Facultatea de Matematică și Informatică,
Universitatea din București

Tutoriat 6

Programare Orientată pe Obiecte - Informatică, Anul I

Tudor-Gabriel Miu
Radu Tudor Vrînceanu
04-19-2024

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Cuprins

Moștenirea multiplă.....2
În acest exemplu, clasa Derivata moștenește atât metodele clasei A, cât și metodele clasei B. Astfel, obiectul clasei Derivata poate accesa și utiliza toate aceste metode.....3
Coliziuni de nume (ambiguități).....3
Moștenirea în diamant.....5
Apelarea constructorilor în exemplul anterior.....6
Moștenirea virtuală.....8
Exerciții:.....12
Ex. 1:.....13
Ex. 2:.....14
Ex. 3:.....15
Ex. 4:.....16

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Moșteniri (part 2)

Moștenirea multiplă

Moștenirea multiplă este un concept din programarea orientată pe obiecte în limbajul C++ care permite unei clase să moștenească membri și funcționalități de la mai multe clase de bază. Cu alte cuvinte, o clasă derivată poate avea mai multe clase de bază și poate moșteni caracteristicile acestora.

Pentru a utiliza moștenirea multiplă în C++, trebuie să declarăm clasa derivată folosind o listă de clase de bază separate prin virgule în definiția clasei. De exemplu:

```cpp
#include <iostream>

// Definim clasa de bază A
class A {
public:
    void functieA() {
        std::cout << "Functie din clasa A\n";
    }
};

// Definim clasa de bază B
class B {
public:
    void functieB() {
        std::cout << "Functie din clasa B\n";
    }
};

// Definim clasa derivată care moștenește atât clasa A, cât și clasa B
class Derivata : public A, public B {
public:
    void functieDerivata() {
        std::cout << "Functie din clasa Derivata\n";
    }
};

int main() {
    // Creăm un obiect al clasei Derivata
    Derivata d;

    // Apelăm metodele din clasele de bază și din clasa derivată
    d.functieA(); // Apelăm o metodă din clasa A
    d.functieB(); // Apelăm o metodă din clasa B
    d.functieDerivata(); // Apelăm o metodă din clasa Derivata

    return 0;
}
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

În acest exemplu, clasa `Derivata` moștenește atât metodele clasei `A`, cât și metodele clasei `B`. Astfel, obiectul clasei `Derivata` poate accesa și utiliza toate aceste metode.

Moștenirea multiplă nu e ceva ce recomandăm. De ce?

Să ne amintim ce vorbeam despre moștenire. Declarăm o clasă derivată când avem nevoie de un tip nou care poate să fie descris ca „un fel de (…)”. Ex: Un obiect din clasa `Cerc` este un fel de obiect din clasa `Formă`, așa că derivăm `Cerc` din `Formă`.

Dacă ne dorim să moștenim din două clase ceea ce spunem, de fapt, e cp noul nostru obiect este și un fel de ceva și un fel de altceva - o struțo-cămilă. Putem găsi cazuri în care se justifică această structură:

Să zicem că avem un magazin unde vindem o multitudine de categorii de obiecte. Toate obiectele din inventarul nostru vor moșteni clasa de bază `Produs`.

În paralel, oferim și servicii: livrare, asamblare, etc. Toate serviciile noastre vor moșteni clasa `Serviciu`.

Un pachet care include obligatoriu un produs și un serviciu ar putea să moștenească ambele clase, nefiind nici produs, nici serviciu, ci ambele.

Chiar și așa, exemplul este forțat. Am putea sparge pachetul în elementele componente. Poate ar trebui să anticipăm asta și să avem o a treia clasă de bază, pachete, care să conțină două liste de pointeri: una pentru produse și una pentru servicii.

Moștenirea multiplă face, în general, codul mai greu de înțeles și de menținut și există în limbaj pentru completitudine.

Coliziuni de nume (ambiguități)

Coliziunile de nume apar în moștenirea multiplă atunci când mai multe clase de bază au metode sau membri cu același nume. Această situație poate duce la ambiguitate în cadrul clasei derivate, deoarece compilatorul nu știe care dintre metodele cu același nume să fie folosite în cadrul acesteia.

Pentru a rezolva coliziunile de nume în moștenirea multiplă, putem folosi rezoluția explicită a numelor sau putem folosi calificatori pentru a specifica din care clasă de bază dorim să accesăm metoda sau membrul.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Iată un exemplu care ilustrează cum putem rezolva coliziunile de nume în cazul moștenirii multiple:

```cpp
#include <iostream>

// Definim clasa de bază A
class A {
public:
    void afisare() {
        std::cout << "Afisare din clasa A\n";
    }
};

// Definim clasa de bază B
class B {
public:
    void afisare() {
        std::cout << "Afisare din clasa B\n";
    }
};

// Definim clasa derivată care moștenește atât clasa A, cât și clasa B
class Derivata : public A, public B {
public:
    // Rescriem metoda afisare pentru a evita coliziunea de nume
    void afisare() {
        // Specificăm din care clasă de bază dorim să apelăm metoda
        A::afisare();
        B::afisare();
        std::cout << "Afisare din clasa Derivata\n";
    }
};

int main() {
    // Creăm un obiect al clasei Derivata
    Derivata d;

    // Apelăm metoda afisare a clasei Derivata
    d.afisare();

    return 0;
}
```

În acest exemplu, am rescris metoda afisare în clasa `Derivata` și am specificat din care clase de bază dorim să apelăm metoda folosind rezoluția explicită a numelor `A::afisare()` și `B::afisare()`. Acest lucru ne permite să evităm coliziunea de nume și să obținem rezultatul dorit fără ambiguitate.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

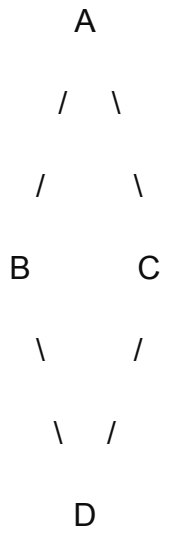
Moștenirea în diamant

Moștenirea în formă de diamant este o situație specifică din programarea orientată pe obiecte în limbajul C++, care apare atunci când o clasă derivată moștenește din două sau mai multe clase care împărtășesc o clasă comună de bază. Această configurație creează o ierarhie de moștenire în formă de diamant, în care clasa derivată are mai multe căi pentru a accesa membrii și funcționalitățile clasei de bază comune.

Să luăm un exemplu pentru a înțelege mai bine acest concept:

```cpp
#include <iostream>

// Clasa de bază
class A {
public:
    void afisare() {
        std::cout << "Afisare din clasa A\n";
    }
};

// Prima clasă derivată care moștenește clasa A
class B : public A {
public:
    void afisare() {
        std::cout << "Afisare din clasa B\n";
    }
};

// A doua clasă derivată care moștenește clasa A
class C : public A {
public:
    void afisare() {
        std::cout << "Afisare din clasa C\n";
    }
};

// Clasa derivată finală care moștenește atât clasa B, cât și clasa C
class D : public B, public C {
public:
    void afisare() {
        // Pentru a evita ambiguitatea, putem specifica din care clasă de
bază vrem să apelăm metoda
        B::afisare();
        C::afisare();
        std::cout << "Afisare din clasa D\n";
    }
};

int main() {
    // Creăm un obiect al clasei D
    D d;

    // Apelăm metoda afisare a clasei D
    d.afisare();
```

```
    return 0;
}
```



În acest exemplu, clasa `D` moștenește atât clasa `B`, cât și clasa `C`, care ambele moștenesc clasa `A`. Astfel, clasa `D` are două instanțe ale clasei `A` în ierarhia sa de moștenire. Atunci când apelăm metoda `afisare()` pentru obiectul clasei `D`, avem două căi posibile pentru a accesa această metodă: una prin clasa `B` și alta prin clasa `C`. Pentru a evita ambiguitatea, putem specifica din care clasă de bază dorim să apelăm metoda, așa cum am făcut în exemplul de mai sus.

Apelarea constructorilor în exemplul anterior

În acest caz observăm că a fost apelat de două ori constructorul clasei A. Simplificând exemplul obținem:

```
#include <iostream>

// Clasa de bază
class A {
public:
    A() {
        std::cout << "Constructor din clasa A\n";
    }

    void afisare() {
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
        std::cout << "Afisare din clasa A\n";
    }
};

// Prima clasă derivată care moștenește clasa A
class B : public A {
public:
    B() {
        std::cout << "Constructor din clasa B\n";
    }
};

// A doua clasă derivată care moștenește clasa A
class C : public A {
public:
    C() {
        std::cout << "Constructor din clasa C\n";
    }
};

// Clasa derivată finală care moștenește atât clasa B, cât și clasa C
class D : public B, public C {
public:
    // Constructorul clasei D
    D() {
        std::cout << "Constructor din clasa D\n";
    }
};

int main() {
    // Creăm un obiect al clasei D
    D d;

    //incercam sa afisam - Decomentați linia următoare
    //d.afisare();

    return 0;
}
```

S-a afișat:

```
Constructor din clasa A
Constructor din clasa B
Constructor din clasa A
Constructor din clasa C
Constructor din clasa D
```

Dacă decomentăm linia 45 programul nu funcționează: Non-static member 'afisare' found in multiple base-class subobjects of type 'A': class D; B; A class D; C; A.

Compilatorul „nu știe din ce clasă de bază A să ia funcția”.

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Moștenirea virtuală

Moștenirea virtuală este un concept în C++ care este utilizat pentru a evita ambiguitatea în cazul moștenirii multiple atunci când o clasă derivată moștenește din mai multe clase de bază care împărtășesc o clasă de bază comună. Moștenirea virtuală asigură că există o singură instanță a clasei de bază comună în clasa derivată.

Pentru a utiliza moștenirea virtuală, se utilizează cuvântul cheie `virtual` înainte de specificatorul de acces `public` în declarația unei clase de bază în clasa derivată.

Iată un exemplu care ilustrează modul de utilizare a moștenirii virtuale:

```cpp
#include <iostream>

// Clasa de bază
class A {
public:
    A() {
        std::cout << "Constructor din clasa A\n";
    }

    void afisare() {
        std::cout << "Afisare din clasa A\n";
    }
};

// Prima clasă derivată care moștenește clasa A
class B : virtual public A {
public:
    B() {
        std::cout << "Constructor din clasa B\n";
    }
};

// A doua clasă derivată care moștenește clasa A
class C : virtual public A {
public:
    C() {
        std::cout << "Constructor din clasa C\n";
    }
};

// Clasa derivată finală care moștenește atât clasa B, cât și clasa C
class D : public B, public C {
public:
    // Constructorul clasei D
    D() {
        std::cout << "Constructor din clasa D\n";
    }
};

int main() {
    // Creăm un obiect al clasei D
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
    D d;

    //incercam sa afisam
    d.afisare();

    return 0;
}
```

Observați adăugarea cuvântului cheie `virtual` la liniile 16 si 24. Am decomentat și linia 45 pentru că acum funcționează. S-a afișat

```
Constructor din clasa A
Constructor din clasa B
Constructor din clasa C
Constructor din clasa D
Afisare din clasa A
```

Reamintim ce am spus mai devreme: moștenirea multiplă are dezavantaje. Tocmai ce am activat virtualizarea (amintiți-vă din tutoriatul trecut). Mai mult, am creat un cod nu foarte prietenos: nu va putea fi portat în alte limbaje de programare dacă e nevoie (C++ printre singurele limbaje în care poți face așa ceva. În Python sau Ruby managementul pentru moșteniri multiple se face printr-un sistem de linearizare care transforma graful mostenrilor într-o listă, dar cam atât. Java, Rust, Go sau alte limbaje populare nu permit asta). Tot din categoria „cod prietenos”, codul e greu de inteles și se bazează pe faptul că baza e comună.

Ce va afișa acest cod?

```
#include <iostream>

// Clasa de bază
class A {
public:
    int x;
    A(int i): x(i) {
        std::cout << "Constructor din clasa A\n";
    }

    void afisare() {
        std::cout << "Afisare din clasa A\n" << x << std::endl;
    }
};
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
// Prima clasă derivată care moștenește clasa A
class B : virtual public A {
public:
    B(): A(10) {
        std::cout << "Constructor din clasa B\n";
    }
};

// A doua clasă derivată care moștenește clasa A
class C : virtual public A {
public:
    C(): A(20) {
        std::cout << "Constructor din clasa C\n";
    }
};

// Clasa derivată finală care moștenește atât clasa B, cât și clasa C
class D : public B, public C {
public:
    // Constructorul clasei D
    D(): B(), C() {
        std::cout << "Constructor din clasa D\n";
    }
};

int main() {
    // Creăm un obiect al clasei D
    D d;

    //incercam sa afisam
    d.afisare();

    return 0;
}
```

E.C.

De ce? Nu mai e suficient să instanțiem `A` din `B` și `C`. Practic, baza `A` e legată de obiectul de tip `D`.

Corectăm codul:

```
#include <iostream>

// Clasa de bază
class A {
public:
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
    int x;
    A(int i): x(i) {
        std::cout << "Constructor din clasa A\n";
    }

    void afisare() {
        std::cout << "Afisare din clasa A\n" << x << std::endl;
    }
};

// Prima clasă derivată care moștenește clasa A
class B : virtual public A {
public:
    B(): A(10) {
        std::cout << "Constructor din clasa B\n";
    }
};

// A doua clasă derivată care moștenește clasa A
class C : virtual public A {
public:
    C(): A(20) {
        std::cout << "Constructor din clasa C\n";
    }
};

// Clasa derivată finală care moștenește atât clasa B, cât și clasa C
class D : public B, public C {
public:
    // Constructorul clasei D
    D():  A(30), B(), C() {
        std::cout << "Constructor din clasa D\n";
    }
};

int main() {
    // Creăm un obiect al clasei D
    D d;

    //incercam sa afisam
    d.afisare();

    return 0;
}
```

Super! Acum să încercăm să afișăm `x` din `B` și `C`:

```
#include <iostream>

// Clasa de bază
class A {
public:
    int x;
    A(int i): x(i) {}

    void afisare() {
```

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

```
        std::cout << "Afisare din clasa A\n" << x << std::endl;
    }
};

// Prima clasă derivată care moștenește clasa A
class B : virtual public A {
public:
    B(): A(10) {}

    void afisare() {
        std::cout << "Afisare din clasa B\n" << x << std::endl;
    }
};

// A doua clasă derivată care moștenește clasa A
class C : virtual public A {
public:
    C(): A(20) {}

    void afisare() {
        std::cout << "Afisare din clasa C\n" << x << std::endl;
    }
};

// Clasa derivată finală care moștenește atât clasa B, cât și clasa C
class D : public B, public C {
public:
    // Constructorul clasei D
    D():  A(30), B(), C() {}
};

int main() {
    // Creăm un obiect al clasei D
    D d;

    //incercam sa afisam
    d.A::afisare();
    d.B::afisare();
    d.C::afisare();

    return 0;
}
```

Ce s-a afișat?
30 în toate cazurile.

Exerciții:

sursa: Tutoriat-POO-2022/Exercitii/T5.md at main · DimaOanaTeodora/Tutoriat-POO-2022 (github.com)

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Ex. 1:

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

class B
{
protected:
   int x;
public:
   B() { x = 78; }
};
class D1 : virtual public B
{
public:
   D1() { x = 15; }
};
class D2 : virtual public B
{
public:
   D2() { x = 37; }
};
class C : public D2, public D1
{
public:
   int get_x() { return x; }
};
int main()
{
   C ob;
   cout << ob.get_x();
   return 0;
}
```

R (font alb) :

Explicație (font alb):

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Ex. 2:

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

class A
{
public:
   int x;
   A(int i = 0) { x = i; }
   virtual A minus() { return (1 - x); }
};
class B : public A
{
public:
   B(int i = 0) { x = i; }
   void afisare() { cout << x; }
};
int main()
{
   A *p1 = new B(18);
   *p1 = p1->minus();
   p1->afisare();
   return 0;
}
```

R (font alb) :

Explicație (font alb):

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Ex. 3:

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

class A
{
public:
   int x;
   A(int i = 0) { x = i; }
   virtual A minus() { cout << x; return A(x - 1);
}
};
class B : public A
{
public:
   B(int i = 0) { x = i; }
   void afisare() { cout << x; }
};
int main()
{
   A *p1 = new A(18);
   *p1 = p1->minus();
   dynamic_cast<B *>(p1)->afisare();
   return 0;
}
```

R (font alb) :

Explicație (font alb):

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Ex. 4:

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

class A
{
public:
   int x;
   A(int i = 13) { x = i; }
};
class B : virtual public A
{
public:
   B(int i = 15) { x = i; }
};
class C : virtual public A
{
public:
   C(int i = 17) { x = i; }
};
class D : public A
{
public:
   D(int i = 19){x = i;}
};
class E : public B, public C, public D
{
public:
   int y;
   E(int i, int j) : D(i), B(j){y = x + i + j;}
   E(E &e) { y = -e.y; }
};
int main()
{
   E e1(1, 2), e2 = e1;
   cout << e2.y;
   return 0;
}
```

R (font alb) :

Explicație (font alb):

Materialul despre care am discutat la tutoriat și din care puteți învăța pentru examen (deși puteți învăța și din tutoriate :)))) Temele - mcmarius/poo: Laborator de

Radu-Tudor Vrinceanu <radu-tudor.vrinceanu@s.unibuc.ro> Tudor-Gabriel Miu <tudor-gabriel.miu@s.unibuc.ro>

Programare Orientată pe Obiecte - Facultatea de Matematică și Informatică, Universitatea din București (github.com)