Tutoriat 2

Cuprins

Moștenire

Al doilea principiu fundamental al OOP este **moștenirea**.

Acesta constă în *extinderea* unei clase, prin crearea altor clase cu proprietăți comune.

Vom introduce 2 noțiuni importante cu care vom lucra în continuare: ***clasă de bază (base class)*** și ***clasă derivată (derived class)***.

Clasă de bază vs Clasă derivată

Pe scurt: ***Clasa de bază* este cea DIN care se moștenește, cea derivată este cea CARE moștenește.**

```
class Animal {
    // date si metode specifice
};

class Dog : Animal {
    // date si metode specifice
};
```

În exemplul de mai sus, facem următoarele corespondențe:

- **Animal** = **clasa de bază**;
- **Dog** = **clasa derivată**.

Observație

Moștenirea se realizează prin simbolul *:* pus între numele clasei derivate și acolade. Apoi, se specifică din ce clasă se moștenește.

Mai exact, moștenirea presupune *transmiterea datelor și metodelor* din *clasa de bază* în *clasa derivată* după anumite criterii despre care vom vorbi la tipurile de moștenire.

Tot ceea ce este PRIVATE în clasa de BAZĂ devine INACCESIBIL în clasa DERIVATĂ.

Moștenire vs Compunere

Înainte de a putea avansa, trebuie stabilită cu certitudine distincția dintre *moștenire* și *compunere*.

Compunerea este procedeul de declarare a unui obiect de tipul unei clase, ca *dată* a altei clase.

Exemplu:

```
class Student {
    // date si metode specifice
};

class Facultate {
    Student s[100]; // compunere
};
```

IMPORTANT

O clasă *D MOȘTENEȘTE* o clasă B, dacă se poate spune ”D ESTE (IS A) un obiect de tip B”.

O clasă *D APARȚINE* (compunere) de o clasă B, dacă se poate spune ”B ARE (HAS A) un obiect de tip D”.

Acest detaliu este foarte important în proiectarea unor clase. De multe ori, suntem tentați să folosim moștenirea când nu e nevoie, sau invers.

De aceea, chiar dacă pare ciudat, este bine să ne punem aceste 2 întrebări când construim relații între clase.

Exemple de moșteniri:

- **FormaGeometrică** ===> **Pătrat / Romb / ...** (Pătratul *ESTE* o formă geometrică).
- **Animal** ===> **Câine / Pisică / ...** (Câinele *ESTE* un animal).
- **Persoană** ===> **Student / Profesor / ...** (Studentul *ESTE* o persoană).
- ...

Exemple de compuneri:

- **Facultate** ===> **Student / Profesor / ...** (Facultatea *ARE* studenți).
- **Firma** ===> **Angajat / Director / ...** (Firma *ARE* angajați).
- ...

Tipuri de moșteniri

1. Moștenire privată

Această moștenire presupune că **totul** din clasa de **BAZĂ** devine **private** în clasa **DERIVATĂ**.

Singurele excepții sunt datele și metodele *private* care devin inaccesibile.

Acest tip de moștenire este cel implicit când nu specificăm niciun modificator după simbolul *:*

Exemplu:

```
class Animal {
    // date si metode specifice
};

class Dog : Animal {
    // date si metode specifice
};
```

```
class Animal {
    // date si metode specifice
};

class Dog : private Animal {
    // date si metode specifice
};
```

2. Moștenire protected

Tot ce nu e **private** în clasa de **bază** devine **protected** în clasa **derivată**.

Exemplu:

```
class Animal {
    // date si metode specifice
};

class Dog : protected Animal {
    // date si metode specifice
};
```

3. Moștenire publică

Aceasta este cea mai des folosită moștenire. Totul rămâne la fel ca în clasa de bază.

Datele și metodele private tot inaccesibile rămân.

```
class Animal {
    // date si metode specifice
};

class Dog : public Animal {
    // date si metode specifice
};
```

Relațiile dintre tipurile de moștenire

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Constructori în moștenire

Trebuie să observăm cum sunt apelați constructorii în clasa derivată. Acest lucru poate provoca uneori confuzie.

```
class Animal {
    public:
        Animal() {
            cout << "Animal ";
        }
};

class Dog : public Animal {
    public:
        Dog() {
            cout << "Dog ";
        }
};

int main() {
    Dog d; // Animal Dog
    return 0;
}
```

Prima dată, în constructorul din clasa DERIVATĂ este apelat constructorul din clasa de BAZĂ.

DESTRUCTORII se apelează în ordinea INVERSĂ a constructorilor.

Apelarea unui anumit constructor din clasa de bază

În anumite situații, putem avea mai mulți constructori în clasa de bază.

Putem apela unul anume folosindu-ne de ***LISTA DE INIȚIALIZARE***.

```
class Animal {
    public:
        Animal() {
            cout << "Animal1 ";
        }

        Animal(int x) {
            cout << "Animal" << x << " ";
        }
};

class Dog : public Animal {
    public:
        Dog() : Animal(4) {
            cout << "Dog ";
        }
};

int main() {
    Dog d; // Animal4 Dog

    return 0;
}
```

Aici putem observa cel mai clar că lista de inițializare este apelată *înaintea* corpului constructorului.

În mod **implicit**, ar fi fost apelat constructorul fără parametri, dar noi l-am apelat **explicit** pe cel cu 1 parametru.

Este *OBLIGATORIU* ca în clasa de bază să existe constructorul *FĂRĂ PARAMETRI* (*dacă nu apelăm explicit alt constructor*).

Moștenire multiplă

Dacă întâmpinăm o situație în care trebuie să moștenim mai multe lucruri din mai multe clase diferite, C++ ne oferă posibilitatea *moștenirii multiple*.

```
class Animal {
    public:
        Animal() {
            cout << "Animal ";
        }
```

```
};

class Reptile {
    public:
        Reptile() {
            cout << "Reptile ";
        }
};

class Snake : public Animal, public Reptile {
    public:
        Snake() {
            cout << "Snake ";
        }
};

int main() {
    Snake s; // Animal Reptile Snake
}
```

La moștenire multiplă, constructorii sunt apelați *ÎN ORDINEA ÎN CARE SUNT SPECIFICAȚI LA MOȘTENIRE.*

Adică, noi am moștenit clasele *Animal* și *Reptile* în această ordine. Atunci, exact în această ordine vor fi apelați constructorii celor 2 clase.

Observație

Chiar dacă specificăm o altă ordine în lista de inițializare, constructorii își păstrează ordinea de apelare.

Exemplu:

```
class Animal {
    public:
        Animal() {
            cout << "Animal ";
        }
};

class Reptile {
    public:
        Reptile() {
            cout << "Reptile ";
        }
};

class Snake : public Animal, public Reptile {
    public:
        Snake() : Reptile(), Animal() {
            cout << "Snake ";
        }
};
```

```
int main() {
    Snake s; // Animal Reptile Snake
}
```

Probleme frecvente cu moștenirea

- ***Lipsa de acces în clasa derivată***

```
class Animal {
    int varsta;
};

class Dog : public Animal {
    public:
        int getVarsta() {
            return varsta; // eroare
        }
};
```

Putem observa că în clasa *Animal*, câmpul varsta este de tip *private*. Astfel, orice fel de moștenire am folosi, acesta devine *inaccesibil* în clasa *derivată*.

Rezolvare: modificăm câmpul *varsta*, astfel încât să devină *protected*.

```
class Animal {
    protected:
        int varsta;
};

class Dog : public Animal {
    public:
        int getVarsta() {
            return varsta; // eroare
        }
};
```

- ***Tipul de moștenire***

```
class Animal {
    public:
        Animal() {
            cout << "Animal ";
        }
};

class Dog : Animal {
```

```
    public:
        Dog() {
            cout << "Dog ";
        }
};

class Puppy : public Dog {
    public:
        Puppy() {
            cout << "Puppy ";
        }
};

int main() {
    Puppy p; // eroare

    return 0;
}
```

Această eroare este cauzată de tipul moștenirii pentru clasa Dog: `class Dog : Animal`. Această moștenire este de tip *private*, așa că totul din Animal devine private în clasa Dog.

Până acum însă nu este nicio problemă. Se poate apela constructorul clasei Animal, chiar dacă este *private*.

Problema apare în `int main()`. Din cauza moștenirii pentru clasa Puppy `class Puppy : public Dog`, tot ce era **private** în clasa Dog, devine **inaccesibil** în clasa Puppy.

Problema este că tot private era și constructorul clasei Animal. Astfel, în clasa Puppy, nu mai putem *instanția clasa Animal*.

Trebuie acordată mare atenție acestor tipuri de moșteniri, mai ales când avem de-a face cu *moșteniri înlănțuite*.

O posibilă rezolvare este transformarea moștenirii pentru clasa Dog într-o moștenire public.

```
class Animal {
    public:
        Animal() {
            cout << "Animal ";
        }
};

class Dog : public Animal {
    public:
        Dog() {
            cout << "Dog ";
        }
};

class Puppy : public Dog {
    public:
```

```
        Puppy() {
            cout << "Puppy ";
        }
};

int main() {
    Puppy p; // Animal Dog Puppy

    return 0;
}
```

- **Ordinea apelurilor constructorilor și destructorilor**

În acest exemplu, vom discuta despre un program care rulează dar cauzează dificultăți în stabilirea a ce se *afișează* pe ecran la sfârșitul execuției.

Exemplul de mai jos este unul mai scurt și simplificat. În general, problemele conțin șiruri de câte 4 clase și se mai pot complica și cu diferite tipuri de constructori.

Trebuie să reținem ordinea. **Constructorii** se apelează de la cel mai de la bază, iar **destructorii** se apelează în ordine inversă.

```
class Animal {
    public:
        Animal() {
            cout << "Animal ";
        }

        ~Animal() {
            cout << "DAnimal ";
        }
};

class Dog : public Animal {
    public:
        Dog() {
            cout << "Dog ";
        }

        ~Dog() {
            cout << "DDog ";
        }
};

class Puppy : public Dog {
    public:
        Puppy() {
            cout << "Puppy ";
        }

        ~Puppy() {
```

```
            cout << "DPuppy ";
        }
};

int main() {
    Puppy p; // Animal Dog Puppy

    return 0;
} // DPuppy DDog DAnimal
```