Derivarea claselor (continuare)

- În cazul moştenirii unei clase, furnizarea unei noi definiţii pentru una din funcţiile membru moştenite din clasa de bază se poate încadra în una din următoarele două situaţii :
    - se furnizează în clasa derivată aceeaşi semnătură şi acelaşi tip returnat, situaţie în care funcţia este **redefinită (redefined)** în cazul funcţiilor obişnuite şi, respectiv, **supradefintă (overriden)** în cazul funcţiilor virtuale;
    - se modifică lista de parametri ai funcţiei sau tipul returnat, situaţie în care funcţia din clasa de bază este **ascuns**ă **(hidden)** şi deci inaccesibilă.
- Deşi cea de a doua modalitate de operare nu este greşită, ea nu se încadrează în spiritul procesului de derivare a claselor, care exploatează în principal polimorfismul.
- În general, **ori de cate ori se redefineşte o funcţie supraîncărcată din clasa de bază, toate celelalte versiuni devin indisponibile**.

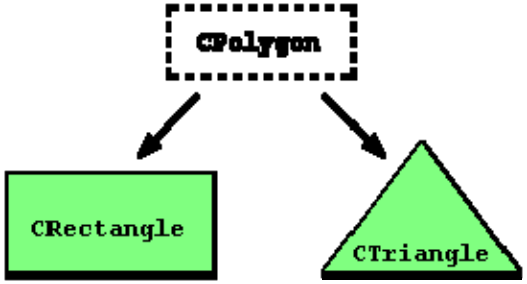
Exemplul 1:

```
#include <iostream>
#include <string>
using namespace std;

class Base {
public:
  int f() const {
    cout << "Base::f()\n";
    return 1;
  }
  int f(string) const { return 1; }
  void g() {}
};

class Derived1 : public Base {
public:
  void g() const {}
};

class Derived2 : public Base {
public:
  // Redefinire:
  int f() const {
    cout << "Derived2::f()\n";
    return 2;
  }
};

class Derived3 : public Base {
public:
//Se schimba tipul returnat: ASCUNDERE!
  void f() const { cout <<
                  "Derived3::f()\n"; }
};

class Derived4 : public Base {
public:
  // Se schimba lista de parametri:
  // ASCUNDERE!
  int f(int) const {
    cout << "Derived4::f()\n";
    return 4;
  }
};

int main() {
  string s("hello");
  Derived1 d1;
  int x = d1.f();
  d1.f(s);
  Derived2 d2;
  x = d2.f();
//d2.f(s);// NU: versiunea string e ascunsa
  Derived3 d3;
  d3.f();
//x = d3.f();// NU: versiunea cu return int
              // e ascunsa
  Derived4 d4;
//x = d4.f(); // versiunea f()e ascunsa
  x = d4.f(1);
}
```

REZULTATE
Base::f()
Derived2::f()
Derived3::f()
Derived4::f()

Polimorfism în C++

- Termenul de polimorfism provine din limba greacă (poly + morphos) şi înseamnă, ad litteram, capacitatea unei entităţi de a avea mai multe forme
- Pentru a ilustra conceptul de polimorfism se consideră exemplul următor în care se derivează public din clasa de bază CPolygon clasele CRectangle şi CTriangle



Exemplul 2.a)

```
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
  };

class CRectangle: public CPolygon {
  public:
    double area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    double area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

- Mecanismul polimorfic se implementează în C++ pe baza proprietăţii pointerilor la o clasă derivată de a fi compatibili ca tip cu pointerii la clasa de bază din care s-a făcut derivarea. Cu alte cuvinte, **unui pointer la clasa de bază i se poate atribui valoarea unui pointer la o clasă derivată din clasa respectivă** (evident în cazul accesului la datele membru, acesta va fi posibil doar pentru cele moştenite din clasa de bază).
- Inversul situaţiei nu este implicit valabil; aceasta înseamnă că unei variabile de tip pointer, care pointează către o clasa derivată, nu i se poate atribui ca valoare un pointer către clasa de bază aferentă. Atribuirea este posibilă doar dacă se fac conversii explicite de tip.

Exemplul de mai sus se poate rescrie:

Exemplul 2.b)

```
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
  };

class CRectangle: public CPolygon {
  public:
    double area ()
      { return (width * height); }
  };


class CTriangle: public CPolygon {
  public:
    double area ()
      { return (width * height / 2); }
  };
```

```
int main ()
{
  CRectangle rect;
  CTriangle trgl;
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;

  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);

 // ppoly1-> area();// ERONAT
 // ppoly2-> area();// ERONAT

  cout << rect.area() << endl;
  cout << trgl.area() << endl;
  return 0;
}
```

- Semnificaţia polimorfismului în C++ este aceea de a furniza o singură interfaţă prin intermediul căreia să fie disponibile mai multe metode
- **Polimorfismul se implementează în C++ prin intermediul funcţiilor virtuale**
- O funcţie membru a unei clase, care poate fi definită identic, cu acelaşi prototip, (termenul specific este **supradefinită**) în clasele derivate se numeşte funcţie virtuală. Declaraţia unei astfel de funcţii este precedată de cuvântul cheie ***virtual***.
- **Atenţie : termenul pentru funcţiile virtuale este de metodă supradefinită (overridden) şi nu de supraîncărcată (overloaded), nici redefinită (redefined)**
- O metodă virtuală, spre deosebire de o funcţie supraîncărcată, trebuie să aibă acelaşi prototip cu cel al metodei corespunzătoare din clasa de bază
- Metoda **supradefinită** ca virtuală este mai departe virtuală şi în clasele derivate ulterior. Implicit deci, caracterul virtual al unei metode se transmite şi metodelor corespunzătoare din clasele derivate.
- Dacă la definirea unei funcţii virtuale în clasele derivate se schimbă fie numărul, fie tipul parametrilor, ea devine o metodă nouă, obişnuită, care ascunde celelalte implementări cu acelaşi nume din clasa de bază şi natura sa virtuală se pierde.
- Funcţiile virtuale trebuie să fie funcţii membre ale unei clase, ceea ce nu este absolut necesar în cazul funcţiilor redefinite prin supraîncărcare.
- Atunci când se implementează o ierarhie de clase folosind mecanismul moştenirii, funcţiile virtuale se supun următoarei reguli : **dacă o clasă derivată nu supradefineşte o funcţie virtuală, este folosită implementarea definită în clasa de bază.**
- La apelul unei metode virtuale se va executa o funcţie ce depinde de tipul obiectului care a invocat metoda
- **Mecanismul polimorfic funcţionează numai în situaţiile în care se face apel la metode virtuale prin intermediul unor pointeri la clasele de bază.**
- Constructorii nu pot fi metode virtuale.
- Clasele unei ierarhii exploatate prin pointeri la clasa de bază trebuie să aibă **destructorul clasei de bază virtual.**
- **Destructorul unei clase de bază care conţine cel puţin o metodă virtuală trebuie să fie obligatoriu virtual**.
- Având în vedere semnificaţia mecanismului de moştenire ca implementare a relaţiei "un fel de" (*kind of* sau *is a*), un obiect al unei clase derivate poate fi folosit drept obiect al clasei de bază (evident nu şi invers).
- **Mecanismul polimorfic se pune în evidenţă şi în cazul funcţiilor ce au drept parametri obiecte ale unei ierarhii de clase, dar numai în situaţiile în care parametrii formali corespunzători sunt pointeri la clasele de bază ale ierarhiilor sau referinţe la clasele de bază**.

Exemplul de mai sus se poate rescrie:

Exemplul 2.c)

```
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual double area ()
      { return (0); }
  };

class CRectangle: public CPolygon {
  public:
    double area ()// merge si fara virtual!
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    double area ()
      { return (width * height / 2); }
  };
```

```
int main () {
  CRectangle rect;
  CTriangle trgl;
  CPolygon poly;

  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  CPolygon * ppoly3 = &poly;

  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  ppoly3->set_values (4,5);

  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  cout << ppoly3->area() << endl;
  return 0;
}
```

➢ **Exemplul 3.** Destructorii claselor de bază care exploatează obiecte din clase derivate prin intermediul pointerilor la clasa de bază trebuie să fie metode virtuale

```
#include <iostream>
using namespace std;

class Base
{
       public:
          Base(){ cout<<"Constructor:
Base"<<endl;}
          ~Base(){ cout<<"Destructor :
Base"<<endl;}
};

class Derived: public Base
{
       public:
           Derived(){ cout<<"Constructor:
Derived"<<endl;}
           ~Derived(){ cout<<"Destructor :
Derived"<<endl;}
};

int main(void)
{
        Base *Var = new Derived;
        delete Var;
        return 0;
}
```

Constructor: Base
Constructor: Derived
Destructor : Base

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

class Base
{
       public:
          Base(){ cout<<"Constructor:
Base"<<endl;}
          virtual ~Base(){
cout<<"Destructor : Base"<<endl;}
};

class Derived: public Base
{
       public:
           Derived(){ cout<<"Constructor:
Derived"<<endl;}
           ~Derived(){ cout<<"Destructor :
Derived"<<endl;}
};

int main(void)
{
        Base *Var = new Derived;
        delete Var;
        return 0;
}
```

Constructor: Base
Constructor: Derived
Destructor : Derived
Destructor : Base

- În cazul în care o funcţie virtuală definită într-o clasă de bază nu realizează nici o acţiune semnificativă, ea fiind prezentă doar pentru a permite implementarea polimorfismului, se poate folosi conceptul de **funcţie pur virtuală**.
- O astfel de funcţie nu este definită în clasa de bază, aceasta conţinând doar prototipul ei.
- Sintaxa generală de declarare a unei funcţii pur virtuale este:

virtual tip nume_functie (lista_de_parametri) = 0;

- Iniţializarea cu zero a acestei funcţii spune compilatorului că în clasa de bază pentru această funcţie nu există nici o definiţie (un corp).
- Nu se pot declara destructori pur virtuali. Chiar dacă un destructor virtual se declară ca pur virtual el trebuie să aibă un corp vid.
- O funcţie pur virtuală trebuie obligatoriu supradefinită în clasele derivate.
- Dacă o clasă conţine cel puţin o funcţie pur virtuală, ea se numeşte **clasă abstractă** .
- Întrucât o clasă abstractă conţine cel puţin o funcţie virtuală care nu prezintă corp propriu, ea este incompletă şi, în consecinţă, nu se pot crea obiecte din acea clasă. Prin urmare, clasele abstracte există doar pentru crea baza ierarhiei de clase .

Exemplul 2.d)

```
#include <iostream>
using namespace std;

class CPolygon {
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual double area()=0;
  };

class CRectangle: public CPolygon {
  public:
    double area ()
      { return (width * height); }
  };

class CTriangle: public CPolygon {
  public:
    double area ()
      { return (width * height / 2); }
  };

int main () {
  CRectangle rect;
  CTriangle trgl;
  // CPolygon poly;
  // cannot instantiate abstract class
  CPolygon * ppoly1 = &rect;
  CPolygon * ppoly2 = &trgl;
  ppoly1->set_values (4,5);
  ppoly2->set_values (4,5);
  cout << ppoly1->area() << endl;
  cout << ppoly2->area() << endl;
  return 0;
}
```

- Polimorfismul se poate pune în evidenţă şi în cazul funcţiilor ce au drept argumente pointeri sau referinţe la clasa de bază

Exemplul 4. Transferul parametrilor

```
#include <iostream>
using namespace std;
class ObjFuncVirtual
{
    public :
        virtual void Func(void) const {cout
<< "Base" << endl;}
};
class ObjFuncDerived1 : public
ObjFuncVirtual
{
    public:
        virtual void Func(void) const {
     cout << "Func1" << endl;}
};
class ObjFuncDerived2 : public
ObjFuncVirtual
{
    public:
        virtual void Func(void) const {
     cout << "Func2" << endl;}
};
class ObjFuncDerived3 : public
ObjFuncVirtual
{
    public:
        virtual void Func(void) const {
     cout << "Func3" << endl;}
};
void FFunc1(ObjFuncVirtual F){
    F.Func();
}
void FFunc2(const ObjFuncVirtual &F){
    F.Func();
}
void FFunc3(ObjFuncVirtual *PF){
    PF->Func();
}
int main(void)
{
    ObjFuncDerived1 Ob1;
    ObjFuncDerived2 Ob2;
    ObjFuncDerived3 Ob3;
    FFunc1(Ob1);
    FFunc1(Ob2);
    FFunc1(Ob3);
    FFunc2(Ob1);
    FFunc2(Ob2);
    FFunc2(Ob3);
    FFunc3(&Ob1);
    FFunc3(&Ob2);
    FFunc3(&Ob3);
    return 0;
}
```

REZULTATE

```
Base
Base
Base
Func1
Func2
Func3
Func1
Func2
Func3
```

Exemplul 5. Definirea în clasa derivată a unei funcţii cu acelaşi nume cu cel al unei funcţii din clasa de bază, dar cu un număr diferit de parametri sau cu tip returnat diferit ascunde funcţia din clasa de bază!

```
class Base
{
  public:
    void f(double x) // Nu conteaza daca e virtuala sau nu
;
...
};

class Derived : public Base
{
   public:
    void f(char c);  // Nu conteaza daca e virtuala sau nu
...

};
...

int main()
{
  Derived* d = new Derived();
  Base* b = d;
  b->f(65.3);  // OK: paseaza 65.3 catre f(double x)
  d->f(65.3);  // Bizar: converteste 65.3 la char ('A' daca e ASCII) si apleaza
               // f(char c); NU apeleaza f(double x) care e ascunsa!!!
//warning C4244: 'argument' : conversion from 'double' to 'char', possible loss of data

  delete d;
  return 0;

 }
```

Mecanismul de implementare a polimorfismului

- Rezolvarea apelului de funcţie virtuală (conexiunea dintre apel şi adresa funcţiei care se execută) se realizează în timpul execuţiei programului şi nu la compilare aşa cum se întâmplă la celelalte funcţii
- În cazul funcţiilor nevirtuale, rezolvarea apelului se face în faza de compilare printr-un mecanism care se numeşte *early binding (legare timpurie, iniţială, statică)*
- În cazul funcţiilor virtuale, rezolvarea apelului se face în faza de execuţie printr-un mecanism care se numeşte *late binding (legare târzie, ulterioară, dinamică)*
- Concret, pentru fiecare clasă ce conţine o metodă virtuală se construieşte un tablou cu numele VTABLE (denumiri alternative : “virtual function table”, “virtual method table” sau “dispatch table”) ce conţine adresele metodelor virtuale ale clasei respective
- Fiecare clasă derivată îşi are propriul sau tablou VTABLE în care sunt înscrise, în aceeaşi ordine cu cea din clasa de bază, adresele metodelor virtuale. Dacă o clasă derivată nu supradefineşte o metodă din clasa de bază, în tabel se înscrie adresa metodei clasei de bază
- La instanţierea unui obiect al ierarhiei de clase se inserează, transparent faţă de utilizator, un pointer către tabloul VTABLE specific clasei căreia îi aparţine obiectul şi cunoscând deplasamentul (decalajul faţă de începutul tabelului) metodei virtuale apelate se poate obţine dinamic adresa funcţiei care trebuie executată

Transferul parametrilor în contextul moştenirii

- Un obiect al unei clase derivate poate fi tratat şi ca un obiect aparţinând clasei de bază
- Utilizarea adresei unui obiect (fie ca pointer, fie ca referinţă) şi tratarea acesteia ca fiind adresa tipului corespunzător clasei de bază se numeşte **upcasting**
- Dacă transferul parametrilor se face prin referinţă sau se utilizează pointeri către clasa de bază, se realizează un upcast al parametrilor şi se pot utiliza proprietăţile ce derivă din polimorfism
- Dacă transferul parametrilor se realizează prin valoare, atunci, dacă parametrul formal este de tipul clasei de bază, iar parametrul efectiv este de tipul clasei derivate, nu mai are loc procesul de upcasting, obiectul clasei derivate este ”trunchiat” (sliced) şi se pierd componentele specifice clasei derivate
- Evitarea unei astfel de situaţii se poate face construind clasa de bază ca abstractă

Exemplul. 6 - Fără metode virtuale

```
#include <iostream>
using namespace std;

class Paralelogram
{
public:
  Paralelogram(){ cout<<"Constr. Paral\n";}
  void afis() const {
    cout << "Paralelogram::afis" << endl;
  }
  ~Paralelogram(){cout<<"Destr. Paral\n";}
};

class Dreptunghi : public Paralelogram {
public:
  Dreptunghi(){ cout<<"Constr.
Dreptunghi\n";}
  void afis() const {
    cout << "Dreptunghi::afis" << endl;
  }
  ~Dreptunghi(){cout<<"Destr. Drept\n";}
};

class Patrat : public Dreptunghi {
public:
  Patrat(){cout<<"Constr. Patrat\n";}
  void afis() const {
    cout << "Patrat::afis" << endl;
  }
  ~Patrat(){cout<<"Destr. Patrat\n";}
};

void display(Paralelogram& p) {
  p.afis();
}
```

```
int main(void)
{
  Paralelogram par;
  Dreptunghi dre;
  Patrat pat;
  Paralelogram tab[]={par, dre, pat};
// se apel. Constr. Cop Paralelog.
  Paralelogram *tabp[]={&par, &dre, &pat};
  cout<< "Afis tab\n";
  for(int i=0;i<3;i++)
      tab[i].afis();
  cout<< "Afis ptab\n";
  for(i=0;i<3;i++)
      tabp[i]->afis();
  cout<<"Display ...\n";
  display(par);
  display(dre);
  display(pat);
  return 0;
}
```

Constr. Paral
Constr. Paral Constr. Dreptunghi
Constr. Paral Constr. Dreptunghi Constr. Patrat
Afis tab
Paralelogram::afis Paralelogram::afis Paralelogram::afis
Afis ptab
Paralelogram::afis Paralelogram::afis Paralelogram::afis
Display ...
Paralelogram::afis Paralelogram::afis Paralelogram::afis
Destr. Paral
Destr. Paral
Destr. Paral
Destr. Patrat Destr. Drept Destr. Paral
Destr. Drept Destr. Paral
Destr. Paral

Exemplul 4. a) Cu metode virtuale şi destructori virtuali

```
#include <iostream>
using namespace std;
class Paralelogram
{
public:
  Paralelogram(){ cout<<"Constr.
Paral\n";}
  virtual void afis() const {
    cout << "Paralelogram::afis" <<
endl;
  }
  virtual ~Paralelogram(){cout<<"Destr.
Paral\n";}
};
class Dreptunghi : public Paralelogram {
 public:
   Dreptunghi(){ cout<<"Constr.
Dreptunghi\n";}
   virtual void afis() const
   {
    cout << "Dreptunghi::afis" << endl;
   }
  virtual ~Dreptunghi(){cout<<
      "Destr. Drept\n";}
};

class Patrat : public Dreptunghi {
 public:
  Patrat(){cout<<"Constr. Patrat\n";}
  virtual void afis() const
  {
    cout << "Patrat::afis" << endl;
  }
  virtual ~Patrat();
// Numai in prototip!
};

Patrat::~Patrat(){
  cout<<"Destr. Patrat\n";}

void display(Paralelogram& p)
{
  p.afis();}
int main(void)
{
  Paralelogram par; Paralelogram *p=
     new Paralelogram;
  Dreptunghi dre;  Paralelogram *pd =
     new Dreptunghi;
  Patrat pat; Paralelogram *pp=
     new Patrat;
  Paralelogram *tabp[]= {p,pd,pp};
  cout<< "Afis ptab\n";
  for(int i=0;i<3;i++)
      tabp[i]->afis();
  cout<<"Display ...\n";
 display(par);display(dre);display(pat);
  for(i=0;i<3;i++)
      delete tabp[i];
   return 0;
}
```

REZULTATE

Constr. Paral
Constr. Paral
Constr. Paral Constr. Dreptunghi
Constr. Paral Constr. Dreptunghi
Constr. Paral Constr. Dreptunghi Constr. Patrat
Constr. Paral Constr. Dreptunghi Constr. Patrat
Afis ptab
Paralelogram::afis
Dreptunghi::afis
Patrat::afis
Display ...
Paralelogram::afis
Dreptunghi::afis
Patrat::afis
Destr. Paral
Destr. Drept Destr. Paral
Destr. Patrat Destr. Drept Destr. Paral
Destr. Patrat Destr. Drept Destr. Paral
Destr. Drept Destr. Paral
Destr. Paral

b) Cu metode virtuale şi destructori Nevirtuali

Constr. Paral
Constr. Paral
Constr. Paral Constr. Dreptunghi
Constr. Paral Constr. Dreptunghi
Constr. Paral Constr. Dreptunghi Constr. Patrat
Constr. Paral Constr. Dreptunghi Constr. Patrat
Afis ptab
Paralelogram::afis
Dreptunghi::afis
Patrat::afis
Display ...
Paralelogram::afis
Dreptunghi::afis
Patrat::afis
Destr. Paral
Destr. Paral
Destr. Paral
Destr. Patrat Destr. Drept Destr. Paral
Destr. Drept Destr. Paral
Destr. Paral

Exemplul 5

```
#include <string>
#include <iostream>
class Animal
{
protected:
    std::string m_strNume;

    // ATTN: constructorul este protected !
    // Nu se permite crearea de obiecte de tip Animal direct,
    // Doar clasele derivate il pot utiliza!
    Animal(std::string strNume) : m_strNume(strNume)
    {
    }

public:
    std::string GetNume() { return m_strNume; }
 //  virtual const char* Glasuieste()
 //         const char* Glasuieste()
//          {
//                  return "???";
//          }
          virtual const char* Glasuieste() =0;
};

class Pisica: public Animal
{
public:
    Pisica(std::string strNume)  : Animal(strNume)
    {
    }
   virtual const char* Glasuieste()
 //    const char* Glasuieste()
          {
            return "Miau Miau";
          }
};

class Caine: public Animal
{
public:
    Caine(std::string strNume) : Animal(strNume)
    {
    }

   virtual const char* Glasuieste()
 //         const char* Glasuieste()
          {
                  return "Hau Hau";
          }
};
// Obs. Animal::GetNume() nu e virtuala.
// Nu e necesat pentru ca GetNume()
// nu e supradefinita de nici una din clasele derivate

void Raport(Animal rAnimal)
//void Raport(Animal &rAnimal)
{
// Atunci cand functia Glasuieste este virtuala
// functia Raport opereaza corect!
    std::cout << rAnimal.GetNume() << " face ... " <<
                  rAnimal.Glasuieste() << std::endl;
}

int main()
{
    Pisica cPisica("Fifi");
    Caine cCaine("Gigi");

    Raport(cPisica);
    Raport(cCaine);
     return 0;
}
```

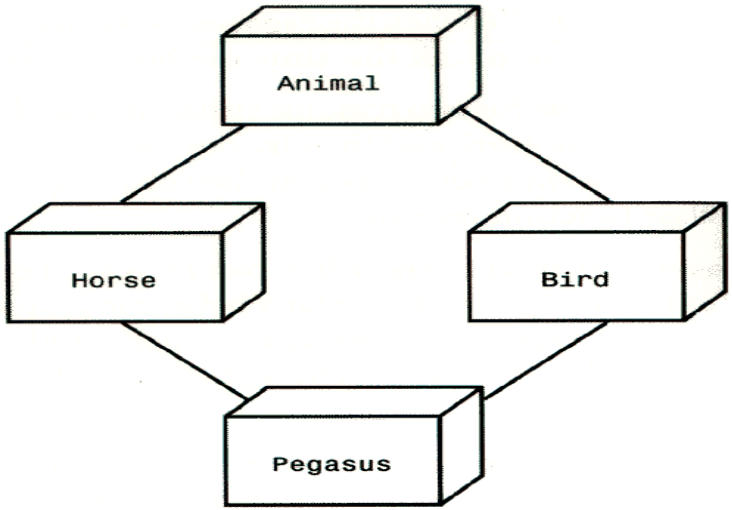
Exemplul 6. Exploatarea polimorfismului şi late binding

```
#define MAX 100
#include <iostream>
#include <string.h>

using namespace std;
class Persoana
{
        protected:
     char* ptrNume;
        public:
    Persoana(char* pn=NULL)
    {
       int lung = strlen(pn);
       ptrNume = new char[lung+1];
       strcpy(ptrNume, pn);
   }
   virtual  ~Persoana()      =0;
   virtual void puneData()
   {
                cout << "\nNume = " << ptrNume;
        }
};

Persoana:: ~Persoana()
{
       cout << "Destructor Persoana\n";
       if(ptrNume != NULL)
         delete[] ptrNume;
}

class Prof : public Persoana
{
        private:
                int numPub;
        public:
        Prof(char* n, int p) :
                Persoana(n), numPub(p)    {}
        virtual void puneData()
         {
                Persoana::puneData();
                cout << " Publicatii=" << numPub << endl;
         }

         virtual ~Prof() {
         cout << "Destructor Prof\n";}
};


class Student : public Persoana
{
  private:
     char* ptrTitlu;
  public:
        Student(char* n, char* t) :
        Persoana(n), ptrTitlu(NULL)
        {
                int lung = strlen(t);
                ptrTitlu = new char[lung+1];
                strcpy(ptrTitlu, t);
        }
         virtual ~Student()
         {
          cout << "Destructor Student\n";
          if(ptrTitlu != NULL)
                delete[] ptrTitlu;
        }
          virtual void puneData()
        {
           Persoana::puneData();
          cout << " Subiectul Tezei = " << ptrTitlu<<endl;
       }
};
```

```
int main(void)
{
	int j, nr;
	Persoana* ptrPers[MAX];
	char nume[40];
	char titlu[80];
	int n = 0;
	char aleg;
	do
	{
		cout << "Numele: ";
		cin >>nume;
		cout << "Student sau profesor(s/p): ";
		cin >> aleg;
		if(aleg=='s')
		{
		   cout<<"Titlul tezei :";
		   cin>> titlu;
			ptrPers[n] = new  Student(nume, titlu);
			n++;
		}
		else if(aleg=='p')
		{
			cout<<"Nr. publicatiilor:";
			cin>>nr;
			ptrPers[n] = new Prof(nume, nr);
			n++;
		}
		cout << "Continuati (d/n)? ";
		cin >> aleg;
	} while( aleg=='d'  && n<MAX);
	 for(j=0; j<n; j++)
		ptrPers[j]->puneData();
	for(j=0; j<n; j++)
		delete ptrPers[j];
	return 0;
}
```

Clase de bază virtuale. Moştenire virtuală.

- În cazul moştenirii multiple există situaţii în care o clasă de bază poate fi moştenită indirect prin intermediul claselor ce au o aceeaşi clasă de bază, aşa cum se sugerează în figura de mai jos :



- Pentru a înlătura posibilele ambiguităţi generate de astfel de situaţii se foloseşte noţiunea de clasă de bază virtuală şi respectiv de moştenire virtuală
- Sintaxa declarării moştenirii virtuale este:

```
class B;
class B1 : virtual public B { .....};
class B2 : virtual public B { .....};
...
class Deriv: public B1, public B2, ..., public Bn { .....};
```

În acest mod, în clasa Deriv se va găsi o singură parte corespunzătoare moştenirii din clasa B. În absenţa modificatorului de acces ***virtual***, în clasa Deriv ar fi existat *n* părţi identice, cu componente moştenite din clasa B, pe filierele B1, B2 ... Bn.

Exemplul 7. Moştenire virtuală

```
#include <iostream>
using namespace std;
typedef int HANDS;
enum COLOR { Red, Green, Blue, Yellow, White, Black, Brown } ;

class Animal        // common base to both horse and bird
{
   public:
      Animal(int);
      virtual ~Animal() { cout << "Animal destructor...\n"; }
      virtual int GetAge() const { return itsAge; }
      virtual void SetAge(int age) { itsAge = age; }
   private:
      int itsAge;
};

Animal::Animal(int age):itsAge(age)
{
      cout << "Animal constructor...\n";
}

//class Horse : virtual public Animal
class Horse : public Animal
{
   public:
      Horse(COLOR color, HANDS height, int age);
      virtual ~Horse() { cout << "Horse destructor...\n"; }
      virtual void Whinny()const { cout << "Whinny!... "; }
      virtual HANDS GetHeight() const { return itsHeight; }
      virtual COLOR GetColor() const { return itsColor; }
   protected:
      HANDS itsHeight;
      COLOR itsColor;
};

Horse::Horse(COLOR color, HANDS height, int age):
      Animal(age),itsColor(color),itsHeight(height)
{
      cout << "Horse constructor...\n";
}

//class Bird : virtual public Animal
class Bird : public Animal
{
   public:
      Bird(COLOR color, bool migrates, int age);
      virtual ~Bird() {cout << "Bird destructor...\n";  }
      virtual void Chirp()const { cout << "Chirp... ";  }
      virtual void Fly()const
      { cout << "I can fly! I can fly! I can fly! "; }
      virtual COLOR GetColor()const { return itsColor; }
      virtual bool GetMigration() const { return itsMigration; }
   protected:
      COLOR itsColor;
      bool itsMigration;
};

Bird::Bird(COLOR color, bool migrates, int age):
      Animal(age),itsColor(color), itsMigration(migrates)
{
      cout << "Bird constructor...\n";
}

class Pegasus : public Horse, public Bird
{
   public:
      void Chirp()const { Whinny(); }
      Pegasus(COLOR, HANDS, bool, long, int);
      ~Pegasus() {cout << "Pegasus destructor...\n";}
      virtual long GetNumberBelievers() const
         { return  itsNumberBelievers; }
      virtual COLOR GetColor()const { return Horse::itsColor; }
   private:
      long itsNumberBelievers;
   };
```

```
    Pegasus::Pegasus( COLOR aColor,
        HANDS height,       bool migrates,
        long NumBelieve,       int age):
    Horse(aColor, height,age),
    Bird(aColor, migrates,age),
    //Animal(age*2),
    itsNumberBelievers(NumBelieve)
    {
      cout << "Pegasus constructor...\n";
    }

    int main()
    {
      Pegasus *pPeg = new Pegasus(Red, 5, true, 10, 2);
       int age = pPeg->GetAge();
       cout << "This pegasus is " << age << " years old.\n";
     delete pPeg;
     return 0;
}
```

Concluzii:

- Constructorii nu pot fi funcţii virtuale
- Destructorii pot fi funcţii virtuale
- Funcţiile virtuale sunt funcţii membru nestatice
- În principiu, dacă se urmăreşte ***late binding***-ul, funcţiile virtuale nu pot fi ***inline***