Tutoriat 6

Polimorfism

Polimorfismul este ultimul principiu fundamental al paradigmei orientate pe obiect. Mai jos voi da 2 definiţii pentru această noţiune, una teoretică şi alta practică.

Def. *Polimorfismul* se referă la abilitatea unui obiect de a lua mai multe forme.

În practică, polimorfismul se referă la următoarele noţiuni:

- *overload / override (metode);*
- *operator overload;*
- *upcasting (pointer către clasa de bază);*
- *metode virtuale*;
- *clase abstracte*.

Un exemplu clasic de polimorfism:

- Un bărbat, de-a lungul vieţii, poate îndeplini mai multe funcţii: *iubit, soţ, tată, bunic…*

În continuare vom parcurge aceste noţiuni enumerate mai sus. Chiar dacă despre câteva dintre ele am mai vorbit, vom face o scurtă recapitulare.

Overload / Override

Despre noţiunile de *overload* şi *override* (supraîncărcare şi suprascriere) am mai discutat în tutoriatul 4 mai în detaliu. Ca o mică recapitulare, să mai trecem o dată prin diferenţele dintre cele 2 noţiuni:

- *Overload (supraîncărcarea)* = rescrierea unei metode în aceeaşi clasă, cu acelaşi nume, dar parametri diferiţi (numărul sau tipul lor). În cazul acesta, ambele metode sunt accesibile prin intermediul unui obiect;

- *Override (suprascriere)* = rescrierea unei metode într-o clasă derivată, adică metodele au acelaşi nume şi parametri. În cazul acesta, un obiect al clasei derivate nu poate accesa direct decât metoda din clasa sa, nu şi pe cea din clasa de bază. O poate accesa însă cu operatorul de rezoluţie.
  Exemple:

```
1 class B {
2   public:
3     void f() {
4       cout << "f()";
5     }
6
7     void f(int x) {
8        cout << "f(x)";
9     }
10 };
11
12 int main() {
13    B b;
14
15    b.f(); // f()
16    b.f(3) // f(x)
17 }
```

Overloading
(supraîncărcare)

```
1 class B {
2   public:
3     void f() {
4       cout << "B::f()";
5     }
6
7 };
8
9 class D : public B {
10   public:
11     void f() {
12       cout << "D::f()";
13     }
14 };
15
16 int main() {
17    D d;
18
19    d.f(); // D::f()
20    d.B::f(); // B::f()
21 }
```

Overriding
(suprascriere)

Vorbim de *polimorfism* pentru aceste 2 cazuri, din cauză că un obiect de un anumit tip poate avea comportament diferit pentru aceeaşi acţiune (metode cu corp diferit, dar acelaşi nume).

Operator Overloading (Supraîncărcare de operatori)

În C++ a fost introdusă o noţiune destul de interesantă şi care îi permite programatorului să realizeze soluţii elegante pentru anumite probleme.

Este permisă *supraîncărcarea operatorilor* pentru a realiza anumite operaţiuni diferite pentru anumite tipuri de obiecte. Acesta e un alt exemplu clasic de *polimorfism,* fiindcă acelaşi operator execută instrucţiuni diferite în funcţie de tipul obiectului.

LISTA DE OPERATORI CARE NU POT FI SUPRAÎNCĂRCAŢI

1.	***Operatorul * (pointer)***
2.	***Operatorul :: (rezoluţie)***
3.	***Operatorul .***
4.	***Operatorul ?:***
5.	***Operatorul sizeof***
6.	***Operatorul typeid***

Exemplu de supraîncărcare:

---------- *PAGINA URMĂTOARE* ----------

```
class B {
  public:
    int x;

    B(int i = 10) { x = i; }
    B operator+(B b) {
      return B(x + b.x);
    }
};

int main() {
   B b1, b2(20);

   B b3 = b1 + b2;
   cout << b3.x; // 30
}
```

Aici am supraîncărcat *operatorul +,* astfel încât de fiecare dată când se încearcă adunarea unui obiect de tip B, cel care apelează metoda (*b1,* în cazul nostru), cu alt obiect de tip B, adică parametrul metodei (*b2,* în cazul nostru), se va întoarce un nou obiect de tip B care primeşte ca parametru suma x-urilor celorlalte 2 obiecte.

Obs. Putem aduce diverse optimizări pentru metoda de mai sus, în sensul că putem face antetul să arate aşa:

```
B operator+(const B& b) { return B(x + b.x); }
```

Această modificare nu este vitală pentru funcţionarea operatorului. Este doar o chestiune de optimizare. Transmiterea prin referinţă duce la neapelarea constructorului de copiere, evitând astfel un apel care nu e necesar.

ATENŢIE: Când transmiteţi prin referinţă, ţineţi minte că orice modificare adusă referinţei se va reflecta asupra obiectului din main. Astfel, dacă nu vă doriţi asta, transmiteţi referinţa constantă ca mai sus.

Când supraîncărcaţi operatori, trebuie să vă gândiţi la el ca la o metodă obişnuită, determinând *tipul returnat* şi *parametri / parametru.* Există *operatori unari (*care nu primesc parametri), *operatori binari (*primesc doar 1 parametru).

Upcasting

Procedeul numit *upcasting* este una dintre cele mai importante noțiuni din OOP. Am vorbit mai în detaliu în tutoriatul 4 (pag 4). Acest proces se referă la *conversia* unui pointer către o clasă *derivată* la un pointer către clasa de *bază.*

```
1  class B { };
2
3  class D1 : public B { };
4  class D2 : public B { };
5
6  class D3 : public D1 { };
7
8  int main() {
9     B* b1 = new D1; //upcasting
10    B* b2 = new D2; //upcasting
11
12    D3 d3;
13    D1* d1 = &d3; //upcasting
14 }
```

Exemple de Upcasting

Upcasting-ul este un exemplu clasic pentru polimorfism, fiindcă avem o clasă (*bază)* care poate lua mai multe forme (*derivate).*

Metode Virtuale

(vezi tutoriat 5)

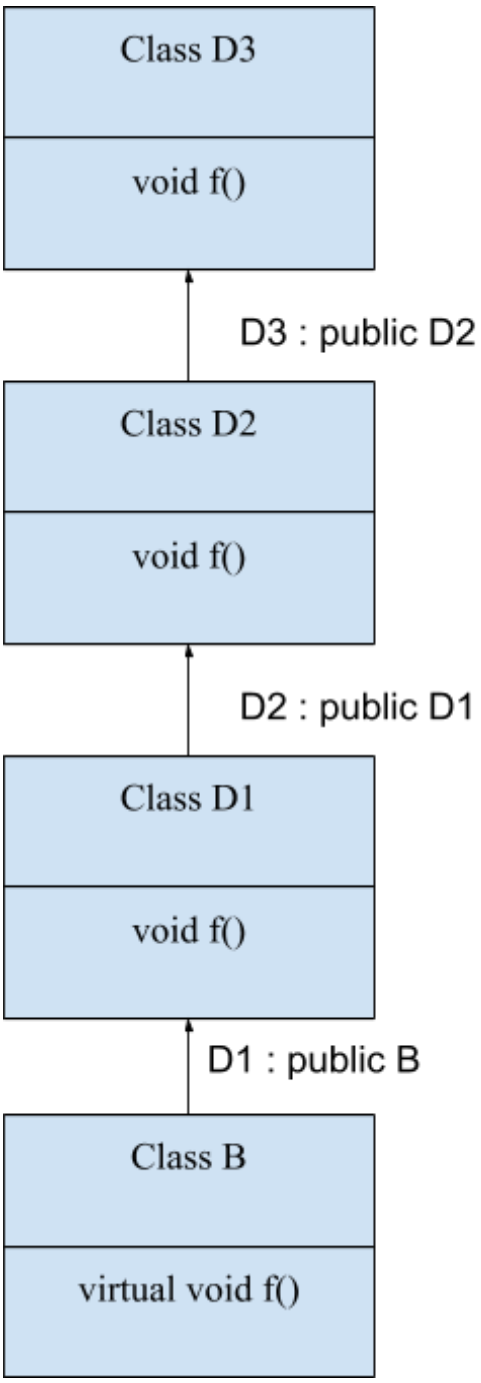
Cuvântul *virtual* se folosește în momentul *suprascrierii (overriding)* a unei metode din clasa de bază. Astfel, dacă în main facem upcasting la un pointer și încercăm să apelăm acea metodă, datorită lui *virtual* se va apela metoda declarată în clasa *derivată,* nu cea din clasa de *bază,* cum s-ar fi întâmplat fără virtual.

```cpp
class Shape {
  public:
    void draw() {
      cout << "Drawing a Shape";
    }

    virtual void erase() {
      cout << "Erasing a Shape";
    }
};

class Square : public Shape {
  public:
    void draw() {
      cout << "Drawing a Square";
    }

    void erase() {
      cout << "Erasing a Square";
    }
};

int main() {
  Shape* s = new Square; // upcasting

  s -> draw(); // Drawing a Shape
  s -> erase(); // Erasing a Square

  return 0;
}
```

Din punct de vedere al polimorfismului, *metodele virtuale* fac parte din categoria de *late-binding polymorphism (*adică polimorfism realizat la rulare), pentru că pointerul *s* are comportamente diferite (corp diferit al funcţiilor), pentru aceeaşi metodă, iar compilatorul îşi dă seama de asta abia la rularea programului, nu de la compilare. Asemănător ca la override, doar că aici *virtual* ”ştie” să apeleze exact metoda cu numele dat, cel mai ”sus” în lista de clase.

Spre exemplu, pentru următoarele clase:

---------- *NEXT PAGE* ----------



Avem următoarele apeluri şi rezultate:

---------- NEXT PAGE ----------

```
1 B* b = new D2;
2 b -> f(); // D2::f()
3
4 D2* d2 = new D3;
5 d2 -> f(); // D3::f()
6
7 D1* d1 = new D3;
8 d1 -> f(); // D3::f()
```

Aici putem observa următorul lucru: *oricum facem upcasting-ul, virtual-ul ajută la apelarea metodei corespunzătoare tipului clasei derivate.*

Clase abstracte

Există anumite situaţii în care dorim să modelăm anumite pattern-uri care au la bază trăsături comune. Putem realiza acest lucru în modul cunoscut până acum, cu clase obişnuite, dar acest lucru nu este indicat în anumite cazuri. De aceea, au apărut aceste *clase abstracte.*

Să ne gândim la următoarea situaţie: vrem să modelăm nişte clase de animale: *Câine*, *Pisică, ...*

- Ei bine, toate aceste animale au trăsături comune. Să spunem: ambele dorm, ambele mănâncă, …

- Astfel, putem spune că putem avea o clasă *Animal* în care se află aceste 2 metode, urmând ca *Pisică şi Câine* să moştenească din clasa *Animal*.

- Totuşi, fiecare face aceste lucruri în moduri diferite. Nu putem să spunem că un câine mănâncă la fel ca o pisică etc.

- Deci, deşi avem metode comune, implementarea diferă de la clasa *Câine,* la clasa *Pisică.* Aşa că, nu putem să oferim acestor metode implementare în clasa *Animal,* pentru că ar urma să le modificăm din mai jos în clasele derivate;

- Pentru a rezolva această problemă, au apărut clasele abstracte:

```cpp
class Animal {
  public:
    virtual void eat() = 0;
    virtual void sleep() = 0;
};

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

class Cat : public Animal {
  public:
    void eat() {
      cout << "Cat::eat()";
    }
    void sleep() {
      cout << "Cat::sleep()";
    }
};
```

Să analizăm codul de mai sus:

```cpp
virtual void eat() = 0;
```

```cpp
virtual void sleep() = 0;
```

Metode pur virtuale

Aceste declarări de mai sus, chiar dacă par ciudate, sunt declarări a 2 *metode pur virtuale*. Ce înseamnă asta?

Metodă pur virtuală = metodă fără implementare care face clasa în care se află să devină o ***clasă abstractă.*** Astfel, din cauza acestor metode nu avem voie să instanţiem niciun obiect de tip *Animal.*

Obs. Dacă o clasă conţine cel puţin o *metodă pur virtuală*, atunci acea clasă devine *abstractă.*

Dacă nici clasa derivată nu oferă o implementare pentru *metoda pur virtuală*, atunci şi clasa derivată rămâne *abstractă*.

……

În lanţul de moşteniri, pentru ca o clasă să nu fie abstractă, ***E OBLIGATORIU*** ca acea clasă să implementeze ***TOATE*** metodele abstracte din clasele de bază.

Declarări legate de exemplul ANIMAL - DOG - CAT:

```
1 Animal a; // eroare - clasa abstracta
2
3 Dog d; // corect
4 Cat c; // corect
5
6 Animal* a = new Dog; // corect
7 Animal* a = new Cat; // corect
```