Tutoriat 4

Overload (Supraîncărcare) vs Override (Suprascriere)

Înainte de a putea vorbi despre cuvântul cheie *virtual,* trebuie să lămurim diferențele dintre 2 noțiuni extraordinar de importante în OOP.

Overloading

Supraîncărcarea se referă la declararea a 2 funcții, în interiorul aceleași clase care au *același nume,* dar *antet diferit.*

```
1 class A {
2    public:
3       int f() { }
4       int f(int x) { }
5 };
```

În C, nu era posibilă această declarare. În schimb, în C++, avem voie să declarăm 2 funcții / metode cu același nume, atât timp cât respectăm următoarele reguli:

- *tipul parametrilor diferă;*
- *numărul parametrilor diferă;*

Atenție: Tipul returnat NU contează în supraîncărcare.

```
1 class A {
2   public:
3      int f() {}
4      void f() {}
5 };
6
7 int main() {
8    A a;
9    a.f(); // eroare
10   return 0;
11 }
```

Bineînţeles, supraîncărcarea se poate realiza şi cu funcţiile obişnuite, din afara unei clase.

Ascunderea metodelor (Hiding)

Când vorbim de supraîncărcare, poate apărea problema ascunderii metodelor clasei de *bază.* Să urmărim exemplul:

```
1 class B {
2   public:
3      void f() {}
4 };
5
6 class D : public B {
7   public:
8      void f(int x) {}
9 };
10
11 int main() {
12    D d;
13    d.f(); // eroare
14    return 0;
15 }
```

În cazul acesta, am crede că în clasa D am *supraîncărcat* metoda f, adăugându-i parametrul x. Ce am făcut de fapt, a fost să ascundem metoda *f* din clasa *B* faţă de clasa *D.*

Declararea unei metode cu acelaşi nume în clasa Derivată, deşi parametri diferă, duce la lipsa accesului direct asupra metodei cu acelaşi nume din clasa Bază.

Pentru a rezolva această problemă vom folosi operatorul de rezoluţie ”::”, astfel:

---------- *CONTINUAREA PE PAGINA URMATOARE* ----------

```cpp
class B {
  public:
     void f() { cout << "1";}
};

class D : public B {
  public:
     void f(int x) {}
};

int main() {
   D d;
   d.B::f(); // 1
   return 0;
}
```

Overriding

Suprascrierea unor metode/funcţii se realizează asemănător cu *supraîncărcarea* lor. Diferenţele majore sunt:

- *Suprascrierea se realizează la moştenire;*
- *La suprascriere, metodele au acelaşi nume şi aceeaşi parametri;*

Practic, în suprascriere, *metoda suprascrisă este înlocuită complet de cea care suprascrie.*

Pentru a apela metoda suprascrisă, ne vom folosi tot de *operatorul de rezoluţie.*

```cpp
class B {
  public:
     void f() { cout << "1";}
};

class D : public B {
  public:
     void f() { cout << "2";}
};

int main() {
   D d;
   d.B::f(); // 1
   d.f(); // 2
   return 0;
}
```

Upcasting vs Downcasting

Upcasting

Upcasting este un concept important din OOP legat de principiul moştenirii. Acesta implică:

Orice referinţă sau pointer către o clasă DERIVATĂ poate fi convertită către o referinţă sau pointer către clasa de BAZĂ.

Exemplu concret:

```
1 class Shape {};
2
3 class Square : public Shape {};
4 class Circle : public Shape {};
5 /* ... */
6
7 int main() {
8    Shape* s1 = new Square;
9    Shape* s2 = new Circle;
10   /* ... */
11 }
```

Upcasting

Această noţiune este folosită în practică în momentul în care se intenţionează declararea unui tablou de elemente, neştiindu-se de ce tip sunt.

Pentru exemplul de mai sus: ***ştiu*** că am *n* forme geometrie, dar nu ştiu de ce fel este fiecare. Astfel, creez un tablou cu elemente de tip *Shape* şi adaug în el câte un *Square, Circle…* depinde de ce am nevoie.

Atenţie: Upcasting şi Downcasting funcţionează **DOAR** cu pointeri şi referinţe.

IMPORTANT

La upcasting, pointerul are acces DOAR la datele şi metodele din clasa de bază.

```
1 class Shape {
2   public:
3     void area() {}
4 };
5
6 class Square : public Shape {
7   public:
8      void f() {}
9 };
10
11 int main() {
12    Shape* s1 = new Square;
13
14    s1 -> area(); // corect
15    s1 -> f(); // eroare
16 }
```

Deşi *s1* e iniţializat apelând constructorul clasei *Square (linia 12),* nu am acces decât la datele şi metodele din tipul cu care a fost declarat *s1,* adică *Shape.*