Derivare, Ierarhii - plan

- Relația de derivare
  - Sintaxa
  - Tipuri de derivare
  - Constructori, Destructori
  - Conversii standard
  - Copiere
  - Conflicte
  - Derivare virtuală
- Polimorfism
  - suprascriere -> legare statică
    - exemplu
  - funcții virtuale -> legare dinamică
    - exemplu

Ierarhii cpp - Introducere

- C++ a împrumutat de la Simula conceptul de clasă (ca tip utilizator) și cel de ierarhie de clase
- Clasele ar trebui utilizate pentru a modela concepte – atât din domeniul aplicațiilor cât și din cel al programării
- Un concept nu există izolat, el coexistă cu alte concepte cu care relaționează
- Automobil
  - roți, motor, șoferi, pietoni, camion, ambulanță, șosea, ulei, combustibil, parcare, motel, service, etc.
- Exprimarea părților comune, a relațiilor ierarhice dintre clase se realizează prin noțiunea de derivare

```
struct Angajat{
  string nume, prenume;
  char initiala;
  Data data_angajarii;
  short departament;
  //…
}

struct Manager{
  Angajat ang; // datele managerului
  list<Angajat*> grup;//echipa condusa
  //…
}
```

- Un manager este un angajat; compilatorul nu poate înțelege acest lucru din codul scris
- Mai mult, un Manager* nu este un Angajat* deci nu pot pune un Manager într-o listă de Angajați decât dacă scriem o conversie explicită
- Soluția corectă: specificarea(și pentru compilator) că un Manager este un Angajat

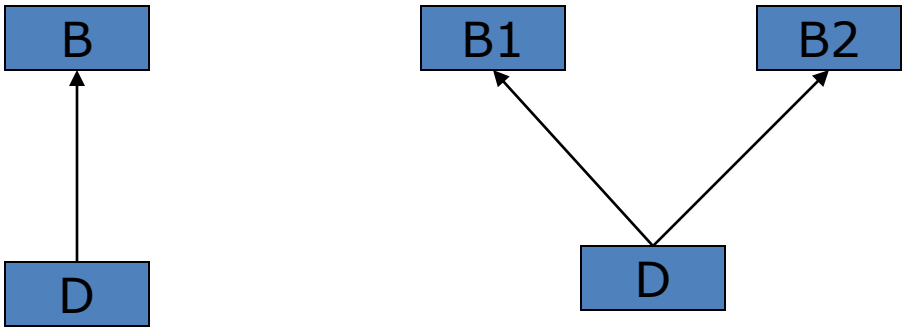
```
struct Manager:public Angajat{
  list<Angajat*> grup;//echipa condusa
  //…
}
```

- Se spune că Manager este derivată din Angajat, că Angajat este clasă de bază pentru Manager
- Clasa Manager are ca membri, membrii clasei Angajat și în plus, membrii declarați acolo
- Clasa derivată moștenește de la clasa de bază, relația se mai numește moștenire.
- Clasa de bază este numită uneori o superclasă iar clasa derivată o subclasă
  - Confuzie uneori: clasa derivată este “mai mare” decât clasa de bază în sensul că ea poate conține date și funcții în plus

Relația de moştenire – Clase derivate

- În C++ : clasă de bază (superclasă), clasă derivată (subclasă)

- Moştenire simplă, moştenire multiplă



Clase derivate

- Sintaxa:
  - Moştenire simplă:

```
class ClsDer:tip_moşt ClsBaza {};
```

  - Moştenire multiplă:

```
class ClsDer :tip_moşt ClB1, tip_moşt ClB2,…
    { };
```

tip_moşt :: **public|protected|private**

Clase derivate

- Nivele de protecție(acces) a membrilor unei clase:
  - **`public`**: cunoscut de oricine
  - **`protected`**: cunoscut de clasa proprietară, prieteni şi de clasele derivate
  - **`private`**: cunoscut de clasa proprietară şi de prieteni

- Tipuri de moştenire:
  - **`public`**: membrii **`public`** (**`protected`**) în bază rămân la fel în clasa derivată
  - **`protected`**: membrii **`public`** în clasa de bază devin **`protected`** în clasa derivată
  - **`private`**: membrii **`public`** şi **`protected`** din clasa de bază devin **`private`** în clasa derivată; este tipul implicit de moştenire

Clase derivate

- Relația **`friend`** nu este moştenită, nu este tranzitivă

- În modul de derivare **`private`** se poate specifica păstrarea protecției unor membri:

```
class D:private B{
protected: B::p; public: B::q;
//…
};
```

- Accesul la membri : Funcțiile membru ale unei clase de bază pot fi redefinite în clasa derivată:
  - Ele pot accesa doar membrii **`public`** sau **`protected`** din clasa de bază
  - Pot accesa funcțiile din clasa de bază folosind operatorul de rezoluție ::

Clase derivate

- Constructori, Destructori
  - Constructorii şi destructorul nu se moştenesc
  - Constructorii clasei derivate apelează constructorii clasei de bază:
    - Constructorii impliciți nu trebuie invocați
    - Constructorii cu parametri sunt invocați în lista de inițializare
    - Ordinea de apel: constructor clasă de bază, constructori obiecte membru, constructor clasă derivată
  - Obiectele clasei derivate se distrug în ordine inversă: destructor clasă derivată, destructori membri, destructor clasă de bază

```
class Angajat{
public:
   Angajat(const string& n, int d);
   //...
private:
   string nume, prenume;
   short departament;
   //..
};
class Manager:public Angajat{
public:
   Manager(const string& n, int d, int niv);
   //...
private:
   list<angajat*> grup;
   short nivel;
   //...
};
```

```
Angajat::Angajat(const string& n, int d)
        :nume(n), departament(d)
{
   //...
}
Manager::Manager(const string& n, int d, int niv)
        :Angajat(n,d), // initializarea bazei
         nivel=niv // initializare membri
{
   //..
}
```

- Constructorul clasei derivate poate inițializa membrii clasei de bază prin invocarea constructorului acesteia în lista de inițializare

Clase derivate - Conversii standard

- Un obiect al clasei derivată poate fi convertit implicit la unul din clasa de bază
  - Un Manager poate fi folosit oriunde este acceptabil un Angajat
- O adresă a unui obiect derivat poate fi convertită implicit la o adresă de obiect din clasa de bază
  - Un Manager& poate fi folosit ca un Angajat&
- Un pointer la un obiect derivat poate fi convertit implicit la un pointer la obiect din clasa de bază
  - Un Manager* poate fi folosit ca un Angajat*
- Conversia reciprocă poate fi definită cu un constructor în clasa derivată

Clase derivate – Copiere

- Copierea o face constructorul de copiere şi **`operator=`**
- În cazul membrilor pointeri aceştia trebuie să existe explicit
- Ordinea de apel a constructorului de copiere :

Clasa de bază	**Clasa derivată**	**Ordinea de apel**
implicit	implicit	clasa baza, clasa derivată
explicit	implicit	clasa baza, clasa derivată
implicit	explicit	constructorul clasei derivate
explicit	explicit	constructorul clasei derivate trebuie sa apeleze constructorul clasei de bază

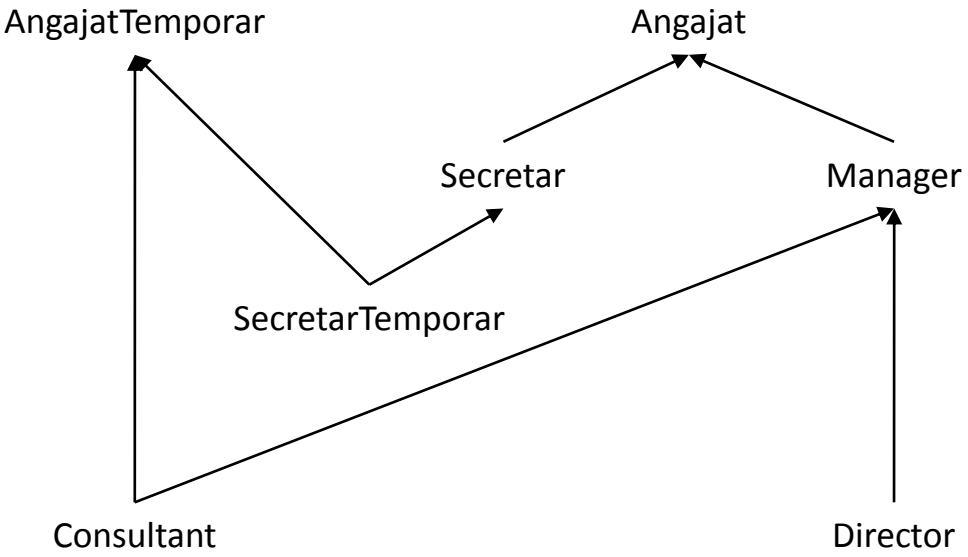
Copiere

```
class Angajat{
public:
  //...
  Angajat&operator=(const Angajat&);
  Angajat(const Angajat&);
  //...
};
void f(const Manager& m){
  Angajat e = m;
  // se construieste e din partea Angajat a lui m
  e = m;
   // se atribuie lui e partea Angajat a lui m
}
```

- Copierea, atribuirea nu se moștenesc

Ierarhii de clase

```
class Angajat{/*...*/);
class Manager::public Angajat{/*...*/);
class Director::public Manager{/*...*/);
class Secretar::public Angajat{/*...*/);
class AngajatTemporar{/*...*/);
class SecretarTemporar::public AngajatTemporar, public Angajat{/*...*/);
class Consultant::public AngajatTemporar, public Manager{/*...*/);
```



- Utilizarea claselor derivate presupune rezolvarea problemei: dat un pointer la clasa de bază, cărui tip derivat aparține obiectul pointat?
- Soluții:
  1. Asigurarea ca sunt pointate doar obiecte de un singur tip
  2. Folosirea unui câmp tip în clasa de bază
  3. Utilizarea operatorului dynamic_cast
  4. Utilizarea funcțiilor virtuale
- Pointerii la clase de bază sunt folosiți la proiectarea containerelor(mulțimi, vectori, liste)
  - Soluția 1 conduce la folosirea containerelor omogene (obiecte de același tip)
  - Soluțiile 2-4 permit construirea de containere eterogene
  - Soluția 3 este varianta implementată de sistem a soluției 2
  - Soluția 4 este o variantă sigură a soluției 2

Câmp tip în clasa de bază

```
struct Angajat{
   enum Ang_type{M, A};
   Ang_type tip;
   string nume, prenume;
   char initiala;
   Data data_angajarii;
   short departament;
   //…
}
```

```
struct Manager{
      Manager() {tip = M;}
      Angajat ang;
      list<Angajat*> grup
      short nivel;
      //…

}
```

```
   void print_angajat(const Angajat* a){
      switch Angajat(a->tip){
      case Angajat::A:
           cout << a -> nume << '\t' << a->departament << '\n';
           //…
           break
      case Angajat::M:
           cout << a -> nume << '\t' << a->departament << '\n';
           //…
           const Manager* p = static_cast<const Manager*>(a);
           cout << "nivel " << p->nivel<< '\n';
           //..
   }
```

Câmp tip în clasa de bază

```
void print_list(const list<Angajat*>& alist){
   for(list<Angajat*>::const_iterator p=alist.begin();
                                p!=alist.end(); ++p)
        print_angajat(*p);
}
```

- Varianta care ține cont de partea comună:

```
void print_angajat(const Angajat* a){
   cout << a -> nume << '\t' << a->departament << '\n';
   //..
   if(a->type == Angajat::M){
        const Manager* p = static_cast<const Manager*>(a);
        cout << "nivel " << p->nivel<< '\n';
        //..
}
```

Funcții virtuale

```
class Angajat{
public:
   Angajat(const string& n, int d);
   virtual void print() const//...
private:
   string nume, prenume;
   short departament;
   //..
};
```

- Compilatorul va ști să invoce funcția potrivită atunci când print() este transmisă unui obiect Angajat
- Tipul argumentelor funcției virtuale în clasele derivate trebuie să fie aceleași

- O funcție virtuală trebuie definită pentru clasa în care a fost prima dată declarată (cu excepția cazului funcție virtuală pură)

```
void Angajat::print() const{
   cout << nume << '\t' << departament << '\n';
}
class Manager:public Angajat{
public:
   Manager(const string& n, int d, int niv);
   void print() const; // nu mai e nevoie de cuvantul virtual
   //...
private:
   list<angajat*> grup;
   short nivel;
   //...
};
void Manager::print() const{
   Angajat::print();
   cout << "\tnivel " << nivel << '\n';
}
```

Funcții virtuale

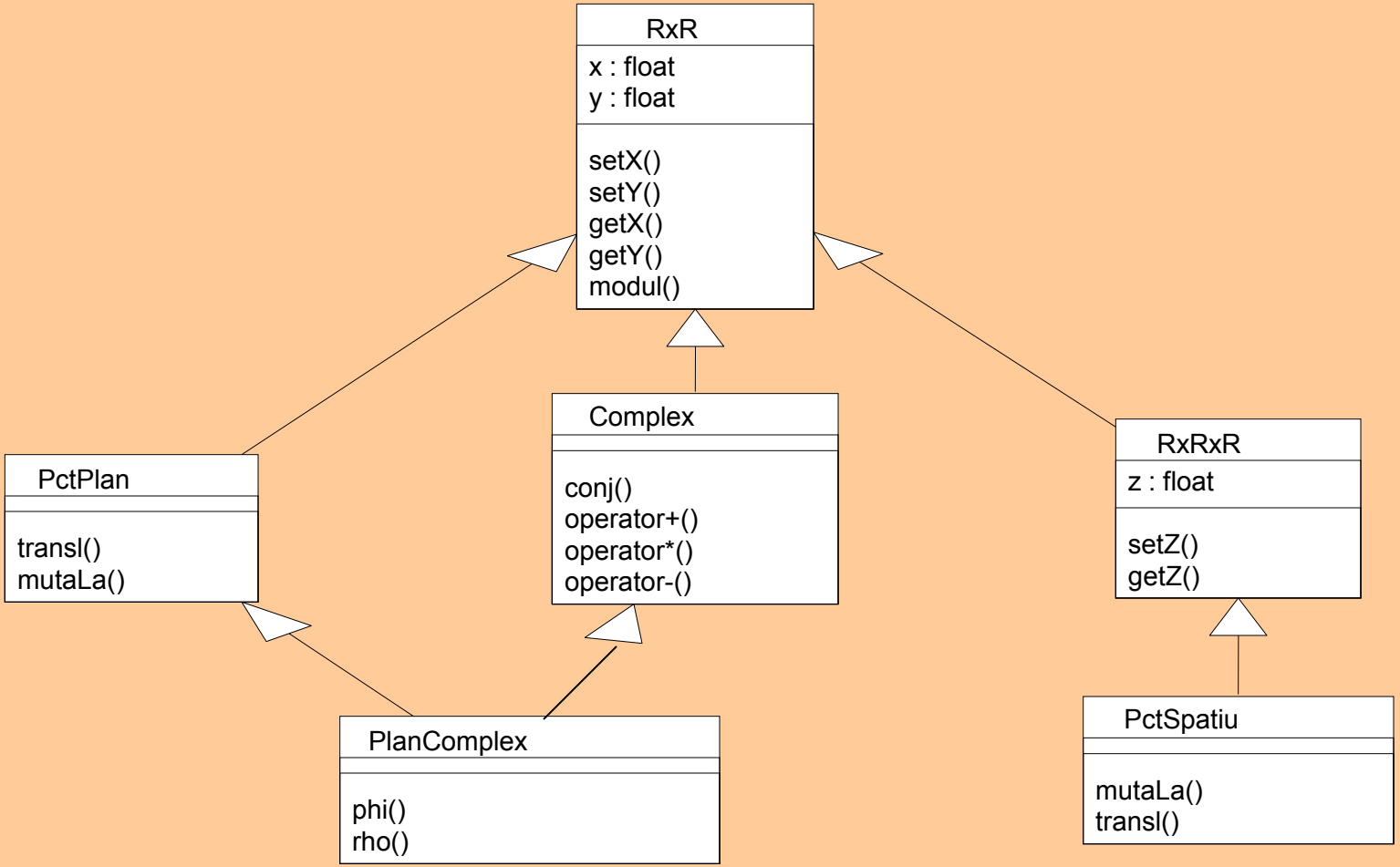
- O funcție din clasa derivată cu același nume și același set de tipuri de argumente cu o funcție virtuală din clasa de bază se zice că extinde (override) versiunea din clasa de bază
- Programatorul poate indica versiunea ce trebuie apelată
  - `Angajat::print();`
- Altfel, compilatorul alege versiunea potrivită, în funcție de obiectul pentru care este invocată

```
void print_list(const list<Angajat*>& alist){
   for(list<Angajat*>::const_iterator p=alist.begin();
       p!=alist.end(); ++p)
       (*p)->print();
}
int main()
{
   Angajat a("Ionescu", 1234);
   Manager m("Popescu", 1234, 2);
   list<Angajat*> ang;
   ang.push_front(&a);
   ang.push_front(&m);
   print_list(ang);
   return 0;
```

- Rezultatul execuției:

```
Popescu 1234
        nivel 2
Ionescu 1234
```

Ierarhia RxR



Ierarhia RxR

```
class RxR {
 protected:
  double x, y;
 public:
  RxR(double un_x = 0, double un_y = 0) : x(un_x), y(un_y) {}
  ~RxR() {}
  void setX(double un_x) { x = un_x; }
  double getX() {return x;}
  void setY(double un_y) { y = un_y; }
  double getY() { return y; }
  double modul();
};
class PctPlan : public RxR {
 public:
  PctPlan(double un_x=0, double un_y=0) : RxR(un_x, un_y) {}
  ~PctPlan() {}
  void translCu(double, double);
  void mutaLa(PctPlan&);
};
```

Ierarhia RxR

```
class Complex : public RxR {
 public:
  Complex(double un_x=0, double un_y=0) : RxR(un_x, un_y) {}
  Complex conj();
  Complex operator+ (Complex&);
};

class RxRxR : public RxR {
 protected:
  double z;
 public:
  RxRxR(double un_x, double un_y, double un_z)
        : RxR(un_x, un_y), z(un_z) {}
  void setZ(double un_z) { z = un_z; }
  double getZ() {return z;}
  double modul();
};
class PlanComplex : public PctPlan, public Complex {};
```

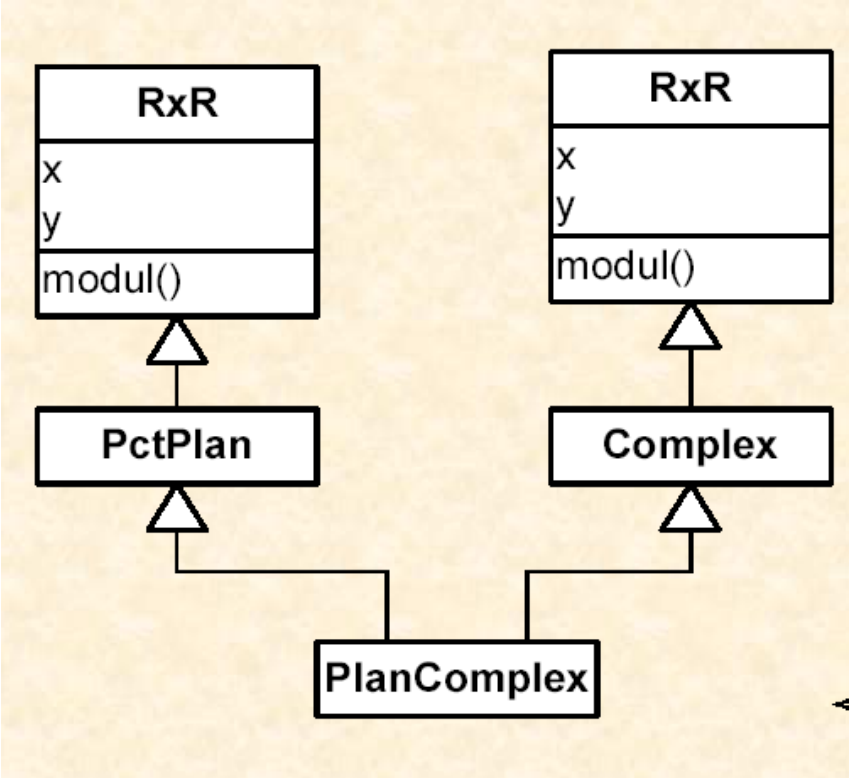
Clase derivate – Conflicte

- Conflict de metodă: metode cu acelaşi nume în clase incomparabile A, B ce derivează o clasă D

- Conflict de clasă: clasa D derivată din A1 şi A2 iar acestea sunt derivate din B: B este “accesibilă” pe două căi din D

Clase derivate – Conflicte

- Conflict de clasă: clasa PlanComplex derivată din PctPlan şi Complex iar acestea sunt derivate din RxR



Ierarhia RxR

```
class Complex : public RxR {
 public:
  Complex(double un_x=0, double un_y=0) :
            RxR(un_x, un_y) {}
  Complex conj();
  Complex operator+ (Complex&);
};
class PlanComplex : public PctPlan, public Complex{
public:
   PlanComplex(double = 0, double = 0);
   ~PlanComplex(){};
};
```

Ierarhia RxR

```
PlanComplex::PlanComplex(double un_x, double un_y) : PctPlan(un_x, un_y),
    Complex(un_x, un_y)
{}
PlanComplex::~PlanComplex()
{
    // nimic
}
void main(void) {
  PlanComplex pc(5, 5);
  cout << pc.modul() << endl;

  PlanComplex pc2;
  pc2.setX(5);
  pc2.setY(5);
  cout << pc2.modul() << endl;
}

// 'PlanComplex::modul' is ambiguous
//could be the 'modul' in base 'RxR' of base 'PctPlan' of class 'PlanComplex`
//or the 'modul' in base 'RxR' of base 'Complex' of class 'PlanComplex'
```

Moştenire fără partajare

- Fiecare clasă derivată are câte un exemplar din datele şi metodele clasei de bază

PctPlan::x

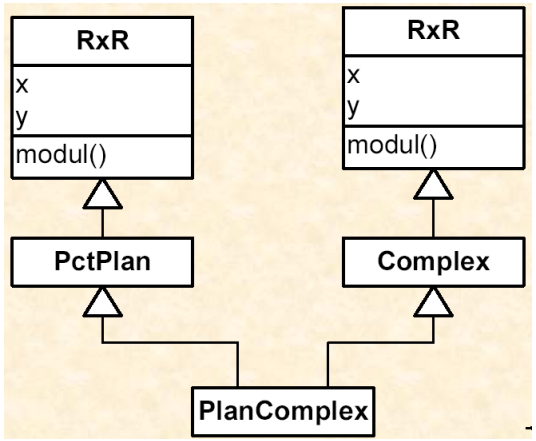
PctPlan::y

Complex::x

Complex ::y

PctPlan::modul()

Complex ::modul()



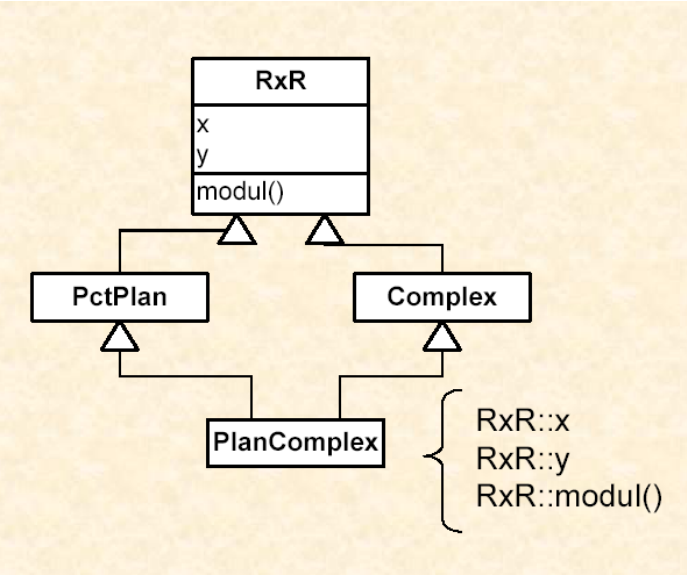
- Soluția (cazul derivării multiple):
  - Clasa de bază virtuală poate fi clasă de bază indirectă de mai multe ori fără a duplica membrii

Partajare: clase derivate virtuale

- Derivare virtuală:

```
class A1:virtual public B{};
class A2:public virtual B{};
class D: public A1, public A2{};
```

```
class PctPlan : virtual public RxR ...
class Complex : virtual public RxR ...
```



Clase derivate virtuale

- În cazul derivării virtuale, constructorul fiecărei clase derivate este responsabil de inițializarea clasei virtuale de bază:

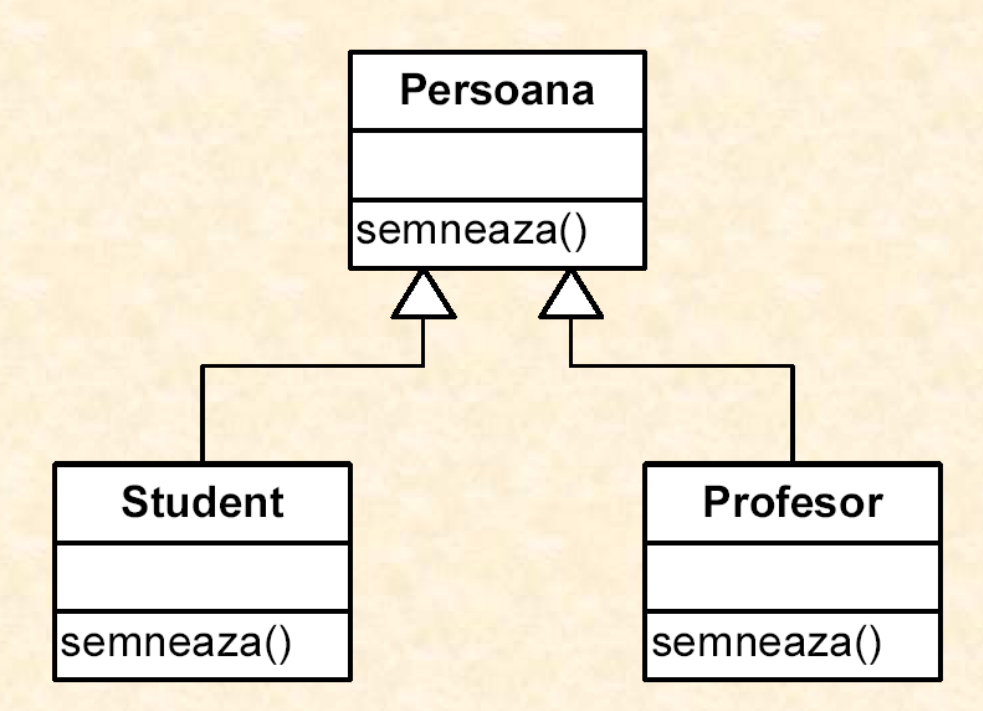
```
class Baza{
public:
   Baza(int){}
   //
};
class D1 : virtual public Baza{
public:
   D1():Baza(1){}
   //
};
class D2 : virtual public Baza{
public:
   D2():Baza(6){}//…
};
```

```
class m1:public D1, public D2{
public:
        m1():Baza(2){}
        //
};
class m2:public D1, public D2{
public:
        m2():Baza(3){}
        //
};
```

- Un constructor implicit în clasa de bază virtuală, simplifică lucrurile (dar le poate şi complica!)

Polimorfism - suprascriere

- Legare statică: asocierea *apel funcție -- corp(implementare) funcție* se face înainte de execuția programului (*early binding*) – la compilare



Polimorfism - suprascriere

- persoanele, studenții, profesorii au abilitatea de a semna:

```
class Persoana {
public:
   //…
   string getNume() const;
   void semneaza();
protected:
   string id;
   string nume;
};
class Student:public Persoana {
   //…
   void semneaza();
}
class Profesor:public Persoana {
   //…
   void semneaza();
}
```

Polimorfism - suprascriere

- fiecare semnează in felul său:

```
void Persoana::semneaza()
{
   cout << getNume() << endl;
}
void Student::semneaza()
{
   cout << "Student " << getNume() << endl;
}
void Profesor::semneaza()
{
   cout << "Profesor " << getNume() << endl;
}
```

Polimorfism - suprascriere

- Exemplu:

```
Persoana pers("1001","Popescu Ion");
Student stud("1002", "Angelescu Sorin");
Profesor prof("1003","Marinescu Pavel");
pers.semneaza();
stud.semneaza();
prof.semneaza();
```

```
Popescu Ion
Student Angelescu Sorin
Profesor Marinescu Pavel
```

Polimorfism - suprascriere

- Exemplu:

```
// studentul poate semna ca persoana
stud.Persoana::semneaza();

// ... si profesorul poate semna ca persoana
prof.Persoana::semneaza();
```

```
Angelescu Sorin
Marinescu Pavel
```

Polimorfism - suprascriere

- Suprascriere -> legare statică (asocierea apel funcție – implementare se face la compilare):

```
void semneaza(Persoana& p)
{
    p.semneaza();
};
semneaza(pers);
semneaza(stud);// &stud poate inlocui &p
semneaza(prof);// &prof poate inlocui &p
```

```
Popescu Ion
Angelescu Sorin
Marinescu Pavel
```

Polimorfism - Funcții virtuale

- Legare dinamică: asocierea *apel funcție -- corp(implementare) funcție* se face la execuția programului (*late binding, run-time binding*), pe baza tipului obiectului căruia i se transmite funcția ca mesaj. În acest caz, funcția se zice *polimorfă*

Implementare polimorfism în C++:

- Declarația funcției în clasa de bază precedată de cuvântul **`virtual`**

```
class Persoana {
public:
   //…
   virtual void semneaza();
   //…
};
```

- O funcție virtuală pentru clasa de bază rămâne virtuală pentru clasele derivate. La redefinirea unei funcții virtuale în clasa derivată (*overriding*) nu e nevoie din nou de specificarea **`virtual`**

Polimorfism – funcții virtuale

- Polimorfism funcții virtuale-> legare dinamică:

```
Persoana pers("1001","Popescu Ion");
Student stud("1002", "Angelescu Sorin");
Profesor prof("1003","Marinescu Pavel");
pers.semneaza();
stud.semneaza();
prof.semneaza();
semneaza(pers);
semneaza(stud);
semneaza(prof);
```

```
Popescu Ion
Student Angelescu Sorin
Profesor Marinescu Pavel
Popescu Ion
Student Angelescu Sorin
Profesor Marinescu Pavel
```

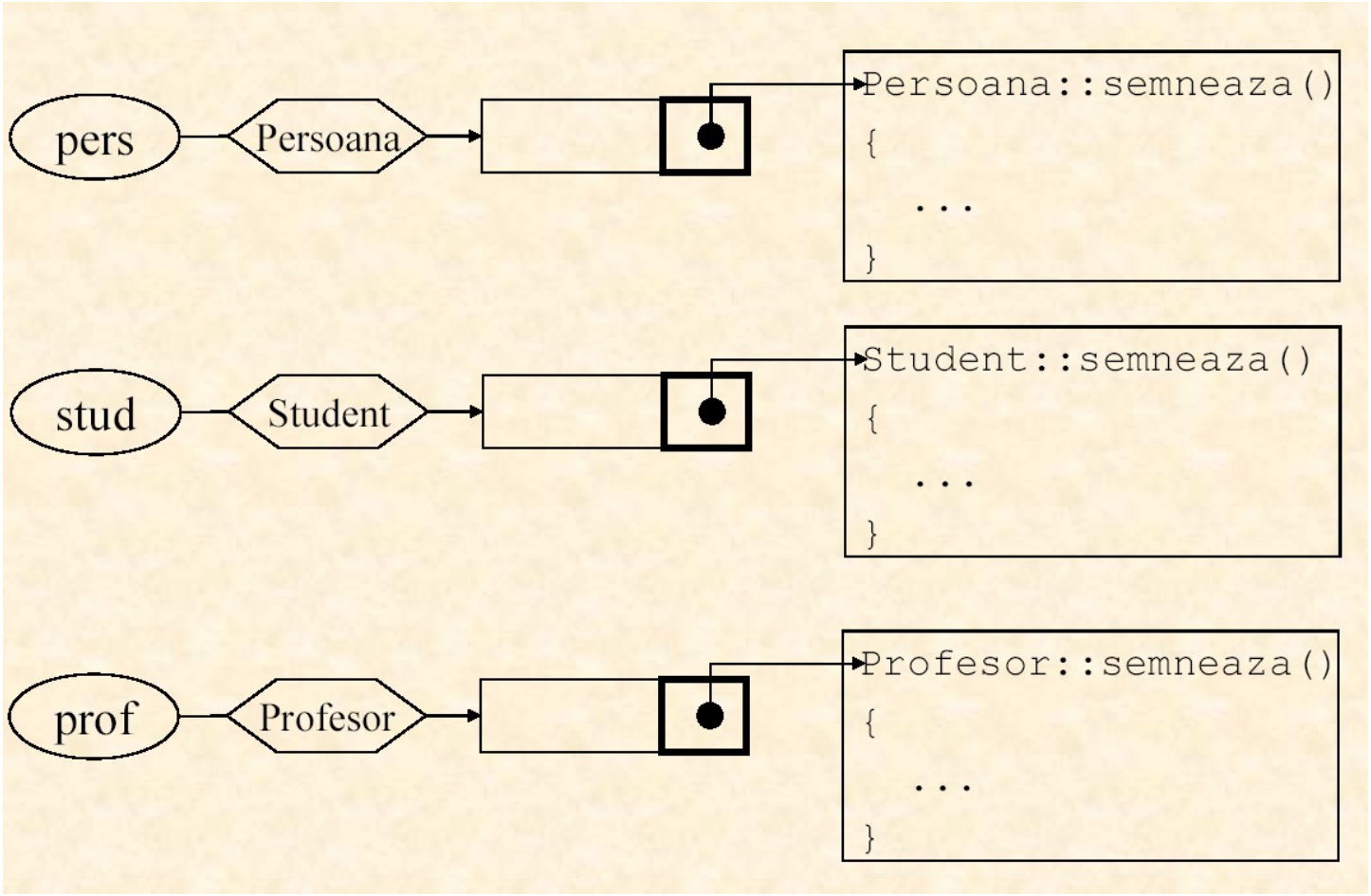
Implementare polimorfism în C++:

- Compilatorul creează o tabelă - VTABLE - pentru fiecare clasă care conține funcții virtuale; adresele tuturor funcțiilor virtuale sunt plasate în această tabelă. Sistemul creează un pointer VPTR în fiecare clasă cu funcții virtuale; el pointează la această tabelă şi alege funcția corectă la un apel polimorfic

```
std::cout << sizeof(pers);
```

- Se obține:
  - 32 dacă funcția `semneaza()` nu este virtuală
  - 36 dacă funcția `semneaza()` este virtuală

Implementare polimorfism în C++:



Implementare polimorfism în C++:

- Orice funcție membru poate fi declarată virtual, dar:
  - constructorii nu pot fi virtual

- Un destructor poate fi virtual; destructorii claselor derivate ar trebui să fie virtuali; asta asigură apelul lor la distrugerea cu **`delete`** a pointerilor la clasa de bază

Destructor virtual

```
class A{
public:
   A(){p = new char[5];}
   ~A(){delete [] p;}
protected:
   char* p;
};
class D:public A{
public:
   D(){q = new char[500];}
   ~D(){delete [] q;}
protected
   char* q;
};
```

```
void f(){
   A* pA;
   pA = new D();
   //…
   delete pA;// doar apel ~A()
}
void main(){
   for(int i = 0; i<9; i++)
        f();
}

// pentru a se apela si ~D()
// se declara virtual ~A(){…}
```