Tutoriat 3

Cuprins

Cuvântul cheie *const*

Sunt anumite situații în proiectarea unei clase în care este necesar ca anumite date să nu poată să fie modificate.

Pentru a realiza acest lucru, avem nevoie de un *cuvant cheie* important pentru oop, dar care poate este și cauza multor ***bug-uri*** urâte și greu de identificat.

Cuvântul este ***const***

Exemple

- pentru o clasă `Student`, avem un câmp `dataNasterii`. Bineînțeles că acel câmp nu trebuie modificat, de aceea îl vom face `const`.
- pentru o clasă `CardBancar`, un câmp de date `numarCard` trebuie păstrat constant.
- ...

Exemple de declarări

```
const int x = 3;
const double y = 3.0;
```

Semnificație

Cuvântul `const` este un ***flag (semnal)*** către compilator care îi transmite să nu permită vreo modificare a valorii acelei variabile.

Orice încercare de modificare produce automat o ***eroare de compilare***.

Observație

Pentru a putea lucra cu o variabilă constantă, aceasta ***TREBUIE INIȚIALIZATĂ***.

Dacă omitem pasul acesta, vom primi o ***eroare de compilare***.

Pointeri constanți vs pointeri către constante

Când declarăm variabile constante obișnuite nu avem probleme cu ordinea cuvintelor din declarare. Adică:

```
const int x = 3;
int const x = 3; // same thing
```

Problemele apar când dorim declararea unor pointeri și avem nevoie și de const.

Aici sunt 2 tipuri de declarări:

```
const int* p;
int* const p; // not same thing
```

Pentru a evita confuzia, ne folosim de citirea tipului de date de la ***dreapta la stânga***:

```
const int* p; // pointer catre un intreg constant
              // pointerul se poate muta, dar valoarea NU se poate schimba

int x = 3;
p = &x; // ok
*p = 5; // not ok
```

```
int* const p; // pointer constant catre un intreg
              // pointerul NU se poate muta, dar valoarea se poate schimba

int x = 3;
p = &x; // not ok
*p = 5; // ok
```

Date membre constante

Ce ne interesează pe noi la oop, în mod special, sunt *membri* constanți dintr-o *clasă*.

Datele constante trebuie cumva inițializate, pentru că altfel sunt inutile. Problema este că nu putem face următorul lucru:

```
class Student {
    private:
        const string CNP;

    public:
        Student(string c) {
            CNP = c; // eroare
        }
};
```

Asupra unei variabile constante *NU* se poate aplica operatorul = în afara declarării.

Cum rezolvăm această problemă?

Listă de inițializare

Singura metodă corectă de a inițializa o dată constantă din ***CONSTRUCTOR*** este prin *lista de inițializare*.

```
class Student {
    private:
        const string CNP;

    public:
        Student(string c): CNP(c) {} // ok
};
```

Astfel, se evită folosirea operatorului *=*, iar *garanția* de constanță este respectată.

Observație

Putem bineînțeles să inițializăm și la declarare datele constante.

```
class Student {
    private:
        const string CNP = "1234567..."; // ok
};
```

Metode constante

Acum putem să vorbim despre o noțiune nouă din oop și care ne salvează de multe erori.

O *metodă constantă* **NU** are voie să schimbe nimic la datele membre ale pointer-ului `this`.

Exemplu

```
class Student {
    private:
```

```
        const string CNP;
        int varsta;

    public:
        int getVarsta() const {
            return varsta;
        }
};
```

Cuvântul cheie pus între `)` și `{` garantează compilatorului că metoda **NU** modifică în vreun fel pointer-ul `this`.

În general, metodele de tip *getter* sunt declarate constante.

FOARTE IMPORTANT

Cuvântul cheie **const** este **OBLIGATORIU** de specificat pentru ca o metodă să fie constantă.

Chiar dacă în metodă nu se modifică nimic, fără **const** compilatorul tot va vedea acea metodă ca fiind *neconstantă*.

```
class Student {
    private:
        const string CNP;
        int varsta;

    public:
        int getVarsta() const { // metoda constanta
            return varsta;
        }

        string getCNP() { // metoda neconstanta
            return CNP;
        }
};
```

De ce avem nevoie de metode constante?

Metodele constante sunt folosite pentru a lucra cu ***obiectele constante***.

Exemplu

```
class Student {
    private:
        const string CNP;
        int varsta;

    public:
        int getVarsta() const {
            return varsta;
```

```
        }

        string getCNP() {
            return CNP;
        }
};

int main() {
    const Student s; // obiect constant
    cout << s.getVarsta(); // ok
    cout << s.getCNP(); // eroare de compilare

    return 0;
}
```

Compilatorul vede metoda `getCNP` ca fiind *neconstantă*. Astfel, nu are garanția că nu are loc vreo modificare, așa că aruncă o eroare.

Dacă obiectul implicit rămâne nemodificat, *recomandat* este să folosiți **metode constante**.

Rezolvare

```
class Student {
    private:
        const string CNP;
        int varsta;

    public:
        int getVarsta() const {
            return varsta;
        }

        string getCNP() const {
            return CNP;
        }
};

int main() {
    const Student s; // obiect constant
    cout << s.getVarsta(); // ok
    cout << s.getCNP(); // ok

    return 0;
}
```

Alte erori comune provocate de const

- ***Referințe constante***

  Trebuie oferită *atenție sporită* metodelor care primesc ca parametri obiecte transmise prin *referință*.

Referința presupune că obiectul **POATE FI MODIFICAT**.

```
class Student {
    private:
        int varsta;

    public:
        void sum(Student& s) {
            cout << varsta + s.varsta;
        }
};

int main() {
    Student s1;
    const Student s2;

    s1.sum(s2); // eroare de compilare
}
```

Eroarea provine din secvența `s1.sum(s2)`, pentru că se încearcă apelarea unei metode care primește ca parametru o **referință** către un Student.

Orice modificare adusă unei variabile transmisă prin referință într-o funcție / metodă **ESTE SALVATĂ** și în afara funcției / metodei.

Obiectul `s2` este *constant*. Dacă e transmis ca parametru cu referință, atunci el poate oricând să fie modificat, ceea ce încalcă promisiunea cuvântului **const**.

!! IMPORTANT !! : Compilatorul nu verifică în interiorul metodei / funcției dacă obiectul chiar este modificat sau nu. El caută cuvântul cheie care să garanteze că va rămâne constant și dacă nu îl găsește întoarce o eroare.

Rezolvare

```
class Student {
    private:
        int varsta;

    public:
        void sum(const Student& s) {
            cout << varsta + s.varsta;
        }
};

int main() {
    Student s1;
    const Student s2;

    s1.sum(s2); // ok
}
```

Observatie: Cuvântul `const` adăugat la parametru face diferența dintre un cod care compilează și unul care nu.

Cuvântul cheie *static*

Variabilă statică într-o funcție

Această noțiune există încă din *limbajul C*. O *variabilă statică* se comportă ca o *variabilă globală* a unei funcții.

Exemplu

```
void f() {
    static int nr = 0;
    nr++;
    cout << nr << '\n';
}

int main() {
    f(); // 1
    f(); // 2
    f(); // 3
    // ...
}
```

Variabila este inițializată doar o dată la *primul apel al funcției*. Apoi, se execută doar liniile de cod în afara inițializării.

Din această cauză, o *variabilă statică* este ca un fel de *variabilă globală*.

Membri statici ai unei clase

Orice clasă poate să aibă *date* și *metode* statice. Comportamentul este asemănător ca la *variabilele statice* ale unei funcții.

Membri statici ai unei clase NU aparțin de o INSTANȚĂ. Ei aparțin de ÎNTREAGA CLASĂ

Ce înseamnă asta?

Un *membru static* are **aceeași valoare pentru orice instanță a clasei**.

Date statice

Sunt primele inițializate într-o clasă și nu aparțin direct de clasă.

NU pot fi inițializate în constructor.

2 tipuri de date statice:

1. ***neconstante*** = cele mai întâlnite

```
class Student {
    public:
        static int nrStudenti;
};

int Student::nrStudenti = 0;

int main() {
    cout << Student::nrStudenti << " "; // 0

    Student::nrStudenti++;
    cout << Student::nrStudenti << " "; // 1
}
```

Este important de reținut cum se inițializează un câmp static:

```
tip_de_date Clasa::nume_variabila = valoare_initiala;
```

Linia aceasta este **OBLIGATORIE** pentru funcționarea programului.

Dacă omitem acea linie nu primim imediat o *eroare de compilare*, ci numai dacă încercăm în vreun moment *să accesăm* acea dată statică.

Observație

ÎN PRACTICĂ, câmpurile statice se accesează folosind *operatorul de rezoluție* (`::`). Totuși, **ESTE PERMISĂ** și următoarea metodă de accesare:

```
class Student {
    public:
        static int nrStudenti;
};

int Student::nrStudenti = 0;

int main() {
    Student s;
    cout << s.nrStudenti << " "; // 0

    s.nrStudenti++;
    cout << s.nrStudenti << " "; // 1

    Student s1;
    cout << s1.nrStudenti << " "; // 1
}
```

IMPORTANT Accesarea câmpurilor statice printr-o instanță a clasei **NU** este recomandată pentru că se creează confuzie.

După cum am mai spus, *câmpurile statice* au aceeași valoare pentru toate instanțele. Așa că nu are rost să folosim o instanță să îl accesăm.

2. ***constante***

```
class Student {
    public:
        const static int nrStudenti = 10;
};

int main() {
    cout << Student::nrStudenti; // 10
}
```

Observație

Metoda de inițializare de la *neconstante* **NU** funcționează aici, cum nici invers nu e cazul.

Metode statice

Se respectă aceeași globalitate ca pentru orice *static*.

O metodă statică poate fi inițializată fie *inline (în interiorul clasei)*, fie în *afara clasei*.

```
class Student {
    private:
        static int nrStudenti;

    public:
        static int getNrStudenti() {
            return nrStudenti;
        }

        static void setNrStudenti(int);
};

void Student::setNrStudenti(int n) {
    nrStudenti = n;
}

int Student::nrStudenti = 0;

int main() {
    Student::setNrStudenti(5);

    cout << Student::getNrStudenti(); // 5
}
```

Rămâne valabil ce am spus la date statice legat de modul de a accesa o metodă statică.

O metodă statică are acces direct doar la datele și metodele STATICE ale unei clase.

Erori comune provocate de static

- ***Dată statică neinițializată***

GRESIT

```
class Student {
    public:
        static string dataUltimeiActualizari;
};

int main() {
    cout << Student::dataUltimeiActualizari; // eroare de compilare
}
```

CORECT

```
class Student {
    public:
        static string dataUltimeiActualizari;
};

string Student::dataUltimeiActualizari = "22.03.2020";

int main() {
    cout << Student::dataUltimeiActualizari; // ok
}
```

Explicație

Prima secvență provoacă o *eroare de compilare*, din cauză că se încearcă folosirea unei *date statice neinițializate.*

Problema se rezolvă prin adăugarea liniei `string Student::dataUltimeiActualizari = "22.03.2020"`.

- **Membri nestatici în metode statice**

```
class A {
    private:
        int x;

    public:
        static int printX() {
            cout << x << '\n';
```

```
        }
};

int main() {
    A::printX(); // eroare de compilare
}
```

Explicație

O metodă statică **NU** aparține de clasă, așa că **NU** are pointerul **this**.

Din cauza asta, nu putem accesa un câmp non-static, pentru că metoda nu va ști de unde să ia acel câmp.

În plus, membri statici se inițializează **ÎNAINTEA** celor non-statici în momentul compilării.

Rezolvare (recomandată)

```
class A {
    private:
        int x;

    public:
        static int printX(A a) {
            cout << a.x << '\n';
        }
};

int main() {
    A a;
    A::printX(a); // ok
}
```

Această metodă statică nu are sens în practică, dar ca să o facem totuși să funcționeze avem nevoie de un **obiect** de tipul clasei să accesăm *câmpul de date.*

În general, când construim o *metodă statică*, nu prea este nevoie să accesăm *câmpuri non-statice* dacă metoda a fost gândită bine înainte.

Observație

În *metoda statică* de mai sus am accesat un *câmp private* al clasei fără *getter*:

```
static int printX(A a) {
    cout << a.x << '\n';
}
```

Acest lucru este posibil, pentru că, deși o **metodă statică** nu are pointerul **this**, aceasta este o **metodă a clasei**.

Astfel, are acces la membri clasei.