AVANSAT · POO ÎN C++ · LECȚIA 3

Supraîncărcarea operatorilor: aritmetică, comparații, <<, >>, [], (), friend

Durată: 80 min Nivel: avansat / pregătire pentru facultate și industrie Înainte: lecțiile oop-01, oop-02

CE VEI ȘTI DUPĂ LECȚIE

- Supraîncarci operatori aritmetici, de comparație, de incrementare și de indexare pentru propriile clase.
- Alegi între operator membru și operator liber (friend) și explici de ce.
- Scrii operator<< și operator>> pentru afișare și citire cu cin/cout.
- Folosești obiecte-funcție (operator()) ca și comparatori pentru sort, set și priority_queue.
- Respecți regulile de bun-simț: semantică naturală, + implementat prin +=, fără surprize.

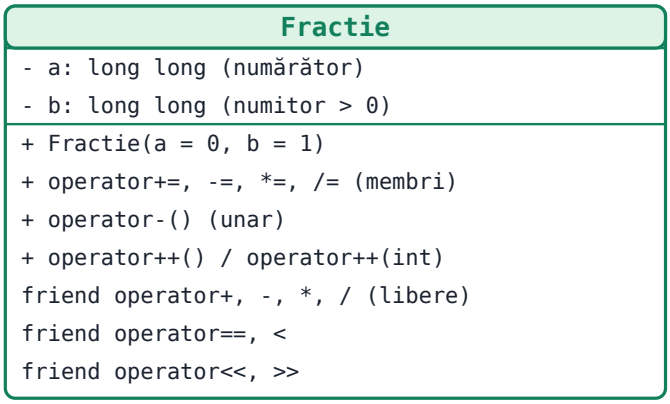
1. De ce operatori?

Pentru `int` scrii `a + b * c`. Pentru o clasă `Fractie` ai putea scrie `a.aduna(b.inmulteste(c))`, dar e greu de citit. **Supraîncărcarea operatorilor** îți permite să dai sens operatorilor obișnuiți pentru tipurile tale: `a + b * c` funcționează și pentru fracții, numere mari, vectori geometrici, matrice. Așa sunt făcute și clasele din biblioteca standard: `string` are `+` și `==`, `cout` are `<<`, `vector` are `[]`.

Un operator este, de fapt, o **funcție** cu un nume special:

Scrii	Compilatorul înțelege (membru)	sau (funcție liberă)
`a + b`	`a.operator+(b)`	`operator+(a, b)`
`a += b`	`a.operator+=(b)`	`operator+=(a, b)`
`-a`	`a.operator-()`	`operator-(a)`
`cout << a`	– (nu putem modifica clasa `ostream`)	`operator<<(cout, a)`
`v[i]`	`v.operator[](i)`	– (doar membru)
`f(x, y)`	`f.operator()(x, y)`	– (doar membru)

2. Exemplul complet: clasa Fractie



Clasa Fractie

Clasa păstrează un **invariant**: fracția este ireductibilă, iar numitorul este pozitiv. Fiecare operație se termină cu `simplifica()`, deci invariantul nu se strică niciodată, iar comparația `==` se reduce la compararea a două numere.

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

class Fractie
{
    long long a, b;                         // invariant: b > 0, cmmdc(|a|, b) = 1
    void simplifica()
    {
        if (b == 0) throw invalid_argument("numitor 0");
        if (b < 0) { a = -a; b = -b; }
        long long d = gcd(a < 0 ? -a : a, b);
        if (d > 1) { a /= d; b /= d; }
    }
public:
    Fractie(long long x = 0, long long y = 1) : a(x), b(y) { simplifica(); }

    Fractie& operator+=(const Fractie &f) { a = a * f.b + f.a * b; b *= f.b; simplifica();
return *this; }
    Fractie& operator-=(const Fractie &f) { a = a * f.b - f.a * b; b *= f.b; simplifica();
return *this; }
    Fractie& operator*=(const Fractie &f) { a *= f.a; b *= f.b; simplifica(); return *this;
}
    Fractie& operator/=(const Fractie &f) { a *= f.b; b *= f.a; simplifica(); return *this;
}

    Fractie operator-() const { return Fractie(-a, b); }            // minus unar
    Fractie& operator++() { a += b; return *this; }                 // ++f (prefix)
    Fractie operator++(int) { Fractie vechi = *this; a += b; return vechi; }   // f++
(postfix)

    friend Fractie operator+(Fractie x, const Fractie &y) { return x += y; }
    friend Fractie operator-(Fractie x, const Fractie &y) { return x -= y; }
    friend Fractie operator*(Fractie x, const Fractie &y) { return x *= y; }
    friend Fractie operator/(Fractie x, const Fractie &y) { return x /= y; }
    friend bool operator==(const Fractie &x, const Fractie &y) { return x.a == y.a && x.b
== y.b; }
    friend bool operator!=(const Fractie &x, const Fractie &y) { return !(x == y); }
    friend bool operator<(const Fractie &x, const Fractie &y) { return x.a * y.b < y.a *
x.b; }
    friend ostream& operator<<(ostream &out, const Fractie &f)
    {
        out << f.a;
        if (f.b != 1) out << '/' << f.b;
        return out;
    }
    friend istream& operator>>(istream &in, Fractie &f)
    {
        long long x, y;
        if (in >> x >> y) f = Fractie(x, y);
        return in;
    }
};

int main()
{
```

```
    Fractie p, q;
    cin >> p >> q;
    cout << p << " + " << q << " = " << p + q << '\n';
    cout << p << " - " << q << " = " << p - q << '\n';
    cout << p << " * " << q << " = " << p * q << '\n';
    cout << p << " / " << q << " = " << p / q << '\n';
    cout << (p < q ? "p < q" : (p == q ? "p = q" : "p > q")) << '\n';
    Fractie r = p++;
    cout << r << ' ' << p << ' ' << -p + 1 << '\n';
    return 0;
}
```

Pe intrarea `-4 6 2 -3`, constructorul aduce ambele fracții la `-2/3` (numitorul devine pozitiv, apoi se simplifică), deci `p == q`.

DE CE AȘA?

De ce `+=` ca membru și `+` ca funcție liberă? `+=` modifică obiectul din stânga, deci este natural să fie membru și să întoarcă `*this` prin referință. `+` creează un obiect **nou** și nu modifică operanzii: îl scriem o singură dată, **prin** `+=` (primește `x` prin valoare, deci o copie, îl modifică și îl întoarce). Așa nu duplicăm formula. **De ce liber și nu membru?** Ca să meargă și `1 + f`: un operator membru cere ca operandul din **stânga** să fie deja `Fractie`; unul liber permite conversia implicită a lui `1` în `Fractie(1)` prin constructor, pe oricare parte.

3. `friend`: acces la membrii privați

O funcție declarată `friend` în interiorul clasei **nu** este membră (nu are `this`), dar are voie să citească membrii `private`. Este exact ce trebuie pentru `operator<<`, `operator+` sau `operator==`, care au nevoie de `a` și `b`, dar nu pot fi membri (sau nu e bine să fie).

DE ȘTIUT

`friend` nu strică încapsularea dacă o folosești cu măsură: funcția prietenă face parte din **interfața** clasei, este declarată chiar în clasă și o scrie tot autorul clasei. Ce trebuie evitat este să faci prietene clase întregi doar ca să ocolești `private`.

4. `<<` și `>>`: afișare și citire

```
friend ostream& operator<<(ostream &out, const Fractie &f);   // cout << f
friend istream& operator>>(istream &in, Fractie &f);          // cin >> f
```

- Primul parametru este fluxul (`cout`, un `ofstream`, un `stringstream`): prin referință, fiindcă fluxurile nu se pot copia.
- **Întorc fluxul prin referință**, ca să meargă înlănțuirea: `cout << a << " " << b` înseamnă `operator<<(operator<<(operator<<(cout, a), " "), b)`.
- La `>>`, obiectul se primește prin referință **ne-constantă** (îl modificăm), iar dacă citirea eșuează nu atingem obiectul.

5. Incrementare: prefix și postfix

Forma	Semnătura	Întoarce
`++f` (prefix)	`T& operator++()`	obiectul **după** incrementare, prin referință
`f++` (postfix)	`T operator++(int)`	o **copie** a valorii vechi

Parametrul `int` din postfix nu se folosește: este doar un semn care deosebește cele două forme. Postfixul face o copie în plus; de aceea, în bucle, `++it` este preferat lui `it++` pentru iteratori și obiecte mari.

6. Indexare `[]` și obiecte-funcție `()`

Operatorul `[]`

Se scrie în două variante: una care permite modificarea (`v[i] = 5`) și una `const`, pentru obiecte constante:

```
int& operator[](int i) { return p[i]; }                // pentru v[i] = 5
const int& operator[](int i) const { return p[i]; }    // pentru citire dintr-un obiect const
```

Operatorul `()`: obiecte-funcție (functori)

O clasă cu `operator()` se poate „apela” ca o funcție, dar spre deosebire de o funcție poate **ține stare** (de exemplu, un prag ales la rulare). Biblioteca standard le folosește peste tot ca **comparatori**.

```cpp
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>
using namespace std;

struct Elev
{
    string nume;
    int nota;
};

struct DupaNota                          // obiect-funcție: compară după notă, descrescător
{
    bool operator()(const Elev &x, const Elev &y) const
    {
        if (x.nota != y.nota) return x.nota > y.nota;
        return x.nume < y.nume;
    }
};

class PesteMedie                         // obiect-funcție cu stare: pragul este ales la rulare
{
    int prag;
public:
    explicit PesteMedie(int p) : prag(p) {}
    bool operator()(const Elev &e) const { return e.nota >= prag; }
};

int main()
{
    int n, prag;
    cin >> n >> prag;
    vector<Elev> v(n);
    for (auto &e : v) cin >> e.nume >> e.nota;
    sort(v.begin(), v.end(), DupaNota());                    // comparatorul este un obiect
    for (int i = 0; i < 3 && i < n; i++) cout << v[i].nume << ' ' << v[i].nota << '\n';
    cout << count_if(v.begin(), v.end(), PesteMedie(prag));  // câți au cel puțin pragul
    return 0;
}
```

SFAT

Același comparator se poate da unui `set` sau unui `priority_queue` ca **parametru de tip**: `set<Elev, DupaNota> s;`. O **lambda** (lecția oop-07) este, de fapt, un obiect-funcție scris pe scurt, generat de compilator.

7. Operatori pentru olimpiadă: matrice și ridicare la putere

Cu `operator*` pentru matrice și ridicarea la putere rapidă (lecția co-09), termenul n al șirului lui Fibonacci se obține în O(log n): `[[1,1],[1,0]]ⁿ` conține `F(n)` pe poziția din dreapta-sus.

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

struct Mat2
{
    long long m[2][2];
    Mat2 operator*(const Mat2 &o) const    // produs de matrice 2 × 2
    {
        Mat2 r{};
        for (int i = 0; i < 2; i++)
            for (int j = 0; j < 2; j++)
                for (int k = 0; k < 2; k++)
                    r.m[i][j] += m[i][k] * o.m[k][j];
        return r;
    }
};

Mat2 putere(Mat2 a, long long n)           // a^n, același algoritm ca la numere
{
    Mat2 r{{{1, 0}, {0, 1}}};              // matricea unitate
    while (n > 0)
    {
        if (n & 1) r = r * a;
        a = a * a;
        n >>= 1;
    }
    return r;
}

int main()
{
    long long n;
    cin >> n;
    Mat2 q{{{1, 1}, {1, 0}}};
    cout << putere(q, n).m[0][1];          // F(n)
    return 0;
}
```

DE CE AȘA?

De ce funcționează același algoritm? Ridicarea la putere rapidă cere doar ca înmulțirea să fie **asociativă**, iar înmulțirea matricelor este. Supraîncărcând `*`, funcția `putere` arată exact ca pentru numere. Aceasta este puterea operatorilor: algoritmii generali funcționează pe tipurile tale (iar cu șabloane, lecția oop-06, poți scrie `putere` o singură dată pentru orice tip).

8. Reguli de bun-simț

Regula	De ce
păstrează sensul natural (`+` adună, `==` compară)	cine citește codul se așteaptă la asta
implementează `+` prin `+=`, `!=` prin `==`, `>` prin `<`	o singură formulă, fără contradicții

Regula	De ce
operatorii care modifică întorc `T&`, cei care creează valori întorc `T`	se comportă ca la tipurile predefinite
comparațiile și aritmetica simetrică: funcții libere	conversiile funcționează pe ambele părți
`=`, `[]`, `()`, `->`: obligatoriu membri	regula limbajului
nu supraîncărca `&&`, `||`, `,`	își pierd evaluarea scurtcircuitată / ordinea
operatorii de conversie: `explicit operator bool() const`	evită conversii surprinzătoare

DE ȘTIUT

În C++20 operatorul „navă spațială” `<=>` poate genera automat toate cele șase comparații (`auto operator<=>(const T&) const = default;`). În lecții folosim C++17, de aceea le-am scris de mână; în proiecte noi, `<=>` scurtează mult codul.

9. Greșeli frecvente

Greșeala	Ce se întâmplă	Corect
`operator<<` membru al clasei	trebuie scris `f << cout`, invers	funcție liberă (friend)
`operator<<` care întoarce `void`	`cout << a << b` nu se compilează	întoarce `ostream&`
`operator+` care modifică operandul stâng	`c = a + b` schimbă și `a`	`+` întoarce o valoare nouă
`+=` care întoarce o copie	`(a += b) += c` nu modifică `a` de două ori	întoarce `T&`
lipsește varianta `const` a lui `[]`	nu se poate citi dintr-un obiect `const`	două variante
comparator `<=` dat lui `sort`	comportament nedefinit (ordine strictă cerută)	`<` strict
`==` scris independent de `<` cu altă logică	rezultate contradictorii	aceeași definiție de bază
depășire în `a * f.b` la fracții mari	rezultat greșit	simplificare înainte, `__int128` sau numere mari

10. Exersează pe hârtie

1. Ce funcție apelează compilatorul pentru `a + b`, `a += b`, `-a`, `cout << a`, dacă `+` este liber și `+=`, `-` sunt membri?

2. De ce `1 + Fractie(1, 2)` funcționează cu `operator+` liber, dar nu cu `operator+` membru?

3. Ce întoarce `++f` și ce întoarce `f++`? Ce afișează: `Fractie f(1, 2); cout << f++ << ' ' << ++f;`?

4. Scrie `operator-=` pentru o clasă `Vector2D { double x, y; }` și apoi `operator-` prin el.

5. De ce `operator<<` întoarce `ostream&`?

6. Pentru `Fractie(6, -8)`, ce valori au `a` și `b` după construire?

7. Scrie un functor `ModuloK` cu un membru `k` care întoarce `x % k == 0`. Cum îl folosești cu `count_if`?

8. Ce face `Mat2 r{};` și de ce este important în `operator*`?

9. De ce comparatorul pentru `sort` trebuie să fie strict (`<`, nu `<=`)?

10. Ce operator ai supraîncărca pentru a putea scrie `if (f)` cu sensul „fracția nu este zero"? Cum îl faci să nu permită `int x = f;`?

11. Mini-grile

1. `operator<<` pentru afișare se scrie ca: a) metodă a clasei b) funcție liberă, de obicei `friend` c) metodă a lui `ostream` d) operator static

2. Postfixul `f++` are semnătura: a) `T& operator++()` b) `T operator++(int)` c) `void operator++()` d) `T operator+(int)`

3. Care operator **nu** se recomandă a fi supraîncărcat? a) `+` b) `[]` c) `&&` d) `==`

4. Un obiect-funcție este: a) un pointer la funcție b) un obiect al unei clase cu `operator()` c) o funcție statică d) o funcție friend

12. Mai departe (pentru cine vrea mai mult)

- `<=>` **(C++20)** și comparațiile generate automat.
- **Literali definiți de utilizator:** `3_kg`, `"abc"s`.
- **Numere mari** (adunare, înmulțire pe cifre) cu operatori: aplicație clasică la olimpiadă.
- `operator->` **și** `operator*` pentru clase care se comportă ca pointeri (smart pointers, iteratori).
- **Expression templates:** cum fac bibliotecile de matrice `a + b + c` fără temporare.

Rezumat

- Un operator este o funcție cu nume special (`operator+`); îl poți scrie ca membru sau ca funcție liberă.
- `+=`, `-=`, `++`, `[]`, `()` ca membri; `+`, `-`, `==`, `<`, `<<`, `>>` ca funcții libere (adesea `friend`).
- `+` se implementează prin `+=`; operatorii care modifică întorc `T&`, ceilalți `T`.
- `<<` și `>>` primesc fluxul prin referință și îl întorc, pentru înlănțuire.
- Obiectele-funcție (`operator()`) sunt comparatori cu stare pentru `sort`, `set`, `priority_queue`.
- Păstrează sensul natural al operatorilor; nu supraîncărca `&&`, `||`, `,`.

Citește și exersează în plus

- cppreference — operator overloading — toate formele canonice ale operatorilor (engleză)
  https://en.cppreference.com/w/cpp/language/operators
- learncpp.com — Introduction to operator overloading — capitolul complet, cu exerciții (engleză)
  https://www.learncpp.com/cpp-tutorial/introduction-to-operator-overloading/
- C++ Core Guidelines — Operators — reguli pentru operatori supraîncărcați (engleză)
  https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Ro-overload

Răspunsuri

Exersează pe hârtie

1. `operator+(a, b)`; `a.operator+=(b)`; `a.operator-()`; `operator<<(cout, a)`.

2. La un operator membru, operandul stâng trebuie să fie deja `Fractie` (`1.operator+(…)` nu există). La funcția liberă, `1` se convertește prin constructorul `Fractie(1)`, care nu este `explicit`.

3. `++f` întoarce obiectul incrementat, prin referință; `f++` întoarce o copie a valorii vechi. Afișează `1/2 5/2` (`f++` dă 1/2 și face f = 3/2; `++f` face f = 5/2).

4. `Vector2D& operator-=(const Vector2D &o) { x -= o.x; y -= o.y; return *this; }` și `Vector2D operator-(Vector2D a, const Vector2D &b) { return a -= b; }`.

5. Ca să poată fi înlănțuit: rezultatul lui `cout << a` este din nou `cout`, la care se aplică `<< b`.

6. Numitorul negativ: `a = -6`, `b = 8`; cmmdc 2: `a = -3`, `b = 4`.

7. `struct ModuloK { int k; bool operator()(int x) const { return x % k == 0; } };` și `count_if(v.begin(), v.end(), ModuloK{3})`.

8. Inițializează toate elementele cu 0. Fără el, `r.m[i][j] += …` ar aduna la valori nedefinite.

9. Algoritmii de sortare presupun o „ordine strictă slabă”: un element nu este mai mic decât el însuși. Cu `<=`, `x <= x` este adevărat și algoritmul poate citi în afara vectorului sau intra în buclă.

10. `explicit operator bool() const { return a != 0; }`. Cu `explicit`, `if (f)` funcționează (context boolean), dar `int x = f;` nu se compilează.

Mini-grile

1. b). **2.** b). **3.** c) `&&`. **4.** b).