Tutoriat 4

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?

- Mostenire
- Constructorii derivatelor
- Redefinirea Functiilor
- Multe Exerciții

01

Mostenire

Informatii de baza

→ unul dintre scopurile mostenirii - reutilizarea de cod (nu mai cream clase de la 0, ci adaugam componente unor clase deja existente)

→ compunerea este considerata si ea o pseudo-mostenire, in sensul in care contine un obiect dintr-o clasa deja existenta

→ mostenirea poarta si numele de derivare: clasa pe care o mostenim = baza, clasa care mosteneste = derivata

→ o clasa poate mosteni mai multe clase (mostenire multipla)

Informatii de baza

→ prin mostenire se obtine o ierarhie de clase (sau arbore de derivare, cu mentiunea ca o clasa poate avea mai multi parinti): o clasa derivata poate fi baza pentru alta clasa derivata

→ clasele derivate contin toti membrii clasei de baza, la care se adauga noi membri, date, functii membre

→ clasa de baza = parinte, superclasa

→ clasa derivata = copil, subclasa

Sintaxa & modificatori de acces

→ **Sintaxa:** class NumeDerivata: ModificatorAcces1 ClasaDeBaza1, ModificatorAcces2 ClasaDeBaza2 ... {...}

→ **Modificatorii de acces al mostenirii** (default: private)**:**

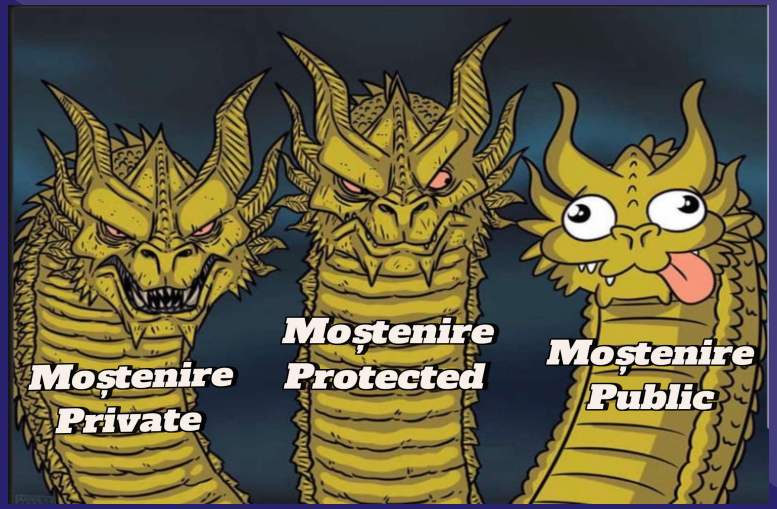
→ **Public** => toti membrii din clasa de baza isi pastreaza tipul de acces si in derivata *

→ **Protected** => membrii “public” din clasa de baza devin “protected” in clasa derivata *

→ **Private** => toti membrii din clasa de baza devin “private” in clasa derivata *

* regulile se aplica, in mod evident, pentru obiectele de tipul clasei derivata. **Obiectele de tipul bazei raman neschimbate ca principii.**

Modificatorii de acces la mostenire



Exemplu – mostenire public

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

class Baza
{
    private:
        int p=100;
    public:
        int x=5;
};
class Derivata1 : public Baza
{
    public:
        void afisare()
        { cout << x<<" "; }
};
class Derivata2 : public Derivata1
{
    public:
        int y;
        void afisare2()
        { cout << x << " " << y; }
};
```

```
int main()
{
    Derivata1 obj1;
    obj1.x = 10; // x este
public in Baza si mostenit
public, deci se poate accesa
    obj1.afisare(); // functia
afisare() este publica in
Derivata1, deci se poate apela

    Derivata2 obj;
    obj.y = 15; // y este public
in Derivata2
    obj.afisare2(); // x este
public in Baza si mostenit apoi
public pana in Derivata 2, deci
se poate accesa
    cout << obj.p; // p este
private in Baza, deci nu se
poate accesa
    return 0;
}
```

Exemplu - mostenire protected

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

class Baza
{
    public:
        int x=5;
};
class Derivata1 : protected Baza
{
    public:
        void afisare()
        { cout << x<<" "; }
};
class Derivata2 : public Derivata1
{
    public:
        int y;
        void afisare2()
        { cout << x << " " << y; }
};
```

```
int main()
{
    Derivata1 obj1;
    obj1.x = 10; // eroare: x
este protected in Derivata1
    obj1.afisare(); // functia
afisare() este publica in
Derivata1, deci se poate apela

    Derivata2 obj;
    obj.y = 15; // y este public
in Derivata2
    obj.afisare2(); // x este
protected in Derivata1, dar se
poate accesa din Derivata2
deoarece Derivata2 mosteneste
Derivata1
    return 0;
}
```

Exemplu – mostenire private

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

class Baza
{
    public:
        int x=5;
};
class Derivata1 : private Baza
{
    public:
        void afisare()
        { cout << x; }
};
class Derivata2 : public Derivata1
{
    public:
        int y;
};
```

```
int main()
{
    Derivata1 obj1;
    obj1.x = 10; // eroare: x
este privat in Derivata1
    obj1.afisare(); // functia
afisare() este publica in
Derivata1, deci se poate apela

    Derivata2 obj;
    obj.y = 15; // y este public
in Derivata2
    obj.x = 10; // eroare: x
este privat din mostenirea Baza
-> Derivata1
    return 0;
}
```

02

Constructorii clasei derivate

Ordinea de apel

→ constructorii sunt apelati in ordinea creării obiectelor, a mostenirii

→ pentru fiecare nivel se apeleaza - constructorul de la mostenire
- constructorii din obiectele membre ale clasei respective
- la final se merge pe următorul nivel în ordinea moștenirii

→ destructorii sunt executați în ordinea inversă a constructorilor

Exemplu

```
#include<iostream>
using namespace std;
class A{
public:
     A(){cout<<"A ";}
     ~A(){cout<<"~A ";}
};

class C{
public:
     C(){cout<<"C ";}
     ~C(){cout<<"~C ";}
};
class B{
     C ob;
public:
     B(){cout<<"B ";}
     ~B(){cout<<"~B ";}
};
```

```
class D: public B{
     A ob;
public:
     D(){cout<<"D ";}
     ~D(){cout<<"~D ";}
};

int main()
{
     D s;
}

// C B A D  ~D ~A ~B ~C
```

Mecanism de apel: Constructorul default/parametrizat

→ pentru a crea un obiect derivat se creeaza initial un obiect al clasei de baza prin apelul constructorului ei (adica cel al clasei de baza)

→apoi se adauga elemente specifice derivatei (adica se apeleaza constructorul clasei derivate)

Exemplu

```
#include <iostream>
using namespace std;
class B
{
    int x;
public:
    B(int i = 2): x(i){cout<<"B ";}
};

class D : public B
{
    int y;
public:
    D(int i=2,int j = 7): B(j) { y=j;
cout<<"D";}
};
```

```
int main()
{
    D d;
    return 0;
}
```

→ se afiseaza B D
→ apelam constructorul clasei B, apoi constructorul clasei D

Mecanism de apel : Constructorul de copiere

B=baza && D=derivata

1) daca **B si D nu** au definit constructorul de copiere ⇒ se apeleaza cel creat automat de catre compilator (presupunem ca nu avem date alocate dinamic. In caz contrar, trebuie sa redefinim tot).

2) daca **B il are si D nu** ⇒ compilatorul creeaza un constructor pentru D care apeleaza constructorul de copiere din clasa B

3) daca **B si D il au** ⇒ lui D îi revine în totalitate sarcina transferării valorilor corespunzatoare membrilor ce apartin clasei de bază

Exemplu

```
#include <iostream>
using namespace std;
class B
{
protected:
    int x;
public:
    B(int x=13) { this->x = x;}
    B(const B&b) {this->x = b.x; cout<<"B
";}
};


class D : public B
{
public:
    D(int x=42):B(x){}
    D(const D&) { cout<<"D ";}
};
```

```
int main()
{
    D d1(15);
    D d2(d1);
    return 0;
}
```

→ se afiseaza D

→ cazul 3: "daca **B si D il au** ⇒ lui D îi revine în totalitate sarcina transferării valorilor corespunzatoare membrilor ce apartin clasei de bază"

03

Redefinirea functiilor in derivata

Redefinirea functiilor membre in derivata

→ Este permisă supradefinirea funcţiilor membre ale clasei de bază cu funcţii membre ale clasei derivate.

→ exista 2 modalitati:

1) cu acelasi antet ca in clasa derivata

2) cu schimbarea listei de argumente / a tipului

→ IMPORTANT : **la redefinirea unei funcţii din clasa de baza, toate celelalte versiuni ale ei sunt automat ascunse**

→Observatii:

-Schimbarea interfeței clasei de bază prin modificarea tipului returnat sau a signaturii unei funcții, înseamnă, de fapt, utilizarea clasei în alt mod.

-constructorii si destructorii nu pot fi mosteniti → se redefinesc altii pentru derivate (la fel si pentru operatorul =)

Exemple

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

class Baza {
public:
    void afis( ){cout<<"Baza\n";}
};

class Derivata : public Baza {
public:
    void afis (int x){cout<<"si
Derivata\n";}
};

int main( )
{
    Derivata d;
    //d.afis(); // nu exista
Derivata::afis() pentru clasa Derivata =>
eroare
    d.afis(3);
    return 0;
}
```

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

class Baza {
public:
    void afis( ){cout<<"Baza\n";}
};

class Derivata : public Baza {
public:
    void afis (
){cout<<"Derivata\n";}
};

int main( )
{
    Derivata d;
    d.afis();
   // se apeleaza versiunea din
Derivata ⇒ se           afiseaza
Derivata
    return 0;
}
```

04

Exercitii