

## C7: Mostenire. Clase derivate.

### Cuprins:

- Reutilizarea codului - Agregare si derivare
- Clase derivate (Mostenire)
  - Apelul constructorilor si al destructorilor
  - Operatorul=
  - Vizibilitatea in clasele derivate
  - Tipuri de derivare
  - Conversii la atribuirea obiectelor
  - Conversii la atribuirea pointerilor
  - Redefinirea metodelor unei clase de baza intr-o clasa derivata

# Reutilizarea codului

## ➤ Agregarea – o relatie intre clase de tipul “has a” / “has many”

```
class A{  
    private : lista attribute  
    public: lista metode  
};
```

```
class B{  
    A a;  
    // alte attribute si metode  
};
```

- clasa B are un atribut de tipul clasei A
- clasa B nu are acces direct la attributele din A, dar poate sa utilizeze functiile membre publice sau friend ale acesteia pentru a realiza operatiile de interes
- **se ascunde interfata clasei A (attributele si metodele din A) - nu se observa in interfata clasei B**
- se protejaza incapsularea datelor (din A)
- se reutilizeaza codul deja implementat pentru A
- pentru a crea un obiect de tip B se apeleaza in mod implicit constructorul default din A (daca nu este specificat altul)
- constructorul de copiere (generat automat) din B il apeleaza automat pe cel din A
- operatorul= (generat automat) al lui B, il apeleaza pe cel al tipului de date A
- destructorul clasei B apeleaza automat destructorul clasei A

**Observatie:** clasa B putea sa aiba - mai multe attribute de tip A: A\*vec

- attribute de tipurile altor clase: A1 a1, ...An an;

## ➤ Derivarea – o relatie intre clase de tipul “is a” / “is many”

```
class Produs
{
    string nume;
    int pret;
    char cod[10];
};
```

```
class Electrocasnic
{
    string nume;
    int pret;
    char cod[10];
    int durata_gar;
};
```

```
class Aliment
{
    string nume;
    int pret;
    char cod[10];
    Data data_exp;
};
```

```
class Jucarie
{
    string nume;
    int pret;
    char cod[10];
    int varsta_recomandata[2];
};
```

```
class TV
{
    string nume;
    int pret;
    char cod[10];
    int durata_gar;
    double diag;
};
```

```
class Masina_cafea
{
    string nume;
    int pret;
    char cod[10];
    int durata_gar;
    string tip;
}; //filtru; espresso; etc
```

....

**=> Ar trebui sa rescriem mult cod.**  
**& Nu avem tocmai relatii “has a”**  
(nu pot sa zic ca un TV are un Produs)

Un TV e un Electrocasnic care este un  
Produs => **mostenire / derivare**

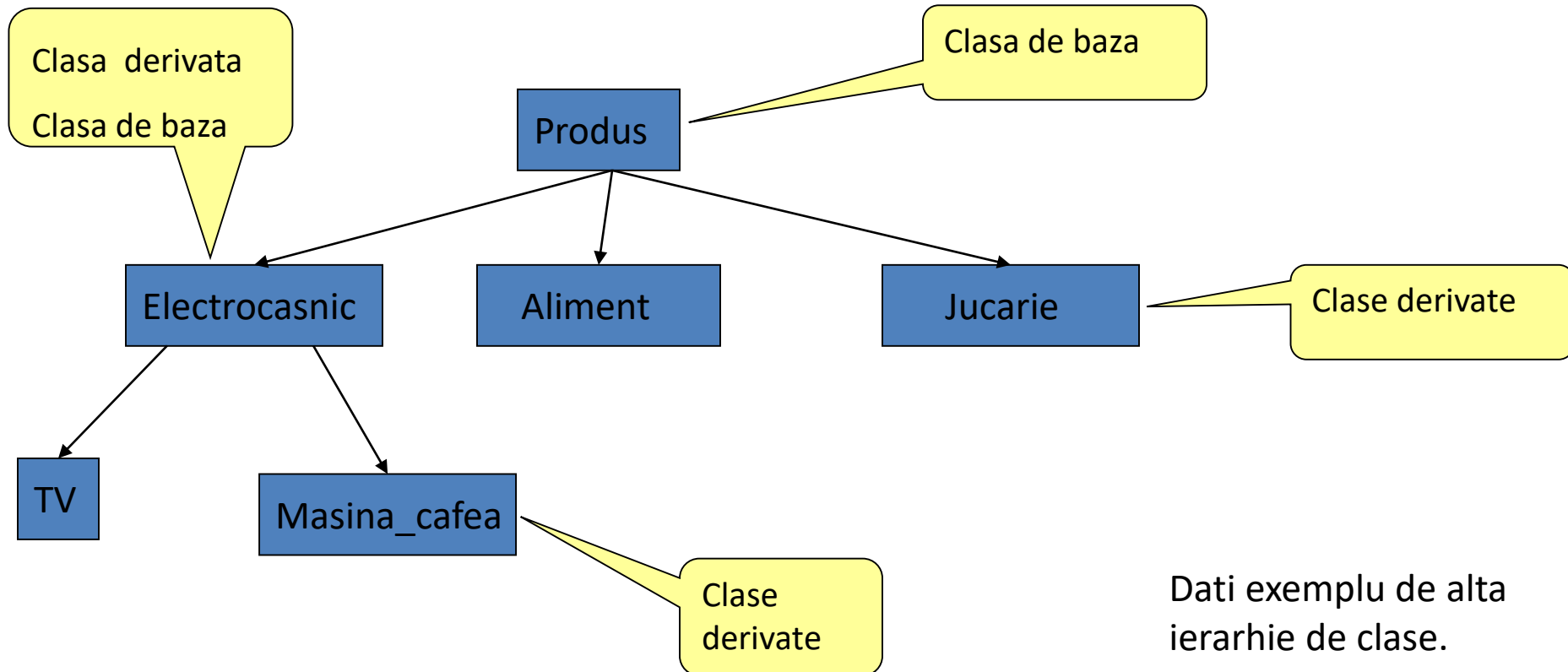
## C7: Mostenire. Clase derivate.

- se poate observa urmatorul aspect: avem o clasa generala (de baza) – **Produs**
- se pot distinge mai multe categorii de produse – care trebuie sa aiba attribute si comportamente de produs, **DAR** care mai au in plus si attribute si comportamente specifice (clase specializate)
- Notiuni folosite: **clase de baza -> clase derivate**

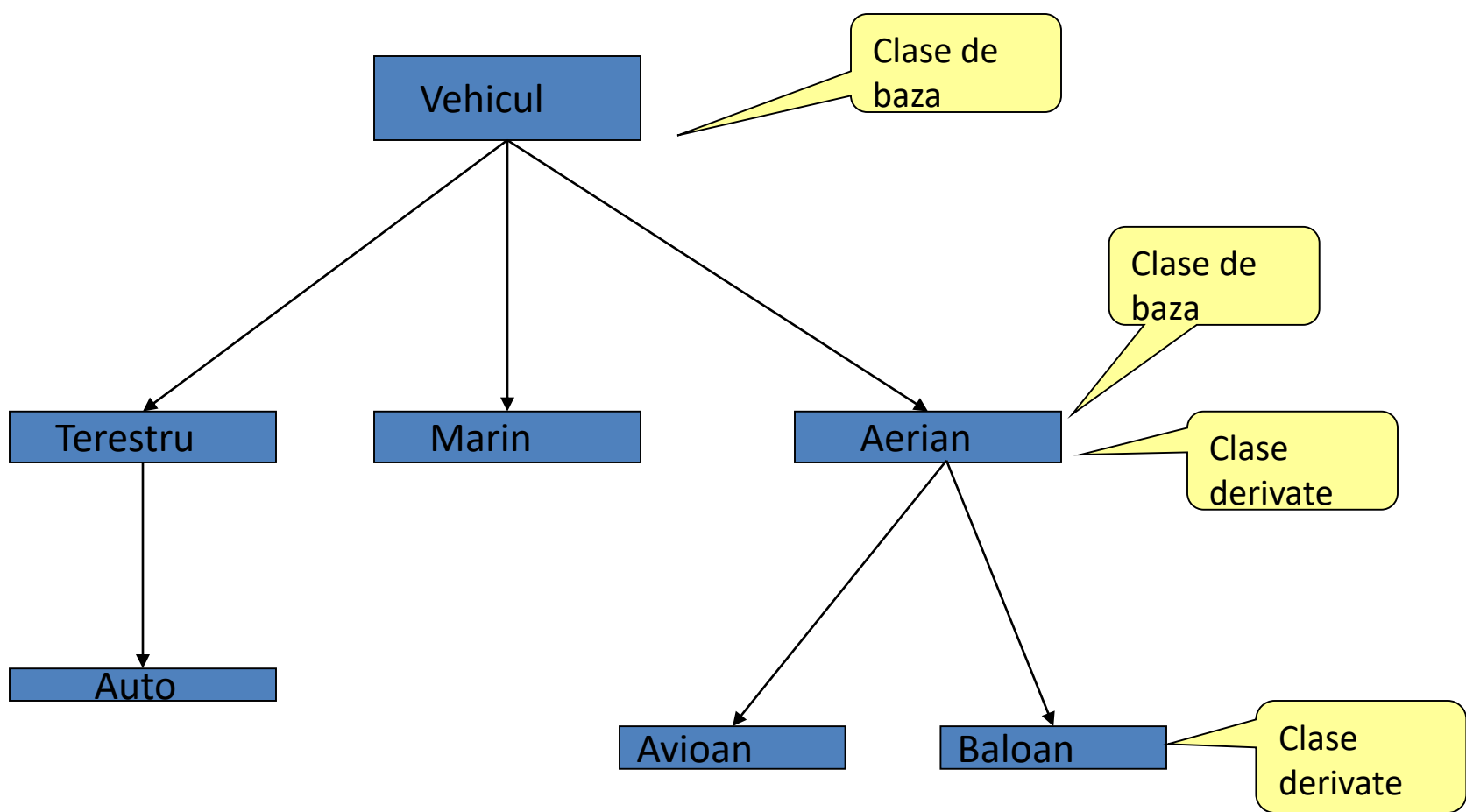
superclase -> subclase

=> ierarhie de clase

clase parinte -> clase copil



Dati exemplu de alta ierarhie de clase.



Pot sa am **oricate clase derivate dintr-o clasa de baza** si **oricate niveluri de derivare** (cat timp este logic). Clasele derivate preiau – **mostenesc** – attribute si metode ale claselor de baza – spunem ca **preiau interfata clasei de baza**.

Pot, in acelasi timp, sa am **mai multe clase parinte (mostenire multipla)**.

**Ex:** Clasele de baza: Angajat - are salariu; lucreaza la o firma

Student - are note; invata la o facultate

Clasa derivata: Student\_Angajat – are salariu, note; lucreaza la o firma, invata la o facultate. Un Student Angajat este si Angajat si Student.

## C7: Mostenire. Clase derivate.

### Exemplu:

```
#include <iostream>
using namespace std;
// clasa de baza
class Baza
{protected:
    int atr1; // attributele si metodele declarate protected sunt vizibile in clasa de baza
              //si toate clasele derivate din ea, dar nu sunt vizibile in afara ierarhiei de clase
public:      //attributele si metodele declarate public sunt vizibile oriunde
    Baza(int i):atr1(i){ //apel pseudoconstructor pentru atr1 de tip int
        //sau //atr1=i; //doar constructorii pot apela alti constructori/pseudoconstructori
    }

    void set_atr1(int i){
        atr1=i;
    }

    void afisare_atr1(){
        cout << " atr1 = " << atr1 << endl;
    }
}; //salvata ca Baza.h –trebuia impartit in interfata si implementare
```

```

#include "Baza.h"           //nu mai includem iostream, a fost inclus in Baza.h
// clasa derivata din tipul Baza – cod salvat in Derivata.h
class Derivata: public Baza
{
    //mosteneste atrib. si met. clasei de baza;adica le trateaza ca fiind attributele/metodele sale
    int atr2;

public:
    // Derivata(){}          //oare ce se intampla?

    Derivata(int a1, int a2):Baza(a1),atr2(a2){ //constructorul pentru Baza trebuie apelat expres inainte
    }                                           //de implementarea constructorului clasei derivate,
                                              // daca nu - se apeleaza constructorul default pentru Baza

    void set_atr2(int n){
        atr2 = n;
    }
    void afisare_atr2(){
        cout << " atr2 = " << atr2 << endl;;
    }
    void set_atr12(int n,int m){
        atr1 = n; //am acces direct la atr1 pentru ca e atribut al clasei Derivata ;
        //pot, daca vreau, sa il modific cu functia membra mostenita din Baza: set_atr1(n);
        atr2 = m;           //sau:      //set_atr2(m);
    }
    void afisare_atr12(){
        cout << " atr1 = " << atr1<<endl;    //sau:      //afisare_atr1();
        cout << " atr2 = " << atr2<<endl;    //sau:      //afisare_atr2();
    }
}; //care sunt attributele si metodele clasei Derivata?

```

**Tema:** Separati  
interfata clasei de  
implementare!

```
//program de test
#include <cstdlib>
using namespace std;
#include "Derivata.h"    //nu includ si "Baza.h" deoarece este inclusa in headerul Derivata.h
                        //altfel am eroare: "redefinition of `class Baza` "
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    Baza b(1);           // creez un obiect b de tip Baza
```

```
    b.afisare_atr1();
```

```
    cout<<endl<<"_____ "<<endl;
```

```
    Derivata d(1,1);     // creez un obiect d de tip Derivata
```

```
    d.afisare_atr12();
```

```
    cout<<endl<<"_____ "<<endl;
```

```
    d.set_atr1(2);        // folosind metoda set_atr1 mostenita din clasa Baza schimb atr1 al
                          //obiectului d de tip Derivata
```

```
    d.afisare_atr1();      // si il afisez cu metoda mostenita din Baza - afisare_atr1()
```

```
    cout<<endl<<"_____ "<<endl;
```

```
    //d.atr1; // nu am acces la el din afara ierarhiei de clase, deoarece este protected
```

```
    return 0;
```

```
} //la iesirea din main se apeleaza destructorii pentru b si d ; cum?
```



## C7: Mostenire. Clase derivate.



**Vizibilitatea membrilor (attribute si metode) unei clase:**

<b>Indicator vizibilitate</b>	<b>Accesibilitate</b>	<b>Zona</b>
public:	accesibil	- din exteriorul ierarhiei
	accesibil	- din clasele derivate
protected:	inaccesibil	- din exteriorul ierarhiei
	accesibil	- din clasele derivate si orice alte clase derivate din clasele derivate
private:	inaccesibil	- din exteriorul ierarhiei
	inaccesibil	- din clasele derivate

## C7: Mostenire. Clase derivate.

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

```
class A{
    public:
    A(){
        cout<<"constr fara param"<<endl;
    }


    A(const A& a){
        cout<<"constr copiere"<<endl;
    }

    A& operator=(const A& a){
        cout<<"op="<<endl;
        return *this;
    }

    ~A(){
        cout<<"destr"<<endl;
    }
};
```

```
class B:public A{
    //cfp, cc, =, destr – generati automat
};

int main() {
    B b;
    B c(b);
    c=b;
    return 0;
}
//tot codul e in acelasi fisier
```

 D:\CURSURI\POO\deriv.exe

```
constr fara param
constr copiere
op=
destr
destr
```

```
-----
Process exited after 0.03392 seconds with r
Press any key to continue . . .
```

## C7: Mostenire. Clase derivate.

### Derivarea

- Procedeu prin care se creaza un nou tip de date (o noua clasa) folosind o clasa existenta, mai exact, se adauga cod nou (attribute si functionalitati noi) la codul deja existent - scris pentru alt tip de date - > **se preia interfata clasei de baza**
- In clasa derivata sunt mostenite toate attributele si metodele clasei de baza (indiferent de vizibilitatea lor) si le pot accesa direct daca sunt declarate public sau protected.
- Mare parte a “muncii” este facuta de compilator (o sa intelegem mai bine la cursul viitor - de ce). Pentru moment, ganditi-va la modul in care sunt construite obiectele si mostenite attribute si metode.
- **“This magical act is called inheritance.”**
- **Mostenirea/Derivarea este unul dintre cele mai importante concepte ale POO**

## C7: Mostenire. Clase derivate.

### Apelul automat al constructorilor/operatorului=/destructorului

#### Constructori

- d.p.d.v. al ordinii, intai este/sunt apelat/i constructor-ul/ii clase-i/lor de baza si apoi implementarea celui din clasa derivata;
- in cazul in care clasa derivata nu are niciun constructor declarat, se genereaza unul default care va apela constructorul fara parametrii al clasei de baza (trebuie sa existe);
- acelasi lucru e valabil si pentru constructorul de copiere din clasa derivata – generat automat -> il apeleaza pe cel de copiere din clasa de baza;
- in cazul in care implementam un constructor al clasei derivate care nu apeleaza (:) un constructor al clasei de baza (este omisa lista de constructori), atunci se va apela constructorul fara parametrii al clasei de baza (daca nu exista ->eroare);

#### Operator=

- daca e generat automat, apeleaza operatorul= din clasele de baza; daca e implementat de noi, trebuie sa ne asiguram ca are acest comportament

#### Destructor

- d.p.d.v. al ordinii, intai este apelat destructorul clasei derivate si apoi cel al clasei de baza;
- nu se face o apelare explicita deoarece exista un singur destructor in clasa de baza

## C7: Mostenire. Clase derivate.



### Observatii:

1. Constructorii, destructorii, operatorii de atribuire (=,...) si functiile friend **NU** se mostenesc in clasele derivate.
2. Functiile friend pot accesa attributele si metodele protected din clasa cu care sunt prietene (si, bineinteles, pe cele publice si private).

# C7: Mostenire. Clase derivate.

## Apelul listei de constructori

Forma generala a constructorilor claselor derivate - **apelul listei de constructori** pentru clasele de baza:

**Constr\_clasa\_derivata (semnatura) : Constr\_clasa\_de\_baza\_1 (lista\_param\_1),  
Constr\_clasa\_de\_baza\_2(lista\_param\_2),...,Constr\_clasa\_de\_baza\_n(lista\_param\_n)**

lista\_param\_i, i=1,n este cuprinsa in semnatura

### Observatii

1. La apelul constructorului clasei de baza (deoarece e un apel de functie) nu se precizeaza tipurile parametrilor:

Ex:      Derivata(int a1, int a2): Baza(a1){...}

2. Ordinea de apel a constructorilor din lista(indiferent de ordinea efectiva in lista) este data de ordinea in care s-a facut derivarea:

```
class A: public B, public C
```

```
{int atr3;//...
```

```
  A(lista_parametrii_formali):atr3(ume_param),C(lista_param_C), B(lista_param_B) {///..}  
};
```

Ordinea efectiva de apel: B(lista\_param\_B), C(lista\_param\_C), atr3(ume\_param)

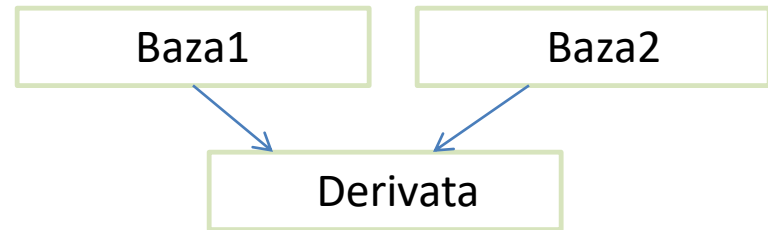
# C7: Mostenire. Clase derivate.

## Mostenire multipla (introducere)

```
#include <iostream>
using namespace std;
// clasa de baza 1
class Baza1
{protected:
    int atr1;
public:
    Baza1(int i =0):atr1(i){}
    void set_atr1(int i){ atr1=i;}
    void afisare_atr1(){ cout << "\n atr1 = " << atr1;}
};
```

---

```
#include <iostream>
using namespace std;
// clasa de baza 2
class Baza2
{protected:
    int atr2;
public:
    Baza2(int i = 0):atr2(i){}
    void set_atr2(int i){ atr2=i;}
    void afisare_atr2(){cout << "\n atr2 = " << atr2;}
};
```



```

#include "Baza1.h"
#include "Baza2.h"
// clasa Derivata
class Derivata: public Baza1, public Baza2
{
    int atr3;                                // mosteneste attributele si metodele claselor de baza
public:
    Derivata(){}    //se apeleaza pe rand Baza1() si Baza2() – exista implementati?
    Derivata(int a1, int a2, int a3):Baza1(a1),Baza2(a2),atr3(a3){}
    void set_atr3(int i){ atr3 = i;}
    void afisare_atr3(){ cout << "\n atr3 = " << atr3;}
    void set_atr123(int n,int m, int q)
    {   atr1 = n;   atr2 = m;   atr3 = q;   }
    void afisare_atr123()
    {   cout << "\n atr1 = " << atr1<< "\n atr2 = " << atr2<< "\n atr3 = " << atr3; }
};

```

---

```

#include "Derivata.h"
int main(){
    Derivata d(1,2,3);//care e ordinea de apel a constructorilor?
    d.afisare_atr123();
    cout<<endl<<"_____ "<<endl;
    d.set_atr1(3); d.set_atr2(6); d.set_atr3(9); //set_atr1, set_atr2, afisare_atr1, afisare_atr2
    d.afisare_atr1(); d.afisare_atr123(); //sunt mostenite si se pot utiliza de obiecte de
    return 0; //tip Derivata
} //cum se apeleaza destructorii?

```



## C7: Mostenire. Clase derivate.

### Mostenirea multipla

- pare simplu de implementat

### DAR

- pot aparea foarte multe probleme:
  - attribute si metode cu acelasi nume in clasele de baza –
  - derivare dubla indirecta din clasa de baza
  - etc...

C8 – C9 - cum se rezolva aceste situatii

## C7: Mostenire. Clase derivate.

### Supraincarcarea (override) si redefinirea(redefine) functiilor membre ale clasei de baza in clasa derivata

Prin derivare se mostenesc in clasa derivata functiile membre ale clasei de baza (mai putin constructorii, destructorii, operatorii de atribuire).

**Ce se intampla daca intr-o clasa derivata o sa dam o noua definitie unei functii din clasa de baza(am acelasi nume)?**

1. **redefinire** (semnatura poate sau nu sa difere)
2. Daca avem **aceeasi semnatura si tip returnat** si functia din clasa de baza era declarata **virtuala** (cursul viitor) - **supraincarcare**

In aceste cazuri – versiunile metodelor din clasa de baza o sa fie ascunse pentru noua clasa (nu am acces direct la functiile din clasa de baza cu acelasi nume ca cele din clasa derivata – chiar daca semnatura difera). Pot sa le apelez doar explicit: `nume_clasa_baza::nume_functie`

```
/*Ex1*/
#include<iostream>
using namespace std;
class Baza
{public:
    int f() { cout << "Baza::f () "; }
    int f(int i) { cout << "Baza::f (int i) "; }
};
```

```
class Der : public Baza
{public:
    int f () { cout << "Der::f () "; } //redefinire
};
int main()
{ Der d; d.fun(5);
  return 0;
}
```

Ce credeti sa se  
intampla? Testati!

## C7: Mostenire. Clase derivate.

### Redefinirea functiilor membre ale clasei de baza in clasa derivata

```
/*Ex2*/
#include <iostream>
using namespace std;
class Baza
{protected: int atr1;
public:      Baza(int i):atr1(i){}

            void set_atr(int i) { atr1=i; }

            void afisare () { cout << "\n atr1 = " << atr1; }
};

#include "Baza.h"
class Derivata: public Baza
{          int atr2;
public:    Derivata(int a1, int a2):Baza(a1),atr2(a2){}

            void set_atr(int n, int m) { atr1=n; atr2 = m;}           //redefinire

            void set_atr(int m) { atr2 = m;}                          //redefinire

            void afisare () {      cout << "\n atr1 = " << atr1<< "\n atr2 = " << atr2; } //redefinire
}; //care sunt attributele si metodele clasei Derivata
```

```
#include "Derivata.h"
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    Baza b(1);
```

```
    b.afisare(); //afisare din Baza
```

```
    cout<<endl<<"_____ "<<endl;
```

```
    Derivata d(1,1);
```

```
    d.afisare(); //afisare din Derivata; identificata in functie de tipul  
                //obiectului care o apeleaza
```

```
    cout<<endl<<"_____ "<<endl;
```

```
    d.set_atr(1,2);
```

```
    //set_atr din clasa Derivata
```

```
    d.afisare();
```

```
    cout<<endl<<"_____ "<<endl;
```

```
    d.Baza::set_atr(1000);
```

```
    //ca sa pot apela set_atr din Baza
```

```
    //nu am acces direct la aceasta functie pentru ca
```

```
    //am implementat set_atr(int) in clasa derivata
```

```
    //si aceasta a ascuns implementarea din clasa de baza
```

```
    d.Baza::afisare();
```

```
    //dar am acces la aceste functii precizand clar
```

```
    //faptul ca ma refer la implementarea din clasa Baza
```

```
    return 0;
```

```
}
```

```
    //Mai pot sa scriu si altfel aceste apeluri?
```

## C7: Mostenire. Clase derivate.

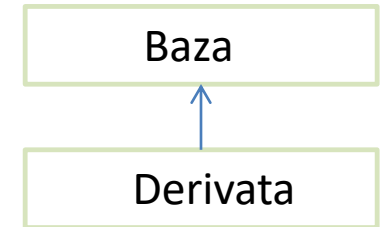
### Conversii la atribuirea obiectelor:

Daca o clasa e derivata dintr-o alta clasa - de baza (relatie “is a”) => obiectele de tipul clasei derivate sunt, in acelasi timp, si obiecte de tipul clasei de baza.

Deci pot sa fac o atribuire de tipul :

`obj_baza = obj_der;` (“upcasting” - conversie la tipul obiectului de baza)

**DAR** se vor pierde informatiile suplimentare stocate in `obj_der`.



#### •REGULA GENERALA:

- sunt permise atribuirile in care se “pierd date” (se intampla automat – upcasting)
- **NU** sunt permise cele in care nu se stie cu ce sa se completeze campurile suplimentare:

~~`obj_der = obj_baza;`~~ **NU**

! Acest tip de atribuire este totusi posibila prin supradefinirea operatorului de atribuire in clasa derivata.

```

class Baza
{protected: int atr1;
 public:  Baza(int i):atr1(i){}

        void afisare() {
                cout << " atr1 = " << atr1<<endl;
        }
};

```

```

class Derivata: public Baza
{
        int atr2;
 public:
        Derivata(int a1, int a2):Baza(a1),atr2(a2){}

        void afisare() {
                cout << " atr1 = " << atr1 <<endl;
                cout << " atr2 = " << atr2 <<endl;
        }
};

```

```

///in main()
    Baza b(1);
    Derivata d(1,1);

```

**b=d;        //Ce functie se apeleaza? Din ce clasa face parte? Ce se intampla de fapt?**  
**b.afisare();    // atr1=1;**

**d=b;        //ERROR: no match for 'operator=' in 'd = b'**

**Ce am putea sa facem ca atribuirea intre un obiect derivat si unul de baza sa functioneze?**

In clasa derivata supradefinim si operatorul = cu parametru de tipul de date Baza:

```
Derivata& operator=(const Baza & b)
{
    atr1=b.atr1;
    atr2=0; // atr2 ia o valoare default
    return *this;
}
```

```
///in main()
```

```
Baza b(1);
```

```
Derivata d(1,1);
```

**d=b; //se face atribuirea si se completeaza campul suplimentar cu valoarea default 0**

Cate metode operator= are clasa Derivata?

## C7: Mostenire. Clase derivate.

### Conversii de tip si pointeri:

#### Exemplu:

```
Baza b(1),b1(2);  
Derivata d(1,1),d1(2,2);
```

```
Baza *bp=new Baza(5);  
bp->afisare(); //5  
bp=&b; //ar trebui sa fac ceva inainte de aceasta atribuire? Ce?  
bp->afisare(); //1  
cout<<endl<<"_____ "<<endl;
```

```
bp=&d;  
bp->afisare(); // apel functia afisare din Baza //1  
cout<<endl<<"_____ "<<endl;
```

```
Derivata *dp=new Derivata(5,5);  
dp->afisare(); //5 5  
dp=&d1;  
dp->afisare(); //2 2  
cout<<endl<<"_____ "<<endl;
```

```
dp=(Derivata*)&b1;  
dp->afisare(); // apel functia afisare din Derivata  
//2 si ce gaseste la adresa urmatoare  
//comportament imprevizibil
```



## C7: Mostenire. Clase derivate.

bp=&d;

- (In exemplu) are loc o conversie implicita la tipul pointerului (Baza \*);
- Prin intermediul pointerului este disponibila doar metoda afisare() din Baza;
- Acelasi fenomen apare si in cazul referintelor: daca o functie are ca parametru o referinta la o clasa de baza, functia poate primi ca parametru efectiv un obiect de tip derivat din clasa de baza si il converteste la tipul baza;
- Cand o metoda este chemata prin intermediul unui pointer, **tipul pointerului si NU cel al obiectului catre care se pointeaza** determina ce functie este apelata. (Daca metoda nu e declarata virtual -> C8)

## Observatii

### 1. Supradefinirea operatorilor:

- in general, operatorii implementati ca functii membre se mostenesc
- **operatorul de atribuire NU** se mosteneste

### 2. Constructorii si destructorii

- **nu** se mostenesc

### 3. Functiile **friend** (si implicit operatorii supradefiniti ca functii friend)

- **nu** se mostenesc

### 4. **Static** – in contextul mostenirii:

- functiile membre statice se comporta ca orice functie membra: sunt mostenite in clasa derivata
- **NU** pot sa fie virtuale (C8)

## C7: Mostenire. Clase derivate.



### Tipuri de derivare:

- derivarea este implicit privata: `class Derivata: Baza`
- dar vrem sa fie publica (aproape mereu): `class Derivata: public Baza`
- rar este protected: `class Derivata: protected Baza`

Drept de acces in clasa de baza	Modificator acces	Drept de acces in clasa derivata
public private protected	public	public inaccesibil protected
public private protected	private	private inaccesibil protected
public private protected	protected	protected inaccesibil protected

Mostenirea private este utila daca vreau sa ascund functionalitati ale clasei de baza.

Daca totusi vreau sa fac publice o serie de attribute atunci o sa le declar in sectiunea public a clasei derivate.

Daca se precizeaza numele unei functii, toate versiunile suprascrise ale ei sunt expuse.

```
#include <iostream>
using namespace std;
class Aparat {
public:
    void functia1() { cout<<"functia1"; }
    int functia1(bool i) {if (i) return 1; return 0;}
    void functia2() { cout<<"functia2"; }
    void functia3() { cout<<"functia3"; }
};
```

//un aparat fara functia 3

```
class Aparat_Defect: Aparat { // mostenire private -> tot ce e public in baza -> private in derivata
public: // numele membrilor din clasa de baza facuti publici
    Aparat :: functia1; // in acest caz, ambele implementari ale functiei1 sunt expuse
    Aparat :: functia2;
};
```

```
int main() {
    Aparat_Defect a;
    a.functia1();
    cout<<a.functia1(1);
    a.functia2();
    //a.functia3(); // ERROR: private member function named functia3
    return 0;
}
```

## C7: Mostenire. Clase derivate.

### #pragma once

In cazul in care am ierarhii complicate de clase, trebuie sa am grija sa nu includ headerul unei clase de mai multe ori in alt header sau in programul principal.

Cel mai sigur este ca fiecare header care poate sa fie inclus de mai multe ori sa contina la inceputul implementarii directiva preprocesor :

**#pragma once** ⇔ **include** aceasta sursa o singura data in momentul compilarii

Ex:

```
//Baza.h
#pragma once
#include <iostream>
using namespace std;
class Baza
{protected:
    int atr1;
public:    Baza(int i):atr1(i){}
};
```

```
//Derivata.h
#include "Baza.h"
class Derivata: public Baza
{
    int atr2;
public:    Derivata(int a1, int a2):Baza(a1),atr2(a2){}
};
```

//daca nu includeam directiva in Baza - **ERROR: redefinition of `class Baza'**

```
#include "Baza.h"
#include "Derivata.h"
int main()
{
    //...
}
```

## C7: Mostenire. Clase derivate.

### Shadowing

```
class A
{
    protected: int atr;
    //...
};

class B:public A
{
    //se mosteneste atr din clasa de baza
    int atr; //si mai am un atribut nou ; dar acesta il acopera ca vizibilitate pe cel mostenit
    //...
    //ca sa am acces la atr mostenit din clasa de baza trebuie sa ma refer la el prin A::atr;

    void set_atr(int i, int j)
    {
        A::atr=i;
        atr=j;
    }
};
```

Tratam situatia la fel ca in cazul metodelor ascunse prin redefinire.

## Cand agregam si cand derivam?

- Amandoua mecanisme reutilizeaza codul scris pentru o clasa simpla/de baza intr-o alta clasa mai complexa.
- In ambele cazuri se folosesc liste de initializare pentru constructori pentru a crea obiectele de baza (chiar daca apelul difera putin ca sintaxa), etc – vezi operator= si destructor.

## Si atunci care e diferenta?

### Agregarea/Compozitia

- este folosita cand se doreste reutilizarea unui tip de date A pentru generarea altui tip de date B - **fara a prelua interfata clasei A.**
- se integreaza in clasa mai complexa un atribut (sau mai multe) de tipul clasei mai simple.
- utilizatorul noii clase va vedea doar interfata clasei noi.
- nu va mai fi de interes interfata clasei simple/de baza.

## Derivarea

- este folosita cand **se doreste preluarea interfetei**
- utilizatorul va vedea atat interfata clasei de baza cat si a clasei derivate

**Si cum implementam corect? Mostenind sau agregand?**

**Cea mai buna regula:** Intrebati-va daca urmeaza sa faceti atribuiiri de tipul:

`obj_baza=obj_der;`

daca **DA** - alegeti **derivarea**.

## Observatie

- nu exista nicio diferenta in termeni de :
  - memorie ocupata
  - durata de executie

a unei aplicatii implementate folosind o solutie prin derivare vs. folosind o solutie prin agregare.



## C7: Mostenire. Clase derivate.

### Dezvoltare continua (Incremental development) si mentenanta usoara

- Un avantaj al mostenirii si agregarii este ca putem **sa adaugam cod nou fara a introduce bugguri in codul deja existent.**
- **Erorile** se gasesc in codul nou => vor fi mai **usor de cautat si gasit**, fiind bine izolate.
- Ce s-ar intampla daca am modifica toata sursa veche? => timp crescut pentru depanare.
- Prin derivare dint-o clasa existenta sau compunere cu o clasa existenta si adaugare de noi attribute si functii, **codul existent nu se modifica** – iar acesta **poate sa fie refolosit si in alte locuri (nu doar in clasa derivata sau compusa) neatins/fara erori.**
- **Clasele sunt clar separate.** Nu e nevoie nici macar sa ne uitam in/avem codul vechi.
- Pentru dezvoltare pe baza unor clase sunt suficiente **headerul/interfata si functiile compilate - modulul obiect cu functiile compilate.**

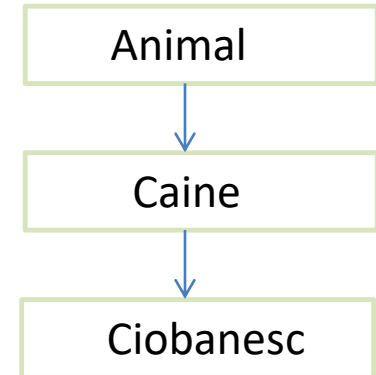
# C7: Mostenire. Clase derivate.

## Un exemplu simplu

```
#include <iostream>
using namespace std;
class Animal //clasa de baza
{
    protected: int varsta; //va fi mostenit in clasele derivate
    public:     Animal(int i=0):varsta(i) {
        }
        //constructorii si destructorul nu se mostenesc
        //operator= nu se mosteneste

        void set_varsta(int i) {
            varsta=i;
        }

        int get_varsta() {
            return varsta;
        }
}; //metodele get_varsta si set_varsta se mostenesc in clasele derivate
```



```

#include "Animal.h"

class Caine: public Animal
{
//mostenesc atributul varsta
protected:
    char *nume; //in plus, am atributul nume

public:

    Caine(){nume=NULL;} //apeleaza Animal()

    Caine(int i, char *n):Animal(i) {
//apelez explicit constructorul Animal(int)
        nume=NULL;
        set_nume(n);
    }

    Caine(const Caine &c):Animal(c) {
//apelez constr. de cop. Animal(const Animal&)
//c este convertit automat la const Animal&
        nume=NULL;
        set_nume(c.nume);
    }

```

```

~Caine(){ //se apeleaza ~Animal
        delete [] nume;
    }

    Caine& operator=(const Caine &c){
        varsta=c.varsta;
        //Animal::operator=(c);
        //(Animal&)(*this)=c;
        set_nume(c.nume)
        return *this;
    }

    char* get_nume() {
        return nume;
    }

    void set_nume(char* n) {
        if (nume!=NULL) delete [] nume;
        if (n!=NULL){
            nume=new char[strlen(n)+1];
            strcpy(nume,n);
        } else nume=NULL;
    } //au fost mostenite metodele get_varsta
}; //si set_varsta

```

```
#include "Caine.h"

class Ciobanesc: public Caine
{
    //mostenesc varsta si nume
    int cate_oi_aduna;
public:

    Ciobanesc(int i, char *n, int ii):Caine(i,n){
        //apeleaz explicit constructorul Caine(int,char*)
        //care il apeleaza pe Animal(int)
        cate_oi_aduna=ii;
    }
}
```

```

Ciobanesc(const Ciobanesc &c):Caine(c) {
    //apeleaz explicit constructorul Caine(const
    //Caine&) care il apeleaza pe Animal(const
    //Animal&)
    cate_oi_aduna=c.cate_oi_aduna;
} //Pot sa nu il implementez ? Da, cel
//generat automat functioneaza corect.
//se mostenesc metodele din clasa de baza:
//set_nume,set_varsta, get_nume, get_varsta
```

```

Ciobanesc& operator=(const Ciobanesc &c){
    Caine::operator=(c);
    cate_oi_aduna=c.cate_oi_aduna;
    return *this;
    //se poate mai scurt?
} //Pot sa nu il implementez? Da, cel
//generat automat functioneaza corect.

int get_cate() {
    return cate_oi_aduna;
}

void set_cate(int i) {
    cate_oi_aduna=i;
}

~Ciobanesc(){
}
};
```

**OBS:** Daca nu am ce spatiu sa eliberez in destructor -> nu il implementez, si nu implementez nici constructorul de copiere si operator=.

```
#include "Ciobanesc.h"
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    Ciobanesc c(3,"bobitza",10);
```

```
    cout<<c.get_varsta()<<endl;
```

```
    cout<<c.get_nume()<<endl;
```

```
    cout<<c.get_cate()<<endl;
```

```
    Ciobanesc d(0,"a",0),dd(d);
```

```
    cout<<dd.get_varsta()<<endl;
```

```
    cout<<dd.get_nume()<<endl;
```

```
    cout<<dd.get_cate()<<endl;
```

```
    d=c;
```

```
    cout<<d.get_varsta()<<endl;
```

```
    cout<<d.get_nume()<<endl;
```

```
    cout<<d.get_cate()<<endl;
```

```
    Animal a(2);
```

```
    a=d;
```

```
    cout<<a.get_varsta()<<endl;
```

```
    //cout<<a.get_nume()<<endl;
```

```
    //cout<<a.get_cate()<<endl;
```

```
    // get_cate si get_nume nu fac parte din  
    //clasa Animal
```

```
    Caine x(1,"labus");
```

```
    x=d;
```

```
    cout<<x.get_varsta()<<endl;
```

```
    cout<<x.get_nume()<<endl;
```

```
    //cout<<x.get_cate()<<endl;
```

```
    // x nu e de tip Ciobanesc get_cate
```

```
    return 0;
```

```
}
```

## Alt exemplu

Care sunt clasele necesare si care sunt relatiile intre ele - daca am urmatoarea problema?

Cineva vrea sa tina evidenta persoanelor dintr-un oras.

O persoana are un nume si locuieste la o adresa (strada, nr).

Ea poate sa fie angajata, caz in care va avea un salariu si lucreaza la o firma.

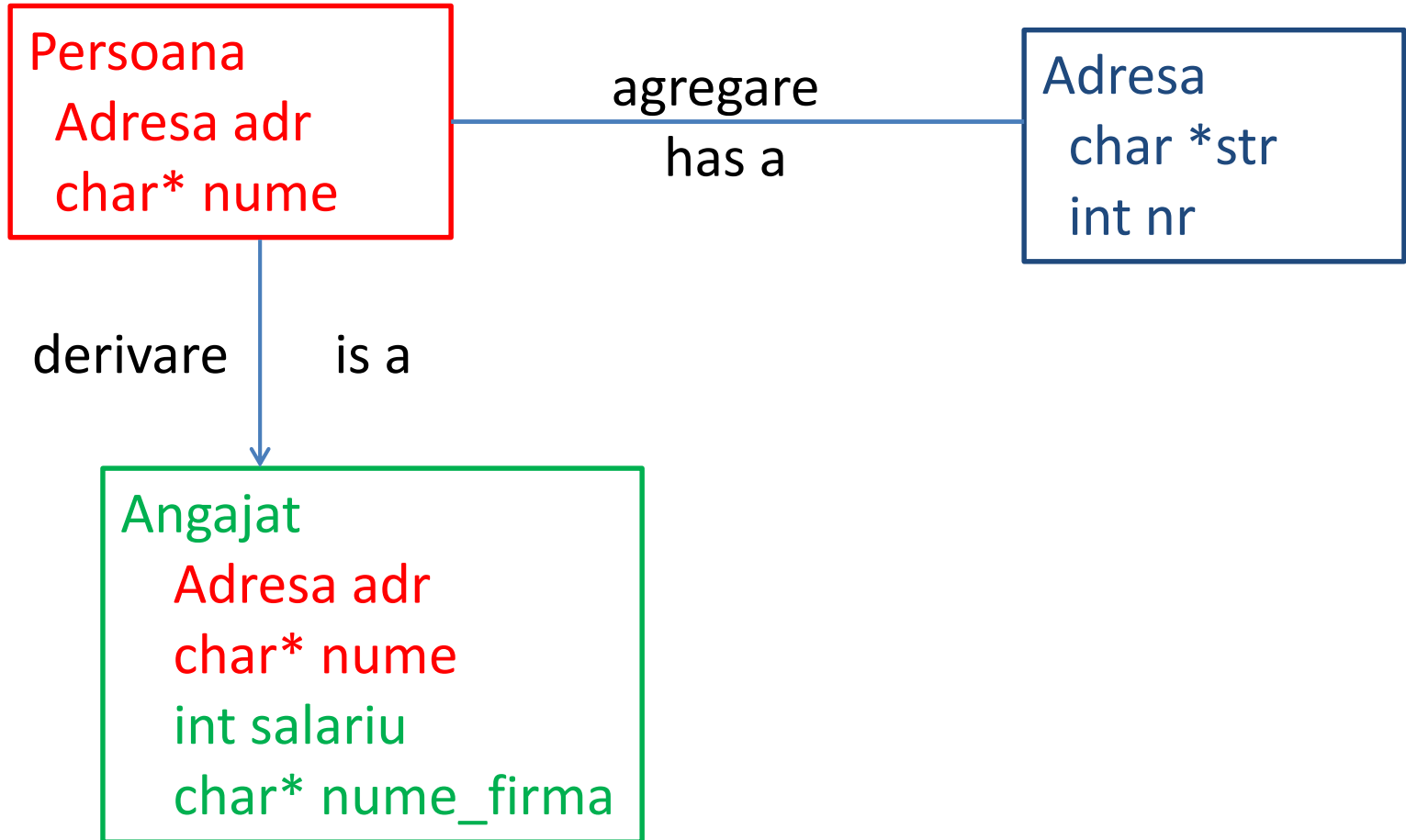
Evidenta se tine sub forma unui vector.

Acesta trebuie sa permita afisarea tuturor datelor despre persoana respectiva: nume, adresa, salariu(daca e cazul).

Ce clase avem?

Care sunt relatiile intre ele?

Ce metode trebuie sa contina? Porniti de la cerinta!!!



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

```
class Adresa {
    private: char* str;
            int nr;
```

```
    public:
```

```
        Adresa() {} //necesar la crearea obiectelor de tip Persoana
```

```
        Adresa(char* c, int n):nr(n) { //necesar in constructorul din Persoana
            //se apeleaza pseudoconstructorul pentru nr de tip int;
            //pseudoconstructorii se folosesc pentru initializarea tipurilor de date de baza
```

```
                str=new char[(strlen(c))+1];
                strcpy(str,c);
        }
```

```
        Adresa(const Adresa&p) {
            //se foloseste la crearea de obiecte de tip Persoana
                str=new char[strlen(p.str)+1];
                strcpy(str,p.str);
                nr=p.nr;
            //sau: str=NULL; *this=p; // folosesc operator= implementat pentru Adresa
        }
```



```

Adresa& operator=(const Adresa &p) {
    //in Persoana se foloseste atribuirea de obiecte de tip Adresa
    if(this!=&p) {
        if (str!=NULL) delete [] str;
        if (p.str==NULL) str=NULL;
        else {
            str=new char[strlen(p.str)+1];
            strcpy(str,p.str);
            nr=p.nr;
        }
    }
    return *this;
}

```

```

friend ostream& operator<<(ostream &dev,const Adresa &p){
    // in operatorul de << din Persoana se foloseste operatorul de << din Adresa
    dev<<"Strada: "<<p.str<<endl;
    dev<<"Nr. : "<<p.nr<<endl<<endl;
    return dev;
}

```

```

~Adresa() {
    delete [] str;
}

```

```

};

```

```

class Persoana {
    protected: Adresa adr; //agregare
               char *nume;

    public: Persoana(){} //apel automat adr()

    Persoana(const Adresa &d, char* n):adr(d) { //construiesc atributul adr de tip Adresa
        nume= new char[strlen(n)+1]; // altfel se apela automat adr()
        strcpy(nume,n);
    }

    Persoana(char* s, int nr, char* n):adr(s,nr) { //construiesc atributul atr de tip Adresa
        nume= new char[strlen(n)+1];
        strcpy(nume,n);
    }

    Persoana(const Persoana & p):adr(p.adr) { //construiesc atributul atr de tip Adresa
        nume= new char[strlen(p.nume)+1];
        strcpy(nume,p.nume);
    }
}

```

//Tema:

//implementati o functie set\_nume care face toate verificarile necesare si utilizati-o acolo

//unde e nevoie

```

Persoana& operator=(const Persoana & p) {
    adr = p.adr;
    //atribuire intre obiecte de tip Adresa folosind Adresa::operator=
    nume= new char[strlen(p.nume)+1];
    strcpy(nume,p.nume);
    return *this;
}

```

Daca nume era declarat de tip string, ce trebuia implementat?

Cum ati fi implementat constructorul cu parametrii?

```

void afis(){
    if (nume!=NULL) cout<<"Nume: "<<nume<<endl;
    cout<<"Adresa: "<<endl;
    cout << adr;
    //afisare obiect de tip Adresa folosind operator<< din Adresa
}

//sau: se supradefineste operator<<

```

```

~Persoana(){
    delete [] nume;
} //se apeleaza automat destructorul din Adresa

```

```
};
```



Reutilizare cod  
prin agregare  
(compunere)

```

class Angajat:public Persoana {
private:
    //mosteneste toate attributele din Persoana si are in plus
    double salariu;
    char *nume_firma;
public:
    Angajat() {nume_firma =NULL;//apel automat constructor Persoana()
    }

    Angajat(const Adresa &d, char* n,double s, char*numf):Persoana(d,n),salariu(s) {
        //apelez constructorul clasei de baza – Persoana - cu parametri
        nume_firma= new char[strlen(numf)+1];
        strcpy(nume_firma, numf);
    }

    Angajat(char* s, int nr, char* n, double sal, char *numf):Persoana(s, nr, n),salariu(sal) {
        nume_firma= new char[strlen(numf)+1];
        strcpy(nume_firma, numf);
    }

    Angajat(const Angajat & p):Persoana(p) {//apel explicit constructor copiere din Persoana
        nume_firma= new char[strlen(p.nume_firma)+1];
        strcpy(nume_firma, p.nume_firma); //nume_firma ar trebui setat cu o metoda
        salariu=p.salariu;                  // speciala. Tema: Implementati!
    }

```

**Angajat& operator=(const Angajat & p) {**

**//vreau sa reutilizez codul scris sau generat automat in clasa de baza pentru =**

**(Persoana&)(\*this)=p;**

**// convertesc obiectul de la adresa lui this la tipul de date Persoana&**

**//folosesc operatorul de atribuire din clasa Persoana ; automat p este**

**//convertit la tipul de date de baza; puteam sa scriu si (Persoana)p;**

**//sau**

**// Persoana::operator=(p);**

**nume\_firma= new char[strlen(p.nume\_firma)+1];**

**strcpy(nume\_firma, p.nume\_firma);**

**salariu=p.salariu;**

**return \*this;**

**}**

**~Angajat(){ delete [] nume\_firma;} // se apeleaza automat si ~Persoana**

**void afis(){ //redefinire**

**((Persoana)(\*this))->afis();**

**// il convertesc pe this la tipul de date Persoana\***

**//apelez functia afis din Persoana**

**//sau Persoana::afis();**

**cout<<"Firma: "<<nume\_firma<<endl;**

**cout<<"Salariu: "<<salariu<<endl;**

**}**

**};**

 Reutilizare cod  
derivare

```
int main()
{
    //ideea
    Persoana *c;
    Adresa a("str",10);
    Angajat ang(a,"ang",10,"HP");
    c=&ang;
    //un pointer de tip Persoana poate sa pointeze catre o zona de memorie de tip Angajat
    ((Angajat*)c)->afis();
    //convertesc ce gasesc la adresa la care pointeaza c - la un pointer de tip Angajat
    //apelez functia de afisare din clasa Angajat => str, 10 ang 10

    int n;
    cin>>n;
    char *num=new char[50];
    char *numf=new char[50];
    char *str=new char[50];
    int nr;
    double sal;

    bool *t=new bool[n];
    //un vector in care memorez daca persoana e de tip angajat sau nu
```

```
Persoana **evidenta=new Persoana*[n];
```

```
//un vector in care vreau sa stochez adrese ale obiectelor de tip Persoana sau Angajat
```

```
for (int i=0;i<n;i++){  
    cout<<"Adaugati salariat sau nesalariat?";  
    bool tip;cin>>tip;  
    t[i]=tip;  
    if (tip==true) {  
        cout<<"Nume: "; cin>>num;  
        cout<<"Str. "; cin>>str;  
        cout<<"Nr."; cin>>nr;  
        cout<<"Nume firma: "; cin>>numf;  
        cout<<"Salariu."; cin>>sal;  
        evidenta[i]=new Angajat(str,nr,num,sal,numf);  
    }else  
    {  
        cout<<"Nume: "; cin>>num;  
        cout<<"Str. "; cin>>str;  
        cout<<"Nr."; cin>>nr;  
        evidenta[i]=new Persoana(str,nr,num);  
    }  
}
```

```

cout<<"_____ "<<endl;
for (int i=0;i<n;i++)
    if (t[i]==true) {
        ((Angajat*)(evidenta[i]))->afis();
        cout<<"_____ "<<endl;
    }else {
        evidenta[i]->afis();
        cout<<"_____ "<<endl;
    }

return 0;
}

```

//”Tema”: implementati operatorii << in clasele Persoana si Angajat si folositi-i!

**//complicat**

**//cum arata vectorul evidenta in memorie?**

**//o sa vedem la C8 cum facem asta mai simplu**