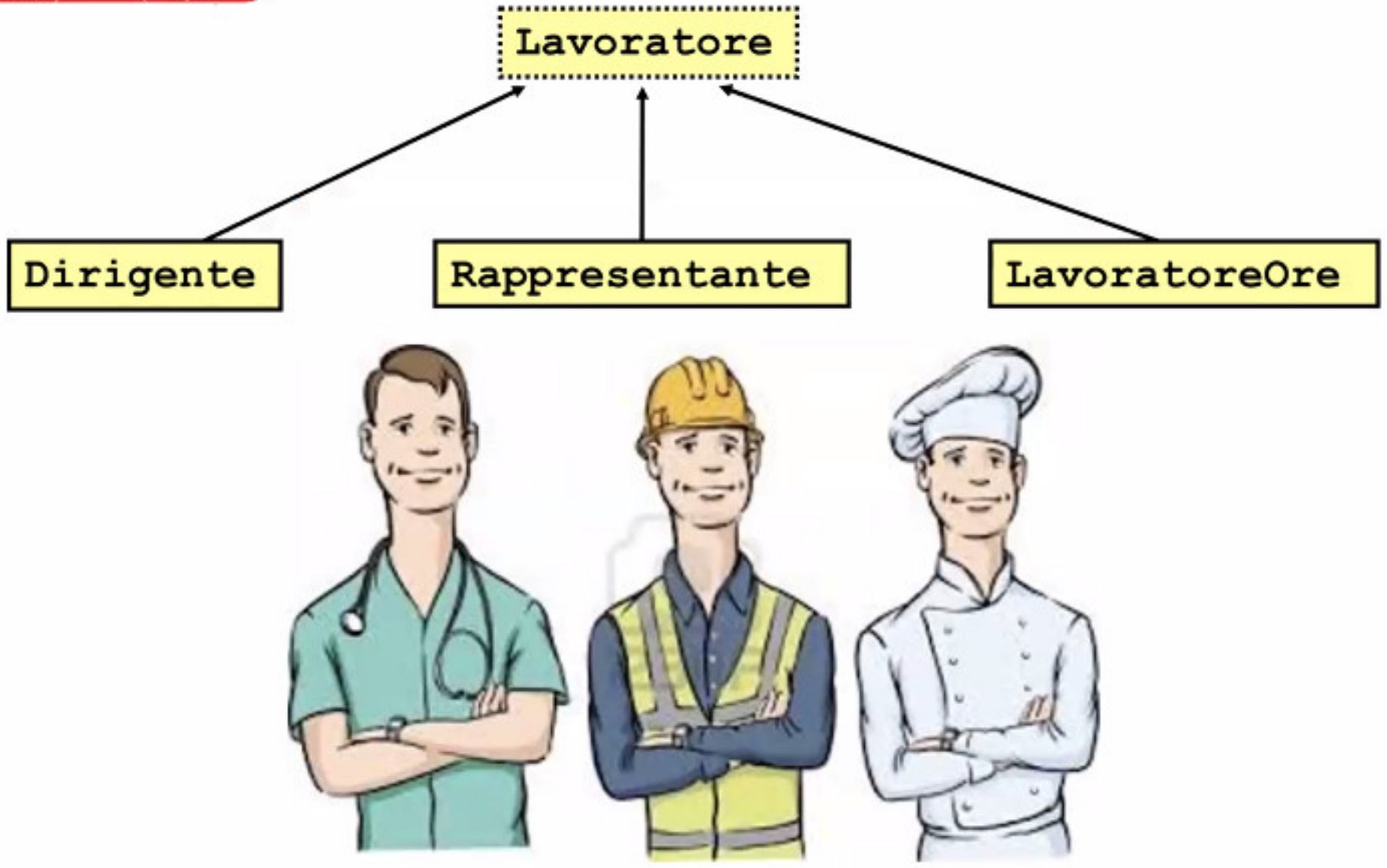

Human
Abstract
Man
Woman


```
class poligono { // classe astratta
 protected:
  int nvertici;
  punto* pp;
 public:
  poligono(int n, const punto v[]);
  ~poligono();
  poligono(const poligono& pol);
  poligono& operator=(const poligono& pol);
  // contratto: ritorno il perimetro del poligono di invocazione
  virtual double perimetro() const; // metodo virtuale
  virtual double area() const =0;    // metodo virtuale puro
};
#endif
```

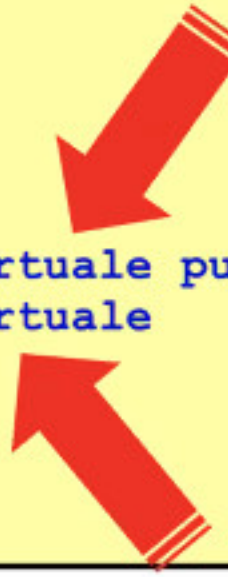
Distruttore virtuale puro

```
class Base {
// classe astratta
 public:
  virtual ~Base() =0;
  // distruttore virtuale puro
};
Base::~Base() {} // definizione comunque obbligatoria!

int main() {
Base b;               // "cannot declare b of abstract type Base"
Base* p = new Base; // "cannot allocate an object of abstract type Base"
}
```

EXAMPLE



```
class Lavoratore { // classe base astratta
public:
  virtual ~Lavoratore() {}
  Lavoratore(string s): nome(s) {};
  string getNome() const {return nome;};
  virtual double stipendio() const = 0;              // virtuale pura
  virtual void printInfo() const {cout << nome;}; // virtuale
private:
  string nome;
};
```



```
class Dirigente : public Lavoratore {
private:
  double fissoMensile; // stipendio fisso
public:
  Dirigente(string s, double d = 0):
                                  Lavoratore(s), fissoMensile(d) {}

  virtual double stipendio() const { // implementazione
    return fissoMensile;
  }

  virtual void printInfo() const { // overriding
    cout << "Dirigente ";
    Lavoratore::printInfo(); // invocazione statica
  }
};
```

```
class Rappresentante : public Lavoratore {
private:
  double baseMensile; // stipendio base fisso
  double commissione; // commissione per pezzo venduto
  int tot;            // pezzi venduti in un mese
public:
  Rappresentante(string s, double d=0, double e=0, int x=0):
    Lavoratore(s), baseMensile(d), commissione(e), tot(x) {}

  virtual double stipendio() const { // implementazione
    return baseMensile + commissione*tot;
  }
  virtual void printInfo() const { // overriding
    cout << "Rappresentante "; Lavoratore::printInfo();
  };
};
```

```
class LavoratoreOre : public Lavoratore {
private:
  double pagaOraria;
  double oreLavorate;  // ore lavorate nel mese
public:
  LavoratoreOre(string s, double d=0, double e=0):
    Lavoratore(s), pagaOraria(d), oreLavorate(e) {}

  virtual double stipendio() const { // implementazione
    if ( oreLavorate <= 160 ) // nessuno straordinario
      return pagaOraria*oreLavorate;
    else  // le ore straordinarie sono pagate il doppio
      return 160*pagaOraria+(oreLavorate-160)*2*pagaOraria;
  };
  virtual void printInfo() const { // overriding
    cout << "Lavoratore a ore "; Lavoratore::printInfo();
  };
};
```

```
// funzione esterna
void stampaStipendio(Const Lavoratore& x) {
   x.printInfo(); // chiamata polimorfa
   cout << " in questo mese ha guadagnato "
 << x.stipendio() <<" Euro.\n"; // chiamata (super)polimorfa
}

int main() {
   Dirigente d("Pippo", 4000);
   Rappresentante r("Topolino",1000,3,250);
   LavoratoreOre l("Pluto",15,170);
   stampaStipendio(d);
   stampaStipendio(r);
   stampaStipendio(l);
}
```

```
// STAMPA:
Dirigente Pippo in questo mese ha guadagnato 4000 Euro.
Rappresentante Topolino in questo mese ha guadagnato 1750 Euro.
Lavoratore a ore Pluto in questo mese ha guadagnato 2700 Euro.
```