Tutoriat 4

Cuprins

- Overloading vs Overriding
  - Overloading (Supraîncărcarea)
  - Ascunderea metodelor (Hiding)
  - Overriding (Suprascriere)
- Upcasting vs Downcasting
  - Upcasting
    - Sintaxa
    - Explicații
    - De ce avem nevoie de asta?
    - Condiții
  - Downcasting (pe scurt)

Overload (Supraîncărcare) vs Override (Suprascriere)

Overloading (Supraîncărcarea)

Supraîncărcarea se referă la declararea a 2 sau mai multe *metode*, în interiorul aceleiași clase care au ***nume identic***, dar ***antet diferit***.

```
class A {
    public:
        int f() { /* ... */ }
        int f(int x) { /* ... */ }
};
```

în C++, avem voie să declarăm *funcții / metode* cu același nume, atât timp cât respectăm următoarele reguli:

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

Atenție: *Tipul returnat* **NU** contează în supraîncărcare.

```
class A {
    public:
        int f() { /* ... */ }
        void f() { /* ... */ }
};

int main() {
    A a;
    a.f(); // eroare
```

```
    return 0;
}
```

Observație: Supraîncărcarea se poate realiza și cu funcțiile obișnuite, din afara unei clase.

Ascunderea metodelor (Hiding)

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

```
class Baza {
    public:
        void f() { /* ... */ }
};

class Derivata : public Baza {
    public:
        void f(int x) { /* ... */ }
};

int main() {
    Derivata d;
    d.f(); // eroare
    return 0;
}
```

În cazul acesta, am crede că în clasa `Derivata` am *supraîncărcat* metoda `f`, adăugându-i parametrul `x`.

Ce am făcut de fapt, a fost să *ascundem metoda* `f` din clasa `Baza` față de clasa `Derivata`.

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

Rezolvare: Operatorul de rezoluție (`::`)

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

class Derivata : public Baza {
    public:
        void f(int x) { /* ... */ }
};

int main() {
    Derivata d;
    d.Baza::f(); // 1
```

```
    return 0;
}
```

Overriding (Suprascriere)

Suprascrierea unor metode/funcții se realizează asemănător cu *supraîncărcarea* lor.

Diferențele majore:

- Suprascrierea se realizează la **moștenire**;
- La suprascriere, metodele au **același nume** și **aceiași 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 rezoluție (`::`)**.

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

class Derivata : public Baza {
    public:
        void f() { cout << "2"; }
};

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

Upcasting vs Downcasting

Upcasting

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

Exemplu:

```
class Shape {};

class Square : public Shape {};
class Circle : public Shape {};
/* ... */

int main() {
```

```
    Shape* s1 = new Square;
    Shape* s2 = new Circle;
}
```

Sintaxă

```
Clasa_de_baza* pointer = new Clasa_derivata;
```

Explicații

Declarările de mai sus sunt posibile datorită **MOȘTENIRII PUBLICE** dintre clase.

Ținem minte de la tutoriatul despre moștenire:

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

Aici putem pune aceeași întrebare:

- A Square *IS A* Shape?
- A Circle *IS A* Shape?

Răspunsul este DA, astfel au sens și declarările:

```
Shape* s1 = new Square; // a new shape that is particularly a square
Shape* s2 = new Circle; // a new shape that is particularly a circle
```

De ce avem nevoie de asta?

Vectori de tip variabil

În general, *upcasting-ul* este folositor când avem de lucrat cu vectori de obiecte care pot fi de mai multe tipuri.

Exemplu:

Se dă un nr *n* de figuri geometrice care pot fi: *pătrate*, *cercuri*. Citiți și memorați figurile geometrice.

Rezolvare (fără upcasting):

```
class Shape {};

class Square : public Shape {};
class Circle : public Shape {};

int main() {
    int n;
    cin >> n;
```

```
    vector<Square> patrate;
    vector<Circle> cercuri;

    for (int i = 0; i < n; ++i) {
        char optiune;
        cout << "Patrat sau cerc (P / C): ";
        cin >> optiune;

        switch (optiune) {
            case 'P': {
                /* E nevoie de pointer pentru ca altfel obiectul se
distruge la iesirea din case */
                Square* s = new Square;

                cin >> *s; // trebuie implementat operatorul de citire

                patrate.push_back(*s);
            }

            case 'C': {
                Circle* c = new Circle; // la fel ca sus
                cin >> *c; // operatorul de citire
                cercuri.push_back(*c);
            }
        }
    }

    return 0;
}
```

Problema cu această abordare este că am declarat *2 vectori* pentru a reține formele geometrice.

Dacă dorim *extinderea* problemei prin crearea mai multor forme geometrice, atunci va fi necesară **declararea altor vectori**.

Rezolvare (cu upcasting):

```
class Shape {};

class Square : public Shape {};
class Circle : public Shape {};

int main() {
    int n;
    cin >> n;

    vector<Shape*> formeGeometrice;

    for (int i = 0; i < n; ++i) {
        char optiune;
        cout << "Patrat sau cerc (P / C): ";
        cin >> optiune;
```

```
        switch (optiune) {
            case 'P': {
                /* E nevoie de pointer pentru ca altfel obiectul se
distruge la iesirea din case */
                Square* s = new Square;

                cin >> *s; // trebuie implementat operatorul de citire

                formeGeometrice.push_back(s);
            }

            case 'C': {
                Circle* c = new Circle; // la fel ca sus
                cin >> *c; // operatorul de citire
                formeGeometrice.push_back(c);
            }
        }
    }

    return 0;
}
```

Prin *upcasting*, nu avem nevoie decât de *un singur vector*, indiferent câte alte forme geometrice mai implementăm.

Condiții pentru upcasting:

Să repetăm condițiile pentru ca upcasting-ul să fie posibil:

- **TREBUIE** să existe relație de moștenire între clase;
- **TREBUIE** ca moștenirea să fie de tip **PUBLIC**;

Observație

Este posibil să se realizeze upcasting și între obiecte alocate static, deși **NU ESTE O PRACTICĂ BUNĂ**.

```
class Shape {};

class Square : public Shape {};
class Circle : public Shape {};

int main() {
    Square s;

    Shape s1 = (Shape)s;
    return 0;
}
```

Downcasting

Procesul invers celui de *upcasting*. Vom vorbi mai în detaliu despre el la lecția despre **virtual**.

Presupune revenirea de la *clasa derivată* la *clasa de bază*.

Exemplu (not best practice):

```
class Shape {};

class Square : public Shape {};
class Circle : public Shape {};

int main() {
    Shape* s1 = new Square;
    /* Modifica s1 in functie de cerinte */

    Shape* s2 = (Shape*)s1; // functioneaza in cazul asta, dar nu e o
conversie recomandata

    return 0;
}
```

Vom detalia acest procedeu în tutoriatele următoare, unde vom introduce și modul corect de a se realiza folosind operatorul **dynamic_cast**.