Tutoriat 3

Moştenire

În C++, există un concept special specific claselor, *moştenirea (inheritance).* Acesta constă în *extinderea* unei clase prin crearea altor clase cu proprietăţi comune.

Acesta este unul din cele trei *principii fundamentale ale OOP.*

Principiul introduce 2 noţiuni importante: clasa *bază (base class)* şi clasa *derivată(derived class).*

Clasa de bază vs clasa derivată

Mai pe scurt, ***clasa de bază este cea DIN care se moşteneşte, cea derivată este cea CARE moşteneşte.***

```
1 class Animal {
2    /* Date si metode specifice*/
3 };
4
5 class Dog : Animal {
6    /* Date si metode */
7 };
```

În acest exemplu, clasa *Animal* este clasa *de bază (din care se moşteneşte)* şi clasa *Dog* este cea *derivată (care moşteneşte).*

Obs. Moştenirea se realizează prin simbolul ***:,*** pus între numele clasei derivate şi acolade. Apoi, se specifică din ce clasă se moşteneşte.

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

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

Moştenire vs Compunere

Înainte de a putea avansa, trebuie stabilită cu certitudine distincţia dintre *moştenire* şi *compunere.*

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

```
1 class Student {
2    /* Date si metode specifice*/
3 };
4
5 class Facultate {
6    Student s[100];
7 };
```

Putem observa că în clasa *Facultate* avem un tablou de obiecte de tip *Student.* Acest procedeu se numeşte *compunere.*

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

O clasă D APARŢINE (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 tentaţi să folosim moştenirea când nu e nevoie, sau invers. De aceea, chiar dacă pare ciudat, este bine să ne punem aceste 2 întrebări când construim relaţii între clase.

Exemple de moştenire: ***FormaGeometrica - Patrat/Romb***/… (Patratul *ESTE* o FormaGeometrică), ***Animal - Caine/Pisica/***… (Cainele *ESTE* un Animal), ***Persoana - Student/Profesor/… (***Studentul *ESTE* o persoana)...

Exemple de compunere: ***Facultate - Student/Profesor/…*** (O Facultate *ARE* Studenţi / Profesori…), ***Firma - Angajat / Director /… (***O Firma *ARE* Angajaţi, Directori…) …

Tipuri de moştenire

- ***Moştenire privată***

Această moştenire presupune că totul din clasa *BAZĂ* devine *private* în clasa *DERIVATĂ.* Singurele excepţii sunt datele şi metodele *private* care devin inaccesibile

Acest tip de moştenire este cel implicit când nu specificăm niciun modificator după simbolul **:** . *Exemplu:*

```
class Animal {
   /* Date si metode */
};

class Dog : Animal {
  /* Date si metode */
}
```

```
class Animal {
   /* Date si metode */
};

class Dog : private Animal {
  /* Date si metode */
}
```

Cele 2 metode de mai sus sunt *echivalente.*

- ***Moştenire protected***

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

```
class Animal {
   /* Date si metode */
};

class Dog : protected Animal {
  /* Date si metode */
}
```

- ***Moştenire public***

Aceasta este cea mai des folosită moştenire. Totul rămâne la fel ca în clasa de *bază.* Datele şi metodele *private* tot inaccesibile rămân.

```
class Animal {
   /* Date si metode */
};

class Dog : public Animal {
  /* Date si metode */
}
```

Relaţiile între tipurile de moştenire

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 moştenire

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

```
1 class Animal {
2   public:
3      Animal() {
4         cout << "Animal ";
5      }
6 };
7
8 class Dog : public Animal {
9    public:
10       Dog() {
11          cout << "Dog ";
12       }
13 }
14
15 int main() {
16    Dog d; // "Animal Dog"
17 }
```

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 situaţii, putem avea mai mulţi constructori în clasa de *bază*. Putem apela unul anume folosindu-ne de *LISTA DE INIŢIALIZARE.*

```
1 class Animal {
2   public:
3      Animal() {
4         cout << "Animal1 ";
5      }
6      Animal(int x) {
7         cout << "Animal" << x << " ";
8      }
9 };
10
11 class Dog : public Animal {
12    public:
13       Dog() : Animal(4) {
14          cout << "Dog ";
15       }
16 }
17
18 int main() {
19    Dog d; // "Animal4 Dog"
20 }
```

Aici putem observa cel mai clar că lista de iniţializare 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).

Moştenire multiplă

Dacă întâmpinăm o situaţie în care trebuie să moştenim mai multe lucruri din mai multe clase diferite, C++ ne oferă posibilitatea *moştenirii multiple.*

```cpp
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 moştenire multiplă, constructorii sunt apelaţi ÎN ORDINEA ÎN CARE SUNT SPECIFICAŢI LA MOŞTENIRE.

Adică, noi am moştenit clasele *Animal* şi *Reptile* în această ordine. Atunci, exact în această ordine vor fi apelaţi constructorii celor 2 clase.

Obs. Chiar dacă specificăm o altă ordine în *lista de iniţializare,* constructorii îşi 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 moştenire

- ***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 moştenire am folosi, acesta devine inaccesibil în clasa *derivată*.

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

```
1 class Animal {
2   protected:
3      int varsta;
4 };
5
6 class Dog : public Animal {
7    public:
8      int getVarsta() {
9         return varsta;
10     }
11 }
```

- ***Tipul de moştenire***

```
1 class Animal {
2   public:
3      Animal() {
4         cout << "Animal ";
5      }
6 };
7
8 class Dog : Animal {
9    public:
10     Dog() {
11       cout << "Dog ";
12     }
13 };
14
15 class Puppy : public Dog {
16   public:
17     Puppy() {
18        cout << "Puppy ";
19     }
20 };
21
22 int main() {
23   Puppy p; // eroare
24 }
```

Această eroare este cauzată de tipul moştenirii de la ***linia 8.*** Acea moştenire este de tip *private*, aşa 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 la ***linia 23.*** Din cauza moştenirii de la ***linia 15,*** tot ce era private în clasa *Dog,* devine inaccesibil în clasa *Puppy.* Problema este că tot private era şi constructorul clasei *Animal.* Astfel, în clasa *Puppy,* nu mai putem instanţia clasa *Animal.*

Trebuie acordată mare atenţie acestor tipuri de moşteniri, mai ales când avem de-a face cu moşteniri înlănţuite. O posibilă rezolvare este transformarea moştenirii de la ***linia 8*** într-o moştenire *public.*

```
1 class Animal {
2   public:
3       Animal() {
4           cout << "Animal ";
5       }
6 };
7
8 class Dog : public Animal {
9    public:
10      Dog() {
11        cout << "Dog ";
12      }
13 };
14
15 class Puppy : public Dog {
16    public:
17      Puppy() {
18          cout << "Puppy ";
19      }
20 };
21
22 int main() {
23    Puppy p; // "Animal Dog Puppy "
24 }
```

- ***Ordinea apelurilor constructorilor şi destructorilor***

În acest exemplu, vom discuta despre un program care rulează dar cauzează dificultăţi în stabilirea a ce se afişează pe ecran la sfârşitul execuţiei. Exemplul de mai jos este unul mai scurt şi simplificat. În general, problemele conţin şiruri de câte 4 clase şi se mai pot complica şi cu diferite tipuri de constructori.

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

```cpp
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 "
} // "DPuppy DDog DAnimal"
```