Tutoriat 5

Ana-Maria Rusu & Ionuț-Daniel Nedelcu
Grupa 242
Facultatea de Matematică și Informatică a Universității din București

Mențiune:

toate programele sunt rulate în Visual Studio Code

compilator g++, comanda terminal: g++ sursa.cpp –o sursa && ./sursa

Ce vom face azi?

- Polimorfism
- Virtualizare
- Multe Exerciții

01

Polimorfism

Polimorfism

→ **Ce inseamna polimorfism?**

Polimorfismul este un concept fundamental al POO, care permite obiectelor (si, mai in general, entitatilor - functii, metode, etc) sa se comporte diferit in functie de context (de a lua mai multe forme). Astfel, codul devine reutilizabil, mai flexibil si extensibil

→ **Polimorfism static** - ce stiam deja la supraincarcarea functiilor si operatorilor pentru diferite clase/tipuri de date, etc. Vom vedea imediat si cum functioneaza pe obiecte - se realizeaza la compilarea programului

Polimorfism

→ **Cum facem polimorfism pe obiecte?**

In C++, atunci cand folosim pointeri si avem o ierarhie de clase, putem retine adresa unui obiect derivat intr-un pointer de tip baza. (dar nu si invers!!)

→ Sintaxa: in main (sau in locul unde dorim sa realizam polimorfismul):

Baza *p = new Derivata; // in pointerul p vom retine adresa unui obiect de tip derivata.

ATENTIE!! Fara downcast (despre care vom vorbi cu alta ocazie), obiectul va apela functiile din Baza, nu Din derivata. De ce? Apelurile se fac din INSTANTA, adica, in acest caz, instanta este “Baza *p”.

Exemplu

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

class Baza
{
    public:
        void afisare() // functia din baza
        { cout << "Baza"; }
};
class Derivata : public Baza
{
    public:
        void afisare() // suprascrierea
functiei din baza
        { cout << "Derivata"; }
};
```

```
int main()
{
    Baza *p = new Derivata; // pointer
de tip Baza care pointeaza catre un
obiect de tip Derivata
    p->afisare(); // se apeleaza functia
afisare() din clasa Baza, desi pointerul
pointeaza catre un obiect de tip
Derivata

    Derivata *q = new Derivata; //
pointer de tip Derivata care pointeaza
catre un obiect de tip Derivata
    q->afisare(); // se apeleaza functia
afisare() din clasa Derivata
}
```

Exemplu

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

class Baza
{
    public:
        int x;
};
class Derivata : public Baza
{
    public:
        int y;
};
```

```
int main()
{
    Baza *p = new Derivata; // pointer de tip Baza care pointeaza catre un obiect de tip Derivata
    p->y = 10; // desi y este public in Derivata, pointerul p este de tip Baza, deci nu poate accesa y

    Derivata *q = new Derivata; // pointer de tip Derivata care pointeaza catre un obiect de tip Derivata
    q->y = 10; // y este public in Derivata, deci se poate accesa
}
```

Polimorfism

→ **Mentiune:** Acest tip de polimorfism este unul foarte rigid, recomandat in putine cazuri (in cele in care ne dorim numai eficienta in executare, nu si ca spatiu, si deloc flexibilitate).

→ Situatiile intalnite mai sus pot fi tratate cu ceea ce vom numi *downcasting*, insa acesta este un subiect pe care il vom discuta cu alta ocazie.

→ Ce trebuie sa retineti, insa: prin acest tip de polimorfism se realizeaza *upcasting implicit* (ca mai sus, vom discuta cu alta ocazie)

02

Virtualizare

Functii Virtuale

→ **Definitie:** functiile (metodele) virtuale, reprezinta functii membre care **sunt definite in clasa de baza si apoi sunt redefinite (overridden) in clasa derivata.**

→ In momentul in care ne referim la un obiect derivat prin intermediul unui pointer catre baza (ex: B*obj =new D) putem folosi functii virtuale pentru a impune obiectului sa apeleze functia din clasa derivata, nu cea din clasa de baza. Astfel, se executa versiunea functiei care se afla cel mai jos in arborele mostenirii.

→ **Sintaxa: virtual** functie (inainte de signatura functiei, se adauga cuvantul cheie virtual)

Functii Virtuale

→ **De ce folosim virtualizarea?**

Pentru a putea extinde codul fara schimbari semnificative si pentru a avea cod mai bine organizat (totodata, se produce **polimorfism la executie**).

→ **Mecanism:**

Daca in clasa derivata se redefineste functia, atunci se va executa versiunea din derivata. In caz contrar, (adica daca nu am redefinit-o in clasa derivata), se executa verisunea functiei din baza.

→Apelarea corecta a functiilor virtuale se realizeaza prin *vptr si VTABLE .Cand declaram o metoda virtuala, fiecare obiect al clasei are un nou membru *vptr pointer catre tabela dinamica VTABLE care contine poineri catre implementarile functiilor virtuale

Precizari

1. O functie virtuala nu poate fi statica
2. O functie virtuala poate fi declarata ca fiind friend cu o alta clasa
3. Pentru a avea polimorfism la executie, functia virtuala trebuie accesata printr-un pointer al bazei ce indica spre derivata
4. Aceste functii incetinesc executia programului
5. O clasa care are in definitia sa o functie virtuala se numeste **clasa polimorfica**

Exemplu

```cpp
//cu virtualizare
//late binding

#include<iostream>
using namespace std;

class B
{
public:
    virtual void f(){cout<<"Baza ";}
};
class D : public B
{
public:
    void f(){cout<<"Derivata ";}
};
int main()
{
    B b; b.f(); // Baza
    D d; d.f(); //Derivata
    B*obj = new D; obj->f(); //Derivata
    return 0;
}
```

```cpp
//fara virtualizare
//early binding

#include<iostream>
using namespace std;

class B
{
public:
    void f(){cout<<"Baza ";}
};
class D : public B
{
public:
    void f(){cout<<"Derivata ";}
};
int main()
{
    B b; b.f();//Baza
    D d; d.f(); //Derivata
    B*obj = new D; obj->f(); //Baza
    return 0;
}
```

Constructori si virtualizare

→ **Constructorii NU pot fi virtuali**

→***Suplimentar:** de ce?

Atunci când constructorul unei clase este executat, nu există niciun vtable în memorie, înseamnă că nu a fost definit încă un pointer virtual. Cu alte cuvinte, atunci cand cream un obiect avem nevoie sa stim tipul sau la **compilare** pentru a aloca corect memoria si pentru a-l initializa corect.

Destructori si virtualizare

→ Este recomandat ca **destructorii sa fie virtuali** .

→ **De ce?** Pentru a asigura distrugerea corecta a obiectelor derivate atunci cand sunt accesate printr-un pointer la clasa de bază.

→Atunci cand lucram cu clase derivate si folosim un pointer la o clasă de baza pentru a gestiona un obiect al unei clase derivate, este esential ca la stergerea obiectului (de obicei, când folosim delete pe un pointer de bază) să fie apelat destructorul corect — adică cel al clasei derivate.

Exemplu

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

class B
{
    int*x;
public:
    B(int x=0) {cout<<"B "; this->x= new
int(x);}
    virtual ~B(){cout<<"~B "; delete x;}
};

class D : public B
{
    int*y;
public:
    D(int y=0):B(0) {cout<<"D "; this->y=new
int(y);}
   ~D(){cout<<"~D ";delete y;}
};
```

```
int main()
{
    B* obj = new D;
    delete obj;
    return 0;
}

// se afiseaza B D ~D ~B
// daca destructorul clasei B nu era
virtual, nu s-ar fi apelat si
destructorul clasei D. Astfel,
nu s-ar fi eliberat memoria pentru
variabila y (care este alocata dinamic).

//Acest fenomen este un caz clasic de
memory leak
```

Vizual, problema arata ceva de genul:

Early & Late Binding

→ **Binding =** procesul de convertire a identificatorilor (varibile, functii etc) in adrese de memorie. Acesta se realizeaza pentru fiecare variabila si functie din program.

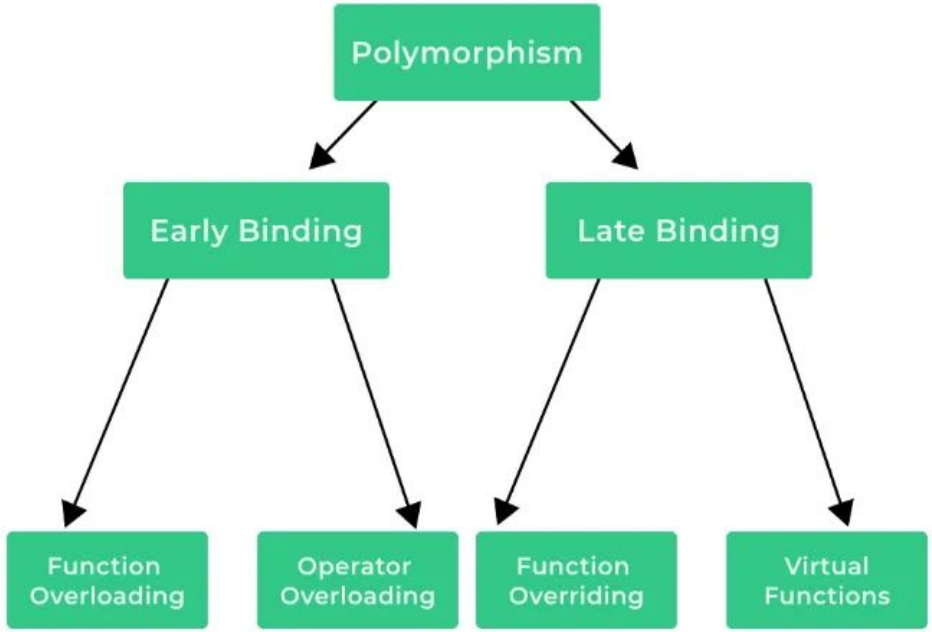
→ ***Early Binding*** : este un tip de *compile time polymorphism* ce asociaza direct (la compilare) o adresa apelului unei functii. In C++ este intampla by default.

→ ***Late Binding*** : este un tip de run time polymorphism in contextul caruia ompilatorul adaugă cod **care identifică tipul de obiect în timpul execuției**, apoi potrivește apelul cu definiția corectă a funcției

-Se realizeaza prin declararea unei functii virtuale

-Altfel spus, *late binding* însemnă că un obiect derivat folosit în locul obiectului de bază îşi va folosi funcţia sa, nu cea din bază

Early binding & Late binding



Comparație între polimorfism static și dinamic

Aspect	Static	Dynamic
Timpul de rezolvare	La compilare	La runtime
Realizare	Prin supradefinire, șabloane, operatori	Prin funcții virtuale
Performanță	Mai rapid (binding static)	Mai lent (binding dinamic, implică vtable)
Flexibilitate	Limitată la timpul de compilare	Extensibil, permite runtime behavior
Utilizare	Cod simplu, operații matematice, generice	Sisteme complexe, scenarii OOP avansate

03

Exercitii