C7: Functii virtuale. Suprascriere. Polimorfism

Cuprins:

- Legarea statica (timpurie) - Early binding
- Functii virtuale
- Suprascrierea functiilor (override)
- Legarea dinamica (tarzie) - Dynamic binding
- Polimorfism

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

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(){   Baza::afisare();  cout<< " atr2 = " << atr2<<endl;}//redefinire
};

int main(int argc, char *argv[])
{
   Baza *bp=new Baza(1);
   bp->afisare();                    //atr1=1
   bp=new Derivata(2,2);
   bp->afisare();    //atr1=2        // afisare() de tipul pointerului adica Baza::afisare()
   ((Derivata*)bp)->afisare();       // afisare() de tipul pointerului adica Derivata::afisare()
                                     // atr1=2 atr2=2
   return 0;
}
```

C7: Functii virtuale. Suprascriere. Polimorfism

Legarea – Binding

= conectarea unui apel de functie cu functia in sine (adresa functiei).

Exista doua tipuri de legare:

- **statica/timpurie** (la compilare)
- **dinamica/tarzie** (in momentul executie)

Legare statica/timpurie (early binding)

- se realizeaza la compilare, inainte de rularea aplicatiei
- de catre compilator si editorul de legaturi (linker)
- **in limbajul C** – toate apelurile de functii presupun realizarea unei legaturi statice

```
functie(); // se face legatura intre apel si implementarea functiei
```

- in plus, **in C++** avem functii membre**:**

1. - functiile membre ale unei clase primesc adresa obiectului care face apelul:

```
tip_date obj;
obj.functie();
```

- in functie de tipul obiectului de la acea adresa - compilatorul si editorul de legaturi stabilesc daca:

  - functie() e membra a clasei tip_date (e declarata)
  - este implementata
  - daca e implementata, se face legarea statica si se poate apela acea functie
  - daca functia era declarata ca membra a clasei, dar nu e implementata

[**Linker error**] undefined reference to `tip_date::functie()'

2. - in cazul apelului prin intermediul pointerilor :

```
tip_date_baza *obj=new tip_date_baza;
obj->functie();
```

se intampla acelasi lucru:

- compilatorul si editorul de legaturi stabileasc daca functia invocata de un pointer este de tipul acelui pointer(adica daca aceasta poate fi apelata)
- mai exact, compilatorul foloseste **tipul static al pointerului** (tipul de care a fost declarata variabila) pentru a determina daca invocarea functiei membre este legala

Este firesc, ca, in cazul in care pointerul poate sa foloseasca acea functie, orice obiect derivat adresat de acel pointer sa o poata utiliza la randul sau

```
tip_date_derivat d;
obj=&d;
obj->functie();  //tipul pointerului este tip_date_baza
```

In programul anterior:

```
Baza *bp=new Baza(2);
//OBS: bp are tipul static: Baza si tipul dinamic: Baza
//tipul dinamic = tipul obiectului catre care se pointeaza la runtime
//poate sa difere de la un moment la altul, se schimba in timpul executiei
bp=new Derivata(1,1);
//OBS:  bp are tipul static: Baza si tipul dinamic: Derivata

bp->afisare();
//se apeleaza Baza::afisare(); tipul static al pointerului fiind Baza
```

- legatura intre numele functiei si corpul ei se face in mod static (in functie de tipul static al lui bp)

- in acest caz se apeleaza functia afisare() din clasa Baza.

DAR daca vrem sa apelam functia de afisare pentru obiectul catre care pointeaza bp

=> trebuie sa facem o conversie explicita a pointerului bp din (Baza*) in (Derivata*):

((Derivata*)bp)->afisare();

astfel incat legatura sa se faca pentru tipul de date al obiectului catre care pointeaza bp.

- aceasta **solutie nu e robusta**

- trebuie mereu sa ne punem problema catre ce tip de obiect pointeaza pointerul de tip clasa de baza si sa facem conversii explicite pentru a apela functia dorita

- solutia e predispusa **erorilor logice**

Alternativa pusa la dispozitie in C++ este mecanismul de **legare dinamica/tarzie(dynamic /late binding) -** nu trebuie sa memorez catre ce tip de obiect se pointeaza in timpul executiei

C7: Functii virtuale. Suprascriere. Polimorfism

Legare dinamica/tarzie

- **legarea se face dupa compilare, la rulare**
- **in functie de tipul dinamic al pointerului (tipul obiectului catre care se pointeaza)**
- **se poate realiza doar in contextul apelului de functii prin intermediul pointerilor**

Pentru a folosi un astfel de mecanism, **compilatorul trebuie sa fie informat** ca exista posibilitatea ca, la rulare, sa se doreasca apelarea unei functii de tipul dinamic al obiectului.

Deoarece nu se cunoaste inca tipul obiectului catre care se va pointa de-a lungul executie programului, **compilatorul si linkerul nu trebuie sa implementeze legatura statica.**

Pentru aceasta este nevoie de un semnal introdus in cod - > **functii virtuale**.

Legarea dinamica poate fi implementata doar in cazul limbajelor compilate!

In Java nu se pune problema asa. Vom vedea de ce.

```
#include <iostream>
using namespace std;
class Baza
{protected: int atr1;
  public:     Baza(int i):atr1(i){}
              virtual void afisare() {  cout << "atr1 = " << atr1<<endl;  }
}; //legarea pentru functia afisare se face la rulare

class Derivata: public Baza
{             int atr2;
 public:    Derivata(int a1, int a2):Baza(a1),atr2(a2){}
            void afisare()  {Baza::afisare(); cout << "atr2 = " << atr2<<endl;} //supraincarcare
}; //legarea pentru functia afisare se face la rulare; e virtuala

int main(int argc, char *argv[])
{
   Baza *bp=new Baza(1);   //bp – tip static Baza, tip dinamic Baza
   bp->afisare();              //legare dinamica (in functie de tipul dinamic)
   cout<<endl<<"_____________________"<<endl;
   Derivata *d=new Derivata(1,1);   //
   bp=d;                       //bp pointeaza catre o adresa  unde se gaseste  un obiect de
                               //tip Derivata
                               //in acest caz bp are tipul static Baza si tipul dinamic Derivata
bp->afisare();       //legare dinamica (in functie de tipul dinamic), apel Derivata::afisare()
   delete bp;  /*ce se intampla?*/ return 0;}
```

C7: Functii virtuale. Suprascriere. Polimorfism

Functii virtuale; supraincarcarea/suprascrierea functiilor

Ce este o functie virtuala?

- functiile virtuale sunt cea mai importanta caracteristica a limbajului C++ (pdpv al POO)

- functiile virtuale permit claselor derivate sa inlocuiasca implementarea metodelor din clasa de baza – **suprascriere/supraincarcare/override** si pun la dispozitie mecanismul de legare dinamica.

Suprascrierea/supraincarcarea functiilor

Presupune ca intr-o **ierarhie** de clase sa avem:

1. functii cu acelasi nume, semnatura si tip returnat in clasa de baza si derivate

2. dar **cu implementari specifice clasei** din care fac parte (ganditi-va la afisare() din Baza si Derivata)

3. functii declarate ca **virtuale**

Daca 1,2,3 =>functii supraincarcate

Daca 1 si 2 – functii redefinite

OBS: Este suficient ca in clasa de baza sa se declare ca functia e virtuala, ceea ce semnaleaza ca in intreaga ierarhie, in cazul in care functia e suprascrisa, ea este virtuala.

In cazul functiilor virtuale:

- compilatorul se asigura ca **implementarea chemata** este cea **pentru tipul dinamic** al obiectului care face apelul.

Compilatorul nu are cum sa prevada catre cine pointeaza bp de-a lungul implementarii - asa ca nu stie (pana in momentul rularii) care functie de afisare ar trebui chemata.

Insa este informat sa nu implementeze o legatura timpurie intre apelul afisare() si corpul functiei de afisare din Baza, ci sa astepte pana in momentul rularii prin cuvantul cheie **virtual** pus in fata functiei afisare().

Astfel, prin **legarea dinamica** sau tarzie (late,dynamic binding), la executiei se va decide care e tipul dinamic al pointerului si se va face legarea cu implementarea din clasa precizata de tipul dinamic.

Legarea dinamica se poate face doar folosind functii virtuale si pointeri.

OBS: Intr-o ierarhie de clase este normal ca obiectele rezultate (de tip baza si derivate) sa aiba comportamente similare - specializate in functie de tipul clasei.

```
class A
{protected: int atr1;
 public: virtual void afisare() {cout<<atr1;}//constr si alte metode
};
class B: public A
{protected: int atr2;
 public:  void afisare() {A::afisare(); cout<<atr2;} //constr si alte metode
};
class C: public B
{protected: int atr3;
 public:  void afisare() {B::afisare(); cout<<atr3;} //constr si alte metode
};
```

Ex: In clasele derivata vrem sa afisam toate atributele acelei clase, nu doar cele mostenite din clasa de baza; functia se numeste tot afisare si va ascunde implementarea din clasa de baza, si va avea un comportament un pic diferit.

Mecanismul este important deoarece, cand o sa fac un apel de functie, o sa scriu mereu afisare() indiferent daca e vorba de un obiect de tip clasa de baza sau de tipul derivatelor.

Observatii:

1. **Functiile membre nevirtuale sunt tratate static** – in momentul compilarii : in functie de tipul static al pointerului se vor apela metodele specifice acelui tip static de date.

2. **Functiile membre virtuale sunt tratate dinamic** – in momentul rularii : in functie de tipul dinamic al obiectului se face legatura apel functie-corp functie.

Observati ca aceasta presupune exclusiv apelul prin intermediul pointerilor.

Daca o functie virtuala nu e chemata prin intermediul unui pointer, legarea e statica.

C7: Functii virtuale. Suprascriere. Polimorfism

Legarea dinamica – “behind the curtains”

Dupa cum am spus in C6 - marea parte a “muncii” - cand vine vorba de derivare - este facuta de compilator, acelasi lucru se intampla si cand declaram o functie membra a unei clase – ca virtuala.

Cuvantul cheie virtual informeaza compilatorul sa nu realizeze o legatura statica.

In schimb, trebuie sa puna in actiune toate mecanismele necesare realizarii de legaturi dinamice.

Pentru aceasta compilatorul creeaza **un tabel** numit **VTABLE pentru fiecare clasa** care contine **macar o functie virtuala**.

In **VTABLE** sunt puse **adresele functiilor virtuale** ale acelei clase.

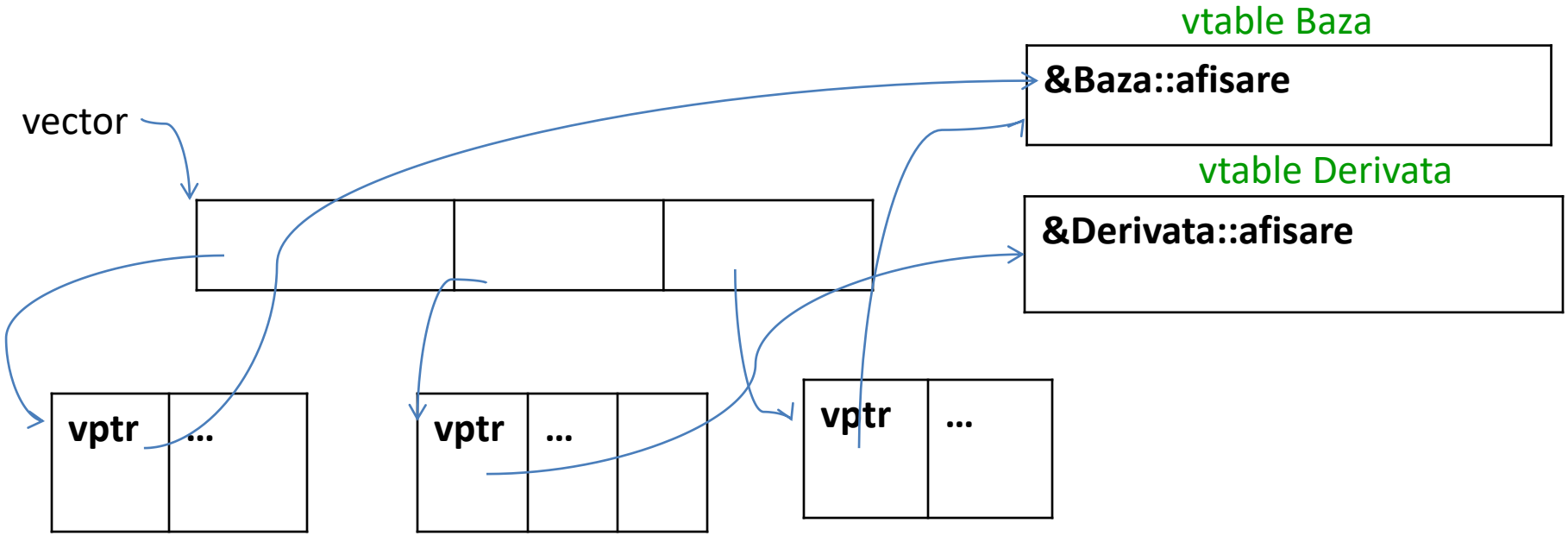
Fiecare obiect de tipul clasei cu functii virtuale va avea in posesie **un pointer** numit vpointer (sau **VPTR**) **catre adresa lui VTABLE**.

Cand se face un apel catre o functie virtuala printr-un pointer de tipul clasei de baza – compilatorul acceseaza VPTR-ul obiectului catre care se pointeaza si, prin intermediul lui, cauta adresa functiei in VTABLE si face legatura (implementeaza legatura tarzie).

Ex:

Baza **vector=new Baza*[3]; //Baza *vector [3];

vector[0]=new Baza(1); vector[1]=new Derivata(2,2); vector[2]= new Baza(3);




```
for (int i=0;i<3;i++)
      vector[i]->afisare();
```

Observatii in ce priveste dimensiunea tipurilor de date care au functii virtuale si viteza de apel a functiilor virtuale

1.Pentru exemplele anterioare testam **dimensiunea obiectelor**:

- in cazul in care **nu am functia de afisare declarata virtuala**:

```
Baza b; Derivata d;
   cout<<sizeof(b)<<endl;    //4
   cout<<sizeof(d)<<endl;    //8
```

In cazul in care **am functia de afisare declarata virtual**:

```
cout<<sizeof(b);              //8
cout<<sizeof(d);              //12
```

In acest caz avem stocat in plus, pentru fiecare obiect, pointerul VPTR.

Dimensiunea obiectelor creste la fel de mult (cu o adresa) - indiferent cate functii virtuale am.

2. Creste oare durata de executie a unei aplicatii daca avem functii virtuale?

Da (nu cu mult), pentru a se face legarea cu adresa functiei - mai am un pointer intermediar (vptr).

Observatii privind crearea VTABLE si initializarea VPTR

Crearea VTABLE –ului pentru clase si initializarea VPTR –ului este automata (facuta de compilator).

Cand se face initializarea VPTR-ului?

In momentul crearii unui obiect. Mai exact la apelul constructorului.

In cazul in care nu se implementeaza niciun constructor, cel generat automat face si aceasta initializare.

Observati inca o data importanta generarii automate a constructorilor.

Ce se intampla daca avem urmatoarea ierarhie de clase si avem un vector:

Ex: Baza **vector=new Baza*[3];
vector[0]=new Baza(1);
vector[1]=new Derivata1(1,2);
vector[2]= new Derivata2(2,3,4);

for (int i=0;i<3;i++) vector[i]->afisare(); //ok

vector[1]->cit_val();
// 'class Baza' has no member named 'cit_val'
// s-a trunchiat VTABLE la dimensiunea celui de tip Baza
// compilatorul previne dezastrul
// oricum nu as putea sa apelez: vector[0]->cit_val();

```
class Baza
{ protected: atr1;
  public:
      Baza(int);
      virtual void afisare();
      virtual void nume();
};
```

```
class Derivata1:public Baza
{ protected: atr2;
  public:
      Derivata(int,int);
      void afisare();
      void nume();
      virtual void cit_val();
};
```

```
class Derivata2;public Derivata1
{ protected: atr3;
  public:
      Derivata2(int,int,int);
      void afisare();
      void cit_val();
};
```

VTABLE-urile pentru fiecare clasa sunt:

Baza	Derivata1	Derivata2
&Baza::afisare	**&Derivata1::afisare**	**&Derivata2::afisare**
&Baza::nume	**&Derivata1::nume**	**&Derivata1::nume**
	&Derivata1::cit_val	**&Derivata2::cit_val**

C7: Functii virtuale. Suprascriere. Polimorfism

1. Nu pot avea constructori virtuali.

De ce?

VPTR este initializat in constructor.

Constructorul, daca era functie virtuala, trebuia sa aiba adresa accesibila prin VPTR, dar VPTR inca nu e initializat.

2. La apelul unei functii virtuale in constructor - se apeleaza versiunea locala a functiei.

Majoritatea compilatoarelor – in clipa in care depisteaza un apel de functie virtuala in constructor, implementeaza imediat mecanismul de legare statica.

3. Initializarea VPTR-ului se face pentru VTABLE-ul clasei respective.

4. Destructori

- daca intr-o clasa avem minim o functie virtuala, se presupune ca avem intentia sa folosim mecanismul de legare dinamica (care la randul lui implica utilizarea de pointeri).

- o sa avem obiecte create dinamic, pentru care **TREBUIE** sa eliberam noi spatiu.

Ex:

```
Baza **vector=new Baza*[3];
vector[0]=new Baza(1);
vector[1]=new Derivata1();
vector[2]= new Derivata2();
delete [] vector;
```

- Pentru exemplul de pana acum, in care nu implementam niciun destructor, se generau automat destructori pentru toate clasele din ierarhie.
- Deci se va apela de 3 ori destructorul pentru tipul de date static al elementelor vector[i], adica se apeleaza de 3 ori destructorul de tip Baza

DAR: noi trebuie sa eliberam spatiu pentru obiecte de tip Derivata1 si Derivata2.

- Pentru a se putea face acest lucru, in clasa Baza trebuie sa declaram destructorul ca fiind functie virtuala:

virtual ~Baza(){}

- astfel toti destructorii claselor derivate vor avea comportament de functii virtuale, chiar daca sunt generati automat de compilator.

REGULA : Daca avem minim o functie virtuala intr-o clasa - declaram destructorul virtual

- **La fel ca in cazul constructorilor, mecanismul virtual nu este luat in considerare in destructor: se poate apela doar varianta locala a unei functii virtuale.**

C7: Functii virtuale. Suprascriere. Polimorfism

Virtual si static

- functiile statice NU pot sa fie virtuale

- deoarece functiile statice nu sunt apelate prin intermediul unui obiect

- nu primesc adresa unui obiect si astfel nu au acces la VPTR.

C7: Functii virtuale. Suprascriere. Polimorfism

Polimorfism si functii virtuale

- “poli” – mai multe; “morf” –forma

Polimorfismul se poate realiza prin:

- **supradefinirea functiilor** - functii cu acelasi nume, dar semnaturi diferite - care se comporta diferit in functie de context – in functie de modul in care sunt apelate – chiar daca au acelasi nume; ***polimorfism ad hoc***

- **suprascrierea functiilor** – functii virtuale: **acelasi pointer poate sa aiba comportamente diferite la apelul unei metode, in functie de tipul lui dinamic**

C7: Functii virtuale. Suprascriere. Polimorfism

Daca sunt asa de importante, de ce nu folosim exclusiv functii virtuale? De ce nu realizeaza C++ decat legaturi dinamice?

- pentru ca legarea dinamica nu este la fel de eficienta ca legarea statica (asa ca o folosim doar cand e nevoie)

❖ C/C++ este un limbaj preocupat de eficienta

Daca nu sunt la fel de eficiente - cand folosim functii virtuale?

De fiecare data cand vrem ca in clasa derivata sa modificam/adaptam/suprascriem comportamentul unei functii deja implementate in clasa de baza.

C7: Functii virtuale. Suprascriere. Polimorfism

Cand este important acest mecanism?

1. Functiile virtuale sunt folosite pentru a implementa polimorfismul in momentul rularii (" Run time Polymorphism“).

2. Un alt avantaj al functiilor virtuale este ca permit realizarea de **liste neomogene** de obiecte (exemplul de la finalul cursului C6)

3. Dezvoltarea de biblioteci in spiritul POO.

---

2. Liste neomogene:

Tipul static al obiectelor din lista este tipul de baza*, dar obiectele adresate pot fi de orice tip din ierarhie.

Operatiile efectuate pe lista pot fi facute fara testarea tipului obiectului, deoarece exista garantia ca metodele apelate sunt cele de tipul dinamic al obiectelor.

```
//Baza.h
#pragma once
#include <iostream>
using namespace std;

class Baza
{protected:
            int atr1;
  public:
    Baza();
    Baza(int );
    void set_atr1(int );
    virtual void afisare();
    virtual ~Baza(){};
};
```

```
//Baza.cpp
 #include "Baza.h"
  Baza::Baza()  {  }

  Baza::Baza(int i):atr1(i)  {  }

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

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

```
//Derivata.h
#pragma once
#include "Baza.h"
class Derivata: public Baza
{
 protected:              int atr2;
 public:
   Derivata();
   Derivata(int , int );
   void set_atr2(int );
   void afisare(); //afisare din Derivata e virtuala
};//destructorul generat automat e virtual
```

```
//Derivata.cpp
#include "Derivata.h"
    Derivata::Derivata()    {    }

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

   void Derivata::set_atr2(int n)    {        atr2 = n;      }

   void Derivata::afisare()
   {       Baza::afisare();
           cout << "atr2 = " << atr2<<endl;
   }
```

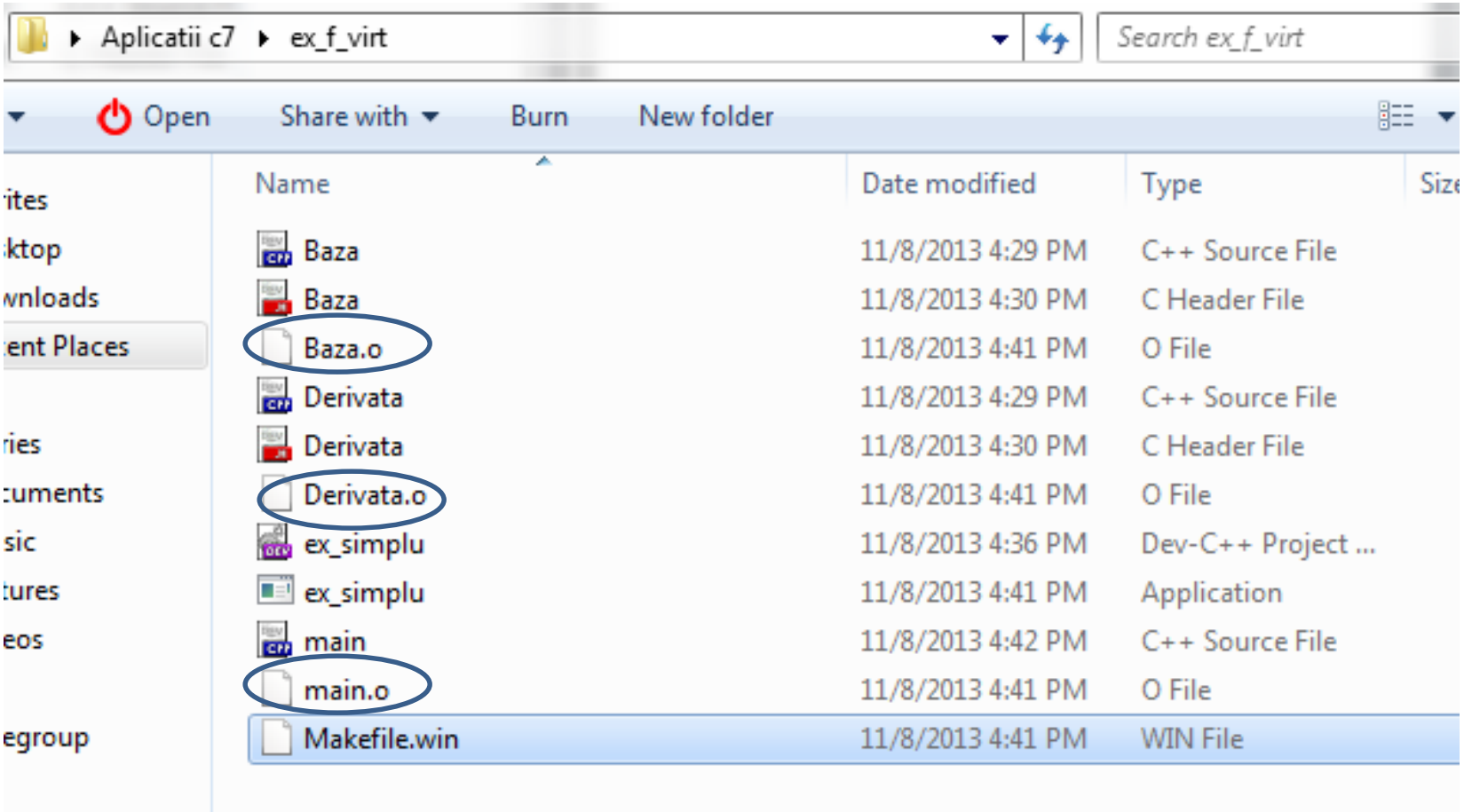
```
#include "Derivata.h"

int main(int argc, char *argv[])
{
   int n;
   cout<<"Dati dimensiunea";
   cin>>n;
   Baza **vec=new Baza*[n];
   for (int i=0;i<n;i++){
      cout<<"Introduceti obiect de tip Baza(0) sau Derivata(1)?";
      int tip;
      cin>>tip;
      if (tip==0){
                cout<<"Dati atr1:";
                int a1;
                cin>>a1;
                vec[i]=new Baza(a1);

      }else if (tip==1){
                              cout<<"Dati atr1 si atr2:";
                              int a1,a2;
                              cin>>a1; cin>>a2;
                              vec[i]=new Derivata(a1,a2);
       } else i--;
  }
```

```
 for (int i=0;i<n;i++)
        vec[i]->afisare();
    //comportament polimorf; nu mai testez eu care e tipul si nu fac conversii explicite


    //Baza *b=new Derivata(1,1);
    // b->set_atr2(2);


   //atentie, set_atr2 nu e o functie din Baza


    //ERROR:'class Baza' has no member named 'set_atr2'


     return 0;
}
```

In exemplul anterior am implementat clasa Baza si Derivata separand headerul de implementare.

La compilare au fost generate **fisierele .o – modulele obiect care contin sursele compilate** pentru implementarile functiilor din Baza si Derivata.

OBS: Daca implementarea era facuta in headere sau daca toate clasele erau in acelasi program, atunci nu am fi avut module obiect distincte.



3. Sa consideram urmatorul scenariu:
- trebuie implementata **o ierarhie de clase**
- sa consideram varianta in care, pentru fiecare clasa se defineste o functie membra **Afisare() nevirtuala**.

Aceasta implementare are un dezavantaj major din punct de vedere al dezvoltarii ulterioare a aplicatiilor: **intreaga ierarhie de clase trebuie recompilata** daca **se mai adauga o noua clasa cu o functie membra Afisare** (care se doreste virtuala), si aceasta indiferent daca implementarile sunt in fisiere sursa diferite sau nu.

Acesta este un dezavantaj major, pe de-o parte datorita efortului de compilare, dar si din ratiuni de marketing.

Sa presupunem ca un producator software vrea sa implementeze ierarhia de clase discutata mai sus si sa o vanda.

Cumparatorul / utilizatorul doreste ca pe baza claselor din biblioteca, sa-si dezvolte propriile aplicatii, pastrand insa functia Afisare in implementarea lor. Si mai mult doreste ca functia de Afisare sa fie virtuala (ceea ce inseamna ca ar trebui sa modifice headerul clasei de baza din ierarhie si sa recompileze toate clasele).

Singura solutie ar fi ca producatorul sa puna la dispozitia utilizatorului modulele sursa ale bibliotecii, ceea ce evident nu este de acceptat.

De ce este insa necesara recompilarea tuturor implementarilor (a tuturor claselor)?

Explicatia provine de la modurile standard in care compilatorul poate distinge intre functii cu acelasi nume (pana acum am vazut 3 asemenea moduri):

- functiile difera prin lista de argumente
- se foloseste operatorul de rezolutie ::, prin care se distinge intre functii redefinite in clasele derivate
- se face distinctie dupa tipul obiectului pentru care a fost apelata functia.

Toate aceste distinctii (rezolutii) se fac insa la compilare, legare timpurie (statica).

In clipa in care declaram functia de afisare virtuala, trebuie compilate inca o data toate clasele, iar **mecanismul de legare statica este inlocuit peste tot pe unde apare functia Afisare cu cel de legare dinamica.**

In plus compilatorul va genera VTABLE-urile pentru clasele din ierarhie si va initializa VPTR pentru obiectele folosite.

In mod normal, utilizatorului trebuie sa i se furnizeze niste fisiere .h (cu definitiile claselor) si niste fisiere .obj , .o sau .lib (cu implementarile gata compilate), pe baza carora sa dezvolte noi tipuri de date, sa deriveze noi clase din cele existente deja .

Prin legarea tarzie/dinamica (late sau dynamic binding), adica prin intermediul functiilor membre virtuale se poate rezolva problema furnizarii surselor.

O functie virtuala Afisare "ascunsa" intr-o clasa de baza B precompilata nu este legata la obiectele din clasa B ca o functie membra obisnuita.

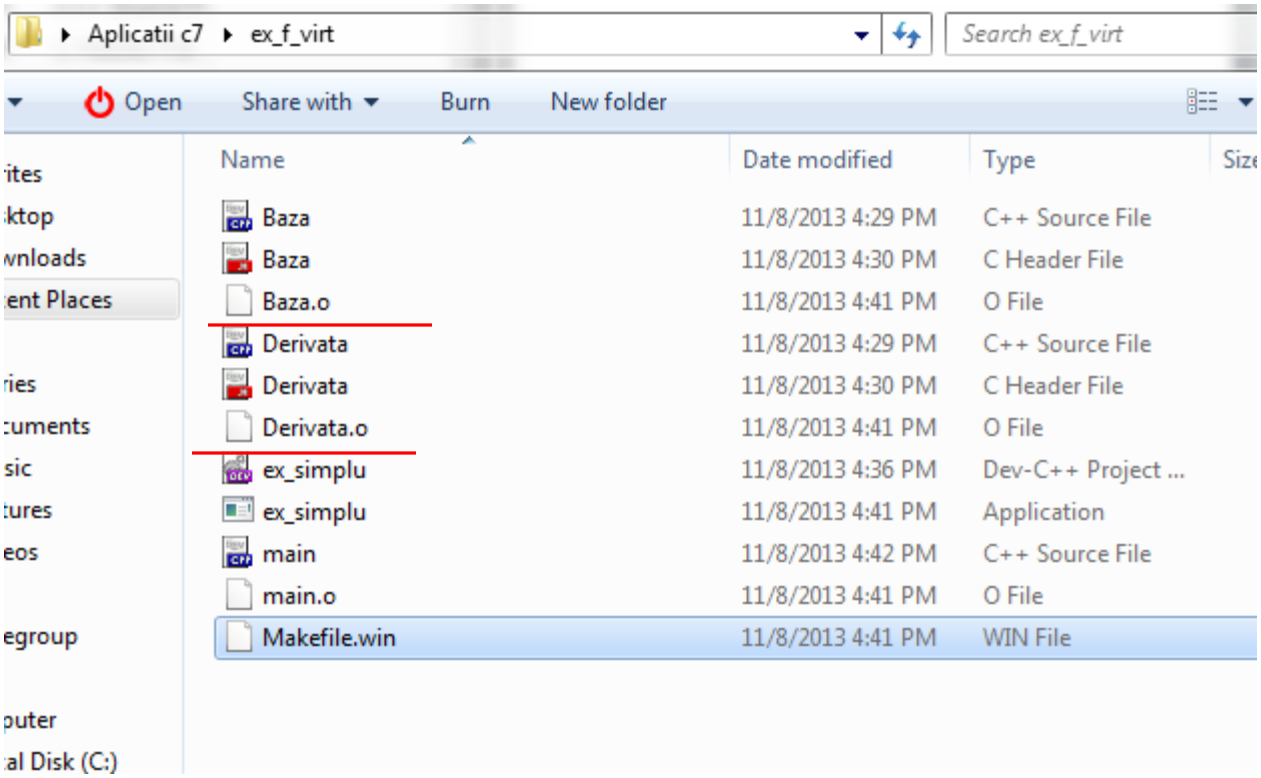
Putem crea o clasa D derivata din B si suprascrie o functie Afisare pentru clasa D.

Daca se compileaza modulul nou de program si se link-editeaza cu modulul .O/.OBJ sau .LIB anterior dezvoltat, apelurile functiei Afisare se vor face corect, fie pentru obiectele din clasa B, fie pentru cele din clasa D.

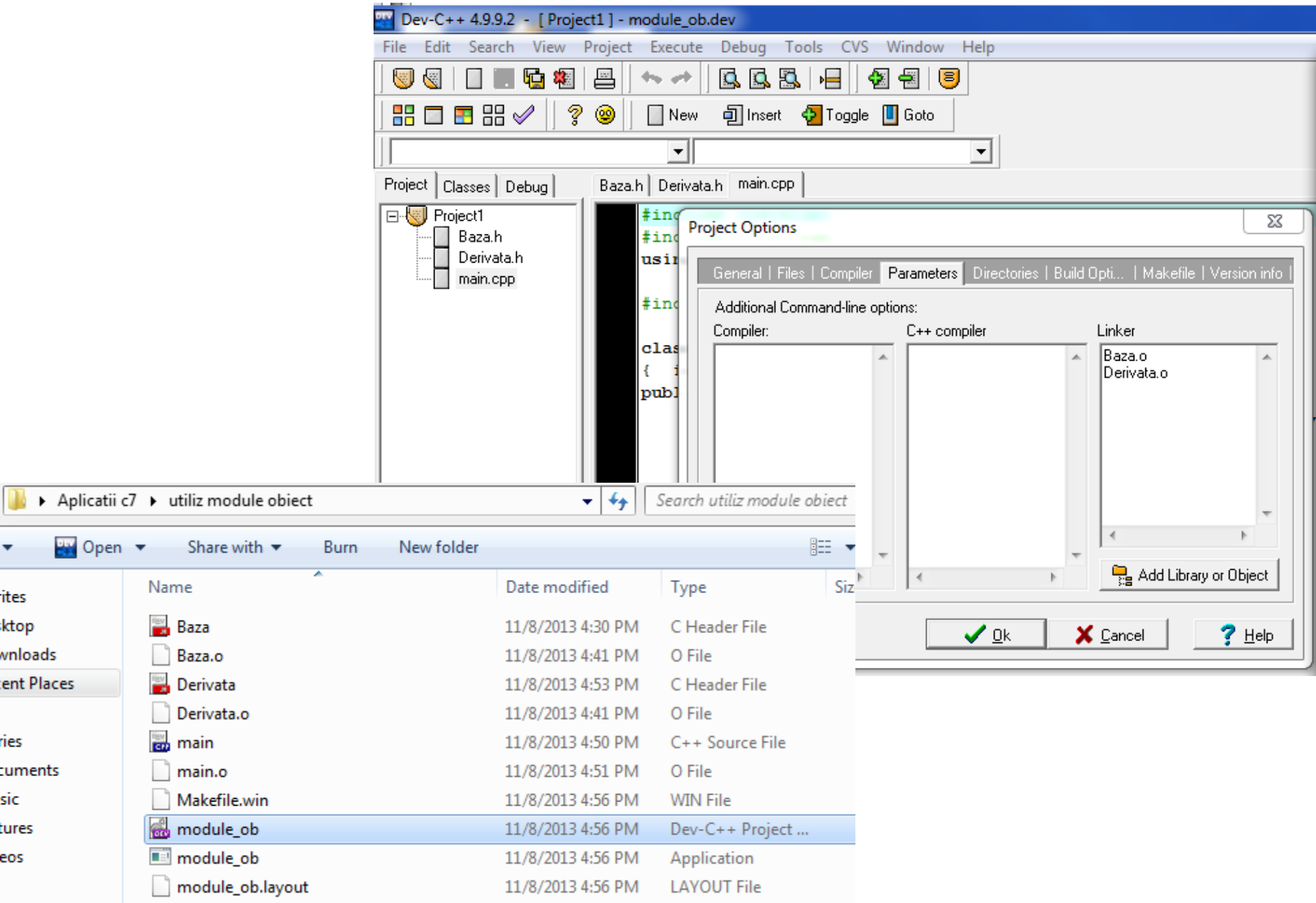
Functia Afisare din clasa D devine si ea functie virtuala (daca are acelasi tip si aceeasi lista de argumente).

Daca se doreste dezvoltarea in continuare a acestei ierarhii, nu mai este nevoie sa recompilez mereu codul pentru clasele Baza si Derivata, atata timp cat nu modific implementarea sau headerele, ci trebuie doar sa includ headerele si modulele obiect in noul proiect.

Daca aplicatia avea dimensiuni mult mai mari, se salva mult timp – aferent recompilarii.



Project -> Project Options -> Parameters -> Add Library or Object



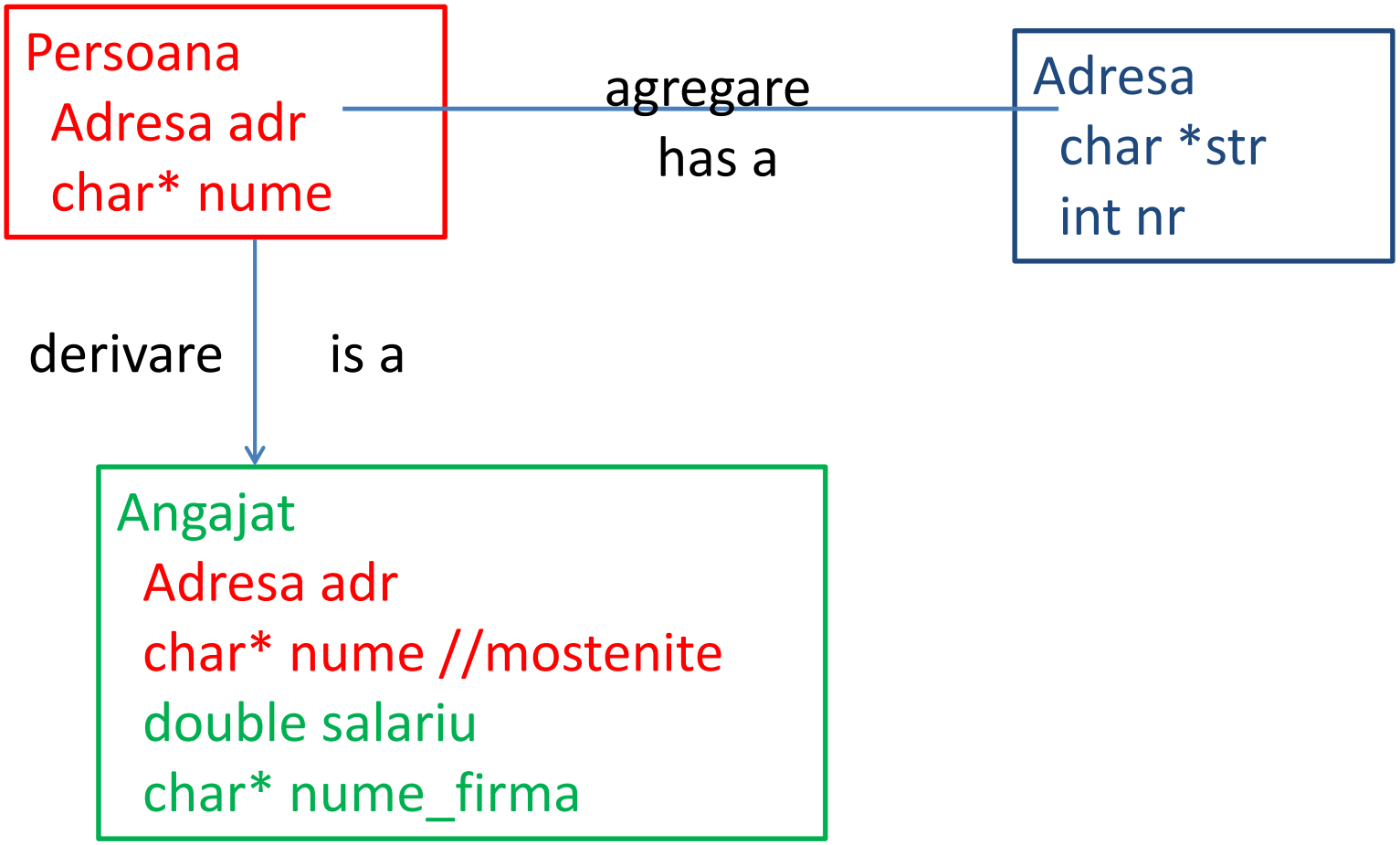
```
#include <iostream>
using namespace std;
#include "Derivata.h"

class Derivata_dupa_cumparare:public Derivata
{  int atr3;
public:
     Derivata_dupa_cumparare(int i, int j, int k):Derivata(i,j),atr3(k) {
     }

     void afisare()
     {
         Derivata::afisare();
         cout<<"atr3="<<atr3<<endl;
     }

     };
int main(int argc, char *argv[])
{
   Baza *b=new Derivata_dupa_cumparare (1,2,3);
   b->afisare(); //1 2 3  //afisare e declarata virtuala in Baza
   return 0;
}
```

Reguli generale:

1. Se recomanda sa se declare ca virtuale, functiile care, in derivari ulterioare NU isi schimba semnificatia (acelasi nume, semnatura, tip returnat), ci doar li se modifica / adapteaza implementarea /comportamentul (in functie de noul tip de date).

2. Daca o clasa are macar o functie virtuala, destructorul trebuie declarat virtual.

Exemplu amplu, C6 reluat.



Vreau sa pot sa creez un vector care sa contina si persoane si angajati (lista neomogena) si sa pot sa fac afisarea tuturor atributelor obiectelor fara sa imi pun probleme legate de tip.

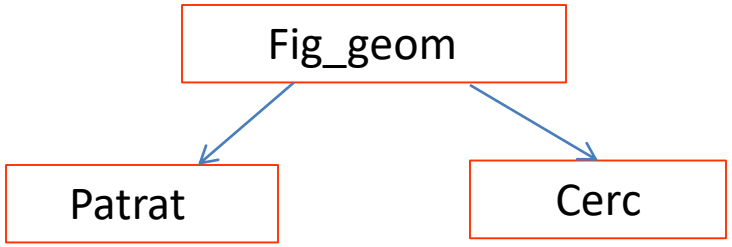
```
class Adresa {
   private: char* str;
            int nr;
   public:
            Adresa();
            Adresa(char*, int);
            Adresa(const Adresa&);
           Adresa& operator=(const Adresa &);
           friend ostream& operator<<(ostream &,const Adresa &);
          ~Adresa();
};

class Persoana {
   protected: Adresa adr; //agregare
                  char *nume;
   public:
  Persoana();
  Persoana(const Adresa &, char* );
  Persoana(char* , int , char* );
  Persoana(const Persoana &)
  Persoana& operator=(const Persoana &);
  virtual  void afisare();
  virtual  ~Persoana();
};
```

```
class Angajat:public Persoana {
   private:
            double salariu;
            char*nume_firma;
   public:
      Angajat();
      Angajat(const Adresa &, char*,double, char*);
      Angajat(char* , int , char* , double, char* );
      Angajat(const Angajat &);
      Angajat& operator=(const Angajat &);
      void afisare();//e virtuala
      ~Angajat(); //e virtual
};
```

```
int main()
{
   int n;
   cin>>n;
   char *num=new char[20]; char *n_firma=new char[20];
   char *str=new char[20];
   int nr;
   double sal;

//    bool *t=new bool[n]; -nu mai e nevoie de un vector in care memorez daca persoana e
//angajata sau nu
 Persoana **evidenta=new Persoana*[n];
//un vector in care vreau sa stochez atat obiecte de tip Persoana* cat si  Angajat*

   for (int i=0;i<n;i++){
      cout<<"Adaugati salariat sau nesalariat?";
      bool tip;cin>>tip;
    //   t[i]=tip; //nu mai am nevoie sa retin tipul intr-un vector auxiliar
      if (tip==true) {
                  cout<<"Nume: "; cin>>num;
                  cout<<"Str. "; cin>>str;
                  cout<<"Nr."; cin>>nr;
                  cout<<"Salariu."; cin>>sal; cout<<“Firma:”; cin>>n_firma;
                  evidenta[i]=new Angajat(str,nr,num,sal,n_firma);
```

```
}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) { //nu mai am nevoie sa testez tipul
    //                    ((Angajat*)(evidenta[i]))->afis();
    //                     cout<<"_____"<<endl;
    //                }else {
                      evidenta[i]->afis();  //comportament polimorfic cu functii virtuale
                                            //legare dinamica
    //                     cout<<"_____"<<endl;
    //                }
    //nu mai este nevoie de conversie explicita, deoarece, in functie de tipul dinamic al
//pointerului  evidenta[i] se apeleaza functia de afisare corespunzatoare obiectului
//catre care se  pointeaza
    delete [] evidenta;
    return 0;}
//TEMA: implementati metode pentru citirea datelor de la tastatura.
```

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

class Fig_geom
{ protected: char *nume;
  public:
     Fig_geom(){  setNume(" ");}

     Fig_geom(const char*n){ setNume(n); }

     Fig_geom(const Fig_geom &n){  *this=n; }

    Fig_geom& operator=(const Fig_geom &n){    setNume(n.nume);
                                               return *this;}
    void setNume(char *n){
              nume=new char[strlen(n)+1];
              strcpy(nume,n); }

    void afisare(){  cout<<nume<<endl; cout<<arie();}

     virtual  double arie(){  return 0;}    //pot sa o fac private ?

     virtual ~Fig_geom(){  delete[]nume;}
    };
```



Creati un vector ce contine figuri geometrice de orice tip si afisati numele/tipul si aria lor

Tema: Implementati testele pentru NULL si eliberarea de spatiu pentru nume.

```
class Patrat:public Fig_geom
{    double lat;
 public:
     Patrat(char*n,double i):Fig_geom(n) {    lat=i;  }

    /*Patrat(const Patrat& i):Fig_geom(i) {     lat=i.lat; }

     Patrat& operator=(const Patrat& i) {
                        (Fig_geom&)(*this)=i;
                         lat=i.lat;
                         return *this;
       }

       ~Patrat(){}*/   //*operator=, constructorul de copiere si destructorul trebuiau implementati
             // daca aveam atribute pointer carora le alocam spatiu; Regula: nu am pt cine elibera
             // Regula: nu am pt cine elibera spatiu in destructor=> nu implementez cele 3 functii
     double arie() { return lat*lat;  } //pot sa o fac private?
    };

class Cerc:public Fig_geom
{    double raza;
 public:
     Cerc(char*n,double i):Fig_geom(n) { raza=i; }
      double arie(){ return 3.14*raza*raza; } //pot sa o fac private?
    };     //se mosteneste in ambele clase functia de afisare care apeleaza functia virtuala arie
```

```
int main()
    {
       Fig_geom ** v=new Fig_geom*[3];
        v[0]=new Fig_geom("punct");
        v[1]=new Patrat("patrat 1",2);
        v[2]=new Cerc("cerc 1",3);

        for (int i=0;i<3;i++)
       {
            v[i]->afisare(); //dar afisare nu e virtuala…
                             //- nu, dar apeleaza o functie virtuala si este mostenita de
                             //     toate clasele derivate! Nu o suprascriu pentru ca face
                             //     mereu acelasi lucru; doar aria se calculeaza altfel
            cout<<endl;
        }
       cout<<endl;

        delete [] v;//cum se apeleaza destructorii?

        return 0;
    }

//ce se intampla daca v continea obiecte de tip Fig_geom si nu pointeri?
```

“Teme”

- **ce se intampla daca Patrat si Cerc nu erau derivate din Fig_geom?**

- **extindeti aceasta ierarhie adaugand functia perimetru si clasele Dreptunghi; Paralelipiped, Sfera si o functie volum**

“Delegating constructors” si atribute cu valori predefinite (C++ 11)

```
#include <iostream>
using namespace std;
class test{         int a;
                    int b=3;
public:
        test(int x=0):a(x){}
        test(int x,int y):test(x),b(y) { )//delegating constructors
         void afis(){cout<<a<<" "<<b<<endl;}
};


int main()
{
test d(2), dd(2,4),ddd;

d.afis();      //2 3
dd.afis();    //2 4
ddd.afis();  //0 3

return 0;
}
```

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

class A{
        int a=3;
        char *f=new char[a];
        int *g=NULL;
public :
/*      A (int x):a(x){  cout<<a<<endl;
                         cout<<(g==NULL)<<endl;   }*/
        void afis(){
                cout<<a;
                strcpy(f,"aa");
                cout<<endl<< (f);
                cout<<endl<<(g==NULL);
        }
};
int main() {
   //A z(1); testati
   A x;
   x.afis();
   return 0;
}
```