PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 8

Polimorfism - continuare

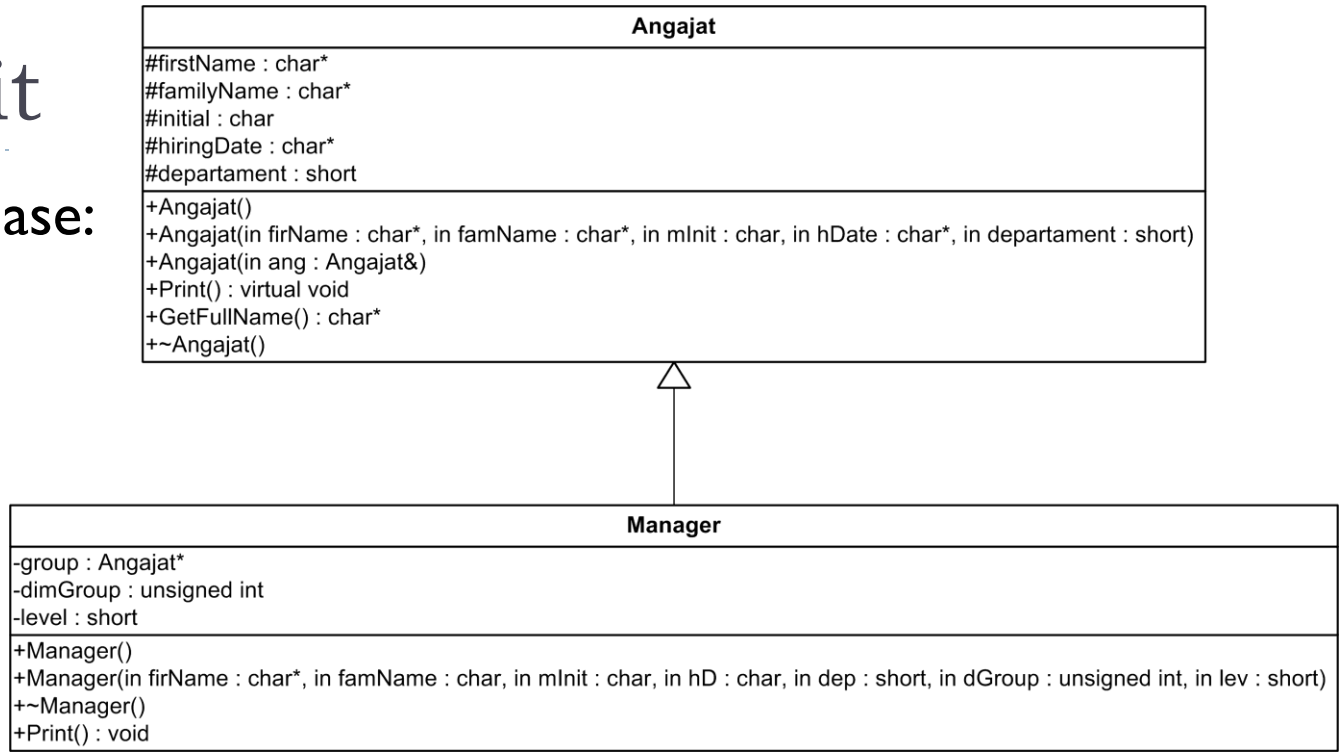
Polimorfism
Type casting

Tabela funcțiilor virtuale

- Conține pointeri către funcțiile virtuale
- Este creată pentru fiecare clasă ce conține funcții membre virtuale sau suprascrie funcții virtuale
- Există numai una pentru o anumită clasă
- Există clase pentru care nu este creată
- Când un obiect este creat, se adaugă un membru ascuns, un pointer către această tabelă virtuală
- Compilatorul generează automat codul în constructori pentru inițializarea pointerului către această tabelă
- Se accesează în momentul când se execută o funcție virtuală

Apel explicit

- Fie ierarhia de clase:



- Apelând o funcție cu ajutorul operatorului apartenență la domeniu asigură că mecanismul virtual nu este folosit:

```
void Manager::Print(void)const
{
    Angajat::Print(); //nu este un apel virtual
    std::cout << "Level = "<<level <<"\n"; //afiseaza informatii
                                            specifice Manager
}
```

Clase abstracte

- Clasa Angajat poate fi utilizată de sine stătător, ca interfață pentru clasele derivate și ca parte a claselor derivate.
- Totuși sunt clase precum Figura care reprezintă concepte abstracte pentru care obiectele nu există.
- O Figura are sens doar dacă o clasă este derivată din ea.

```
class Figura
{
public:
    virtual void rotate(int) { cout << "Figura::rotate\n"; }
    virtual void draw() const { cout<<"Figura::draw\n"; }
};
```

- O alternativă este ca funcțiile din clasa Figura să fie declarate ca fiind funcții virtuale pure

Clase abstracte

```
class Figura
{
public:
        virtual void Rotate(int) = 0; // pure virtual function
        virtual void Draw() const = 0; // pure virtual function
        virtual ~Figura(){}; //virtual
};
```

- O clasă cu cel puțin o funcție virtuală este o clasă abstractă și nu poate instanția nici un obiect.

```
Figura fig;
error C2259: 'Figura' : cannot instantiate abstract class
```

- O clasă abstractă se dorește a fi o interfață pentru obiectele accesate prin intermediul pointerilor și referințelor.
- O clasă abstractă trebuie să aibă un destructor virtual.
- Deoarece o clasă abstractă nu instanțiază obiecte, în mod uzual nu are constructori.

Clase abstracte

- O clasă abstractă poate fi folosită doar ca interfață pentru alte clase:

```
class Punct
{
public:
        int x;
        int y;
        Punct():x(0),y(0){};
};
class Cerc : public Figura
{
private:
        Punct centru;
        int raza;
public:
        Cerc(): raza(0){};
        Cerc(Punct p, int r);
        void rotate(int) {}
        void draw() const {};
};
```

Clase abstracte

- O funcție virtuală pură ce nu a fost definită în clasa derivată rămâne o funcție virtuală pură. Astfel clasa derivată este de asemenea abstractă.

```
class Poligon : public Figura //clasă abstracta
{
public:
    void draw()const {};
};
```

- O clasă abstractă furnizează o interfață fără a expune detaliile de implementare

Constructori şi destructori

- Constructori virtuali ? – ***NU***

```
class Punct
{
public:
    int x;
    int y;
    virtual Punct():x(0),y(0){};
};
…
error C2633: 'Punct' : 'inline' is the only
legal storage class for constructors
```

- În cazul constructorilor, pentru crearea unui obiect este necesară cunoașterea exactă a tipului acestuia
- În plus, tabela virtuală nu a fost inițializată și deci nu se poate crea dacă nu se cunoaște tipul obiectului

Constructori şi destructori

- Destructorii claselor de bază trebuie declarați virtuali pentru a asigura apelarea destructorilor din clasele derivate
- În caz contrar există pericolul de a nu dealoca toată memoria utilizată
- În cazul destructorilor, se apelează toți destructorii dinspre clasa derivată către clasa de bază (în ordinea inversă a apelării constructorilor)

Type casting

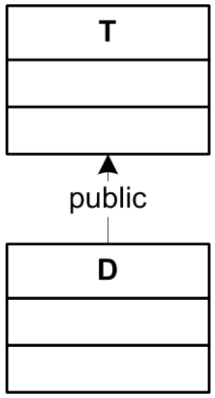
- Conversia unei expresii dintr-un anumit tip în alt tip este cunoscută sub numele de Type casting
- Conversia dintr-o clasă de bază către o clasă derivată este în mod uzual denumită *`downcast`* datorită modului în care este reprezentată grafic ierarhia de clase.
- Conversia dintr-o clasă derivată către o clasă de bază este în mod uzual denumită *`upcast`*.
- Similar, conversia de la o clasă derivată către o altă clasă derivată se numește *`crosscast`*.
- Conversia se realizează cu ajutorul operatorilor:
  - *`dynamic_cast <>`*
  - *`static_cast <>`*
  - *`reinterpret_cast <>`*
  - *`const_cast <>`*

Operatorul *dynamic_cast*

- Operatorul *dynamic_cast* are nevoie de doi operanzi: un tip de date încadrat între două paranteze unghiulare *< T* >* și un pointer încadrat de două paranteze *(p)*

dynamic_cast<T>(p)*

- Dacă *p* este de tipul *T** sau a unui tip *D** derivat din *T* atunci rezultatul este exact ca și cum s-ar atribui *p* unui *T**.
- Se observă că prezența operatorului *dynamic_cast* nu este necesară. Totuși este bine de știut că operatorul *dynamic_cast* nu permite accidental încălcarea regulei de protecție private sau protected a clasei de bază.

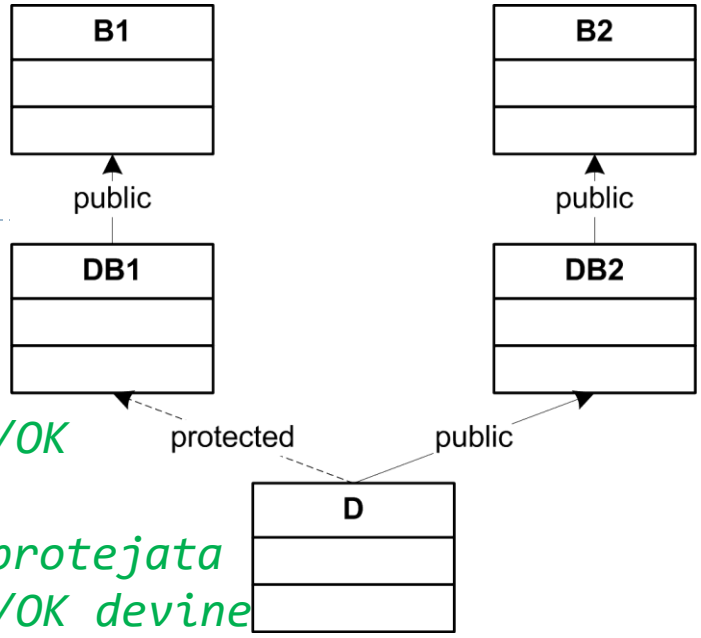


Operatorul *dynamic_cast*

```
void f(D* p)
{
    DB2* pdb2_1 = p;              //OK
    DB2* pdb2_2 = dynamic_cast<DB2*>(p);//OK

    DB1* pdb1_1 = p;  //eroare DB1 clasa protejata
    DB1* pdb1_2 = dynamic_cast<DB2*>(p);//OK devine
nullptr

}
```



- Odată ce *dynamic_cast* folosit ca upcast este exact o simplă atribuire nu implică nici un overhead și este senzitiv contextului lexical.
- Scopul operatorului este de a fi utilizat în situațiile în care corectitudinea conversiei nu poate fi determinată de compilator.

Operatorul *dynamic_cast*

- Astfel operatorul *dynamic_cast<T*>(p)* analizează tipul obiectului pointat de *p*. Dacă acel obiect este de clasa *T* ori are o clasă unică *T*, dynamic_cast returnează un pointer de clasă *T**, altfel *nullptr*.
- Dacă p este *nullptr* atunci *dynamic_cast<T*>(p)* returnează *nullptr*.
- Operatorul *dynamic_cast* necesită un pointer sau o referință către un tip polimorfic pentru downcast ori crosscast.

```
class B1
{
public:
    virtual void f(void){};
}
```

```
class DB1:public B1
{
public:
    void f(void){};
}
```

```
void g(B1* pb1)
{
   DB1* pdb1 = dynamic_cast<DB1*>(pb1); //OK
}
```

Operatorul *dynamic_cast*

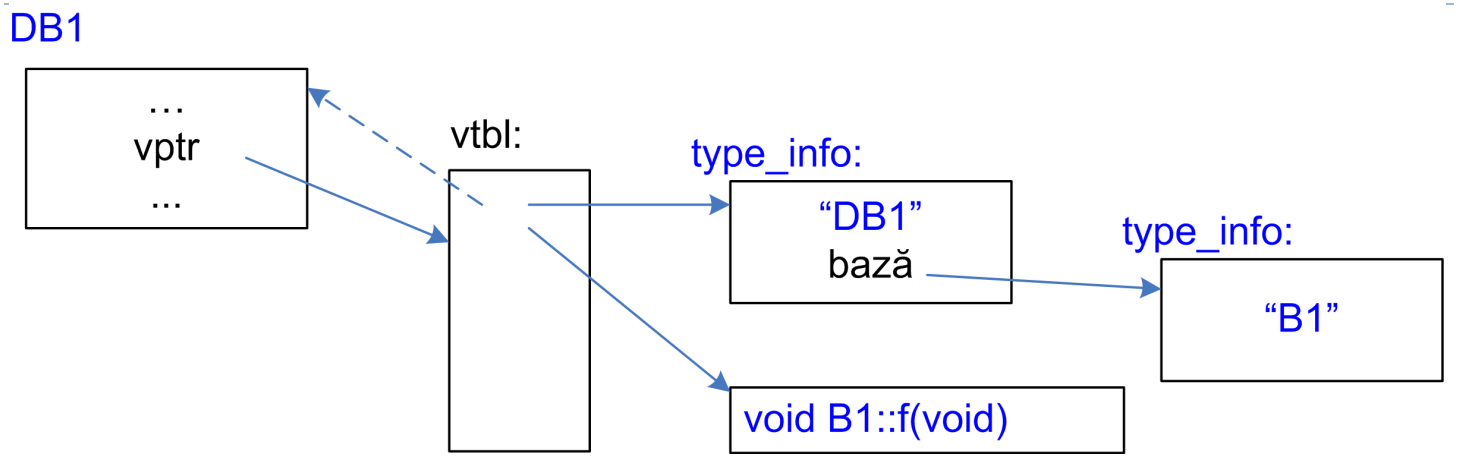
```
class B2
{
public:
    void f(void){};
}
```

```
class DB2:public B1
{
    //…
public:
    void f(void){};
}
```

```
void h(B2* pb2)
{
    DB2* pdb2 = dynamic_cast<DB2*>(pb2); //eroare B2 nu este
                                         //polimorfic
}
```

- Necesitatea ca tipul pointerului să fie polimorfic simplifică implementarea operatorului *dynamic_cast* deoarece există posibilitatea de atașare a „*type information object*” unui obiect plasând un pointer către *type information* în *vtbl*.

Operatorul *dynamic_cast*



- Săgeata punctată reprezintă un offset (informație) de regăsire a obiectului având un pointer către un subobiect polimorfic.
- Restricționarea data de utilizarea operatorului dynamic_cast are sens din punct de vedere logic. Dacă un obiect nu are funcții virtuale nu poate fi manipulat în siguranță fără a se cunoaște exact tipul acestuia. Totodată trebuie avut grija ca un astfel de obiect sa nu fie adus într-un context în care să nu i se cunoască tipul.

Operatorul *dynamic_cast*

- Tipul pointerului țintă a operatorului *dynamic_cast* nu trebuie să fie polimorfic. Acest lucru permite ascunderea unui tip concret într-un tip polimorfic, trasmisia lui prin intermediul unui obiect al sistemului I/O și apoi despachetarea acestuia într-un tip concret.
- Se poate face un *dynamic_cast* către *void** cu scopul de a determina adresa de început a unui obiect polimorfic.

```
void h(B2* pb2, B1* pb1)
{
    void* pdb1 = dynamic_cast<DB1*>(pb1); //OK
    void* pdb2 = dynamic_cast<DB2*>(pb2); //eroare B2 nu este
                                          //polimorfic
```

- Aceste conversii sunt utile atunci când se interacționează cu funcții low level.

dynamic_cast la referință

- Dynamic_cast la o referință este o aserțiune: „obiectul referit este de tipul respectiv”.

```
dynamic_cast<T&>(r)
```

- Rezultatul operatorului *dynamic_cast* este implicit testat de implementarea operatorului. Dacă operandul referit nu este de tip așteptat se aruncă o excepție *bad_cast*

Operatorul *static_cast*

- Utilizarea operatorului dynamic_cast implică existența unui operator polimorfic deoarece nu există informație stocată într-un operator nonpolimorfic.
- Utilizarea operatorului dynamic_cast implică un cost de timp, cu alte cuvinte sunt mii de linii de cod scrise ce trebuie executate până când dynamic_cast este disponibil.
- Totodată dynamic_cast nu poate converti dintr-un void*.
- Pentru acest lucru se utilizează static_cast:

```
DB1* f1(void* p)
{
    B1* pb1 = static_cast<B1*>(p);
    return dynamic_cast<DB1*>(pb1);
}
```

Operatorul *static_cast*

- Realizează conversii între pointeri către clase înrudite (bază-derivată, derivată - bază). Acest lucru asigură faptul că cel puțin clasele sunt înrudite dacă un pointer corespunzător este convertit.

```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

- Nu efectuează nici o verificare la rulare asupra obiectului, deci nu se recomandă a fi folosit decât dacă știi ce faci.
- Necesită verificări suplimentare din partea programatorului pentru a fi sigur ca s-a efectuat cu succes conversia
- Poate fi folosit și la o conversie ce nu implică pointeri dar implică conversii implicite

```
double d=3.14159265;
int i = static_cast<int>(d);
```

Operatorul *reinterpret_cast*

- Convertește un pointer către orice tip de pointer, chiar dacă pointează către clase complet diferite (nerelaționate).
- Nu este verificat nici conținutul pointat nici tipul către care se pointează
- Se utilizează în cod low-level

```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

Operatorul *const_cast*

- Se utilizează în manipularea proprietății *const* a unui obiect

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

void Print (char * str)
{
    cout << str << endl;
}

int main (void)
{
    const char * c = "exemplu";
    Print ( const_cast<char *> (c) );
    return 0;
}
```

Vă mulțumesc !