INFORMATICĂ · CLASA A XI-A · POO (INTENSIV) · LECȚIA 21

Programare orientată pe obiecte în C++: clase și obiecte

Durată: 60 min Nivel: intensiv informatică (și introducere pentru clasa a XII-a) Înainte: lecțiile x-12, xi-02

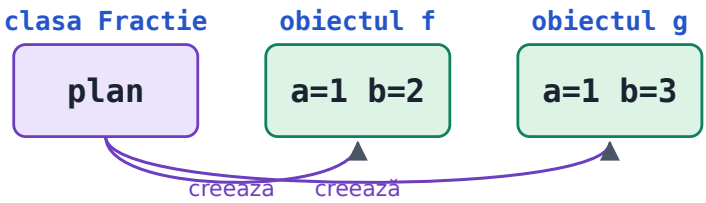
CE VEI ȘTI DUPĂ LECȚIE

- Înțelegi conceptele: clasă, obiect, date membre, metode, încapsulare.
- Scrii o clasă cu constructor, metode get/set și specificatori public/private.
- Cunoști ideile de moștenire și polimorfism.

1. De ce clase?

Până acum, datele (un `struct`) și funcțiile care le prelucrează stăteau separat: ai o `struct fractie` și funcții `aduna(f, g)` risipite prin program. În programele mari, oricine putea modifica orice, oricând, și apăreau erori greu de găsit (de exemplu, o fracție cu numitorul 0). Programarea orientată pe obiecte (**POO**) pune datele și funcțiile care au voie să le modifice **în același loc**: clasa.

Termen	Ce înseamnă	Comparație
clasă	tip nou: date + funcții (metode)	planul unei case
obiect	variabilă de tipul clasei	casa construită după plan
atribut (dată membră)	variabilă din interiorul clasei	camerele
metodă	funcție din interiorul clasei	ce poți face în casă
constructor	metoda care „construiește” obiectul	zidarii



din aceeași clasă faci oricâte obiecte; fiecare are propriile valori ale datelor

Clasă și obiect

2. Prima clasă: Dreptunghi

```
class Dreptunghi
{
private:                    // vizibil doar din interiorul clasei
    int L, l;
public:                     // vizibil de oriunde
    Dreptunghi(int a, int b)   // constructor: același nume ca clasa, fără tip returnat
    {
        L = a;
        l = b;
    }
    int arie()      { return L * l; }
    int perimetru() { return 2 * (L + l); }
};
```

Folosire: `Dreptunghi d(3, 4);` creează obiectul `d`; `d.arie()` apelează metoda pe acel obiect (ca la `struct`: `d.L`, dar aici `L` este privat).

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

class Dreptunghi
{
private:
    int L, l;
public:
    Dreptunghi(int a, int b) { L = a; l = b; }
    int arie()      { return L * l; }
    int perimetru() { return 2 * (L + l); }
};

int main()
{
    int a, b;
    cin >> a >> b;
    Dreptunghi d(a, b);
    cout << d.arie() << ' ' << d.perimetru();
    return 0;
}
```

Pas	Instrucțiune	Efect
1	`Dreptunghi d(3, 4);`	se apelează constructorul: L = 3, l = 4
2	`d.arie()`	returnează 3 · 4 = 12
3	`d.perimetru()`	returnează 2 · (3 + 4) = 14

DE CE AȘA?

De ce `private`? Dacă `L` ar fi public, oricine ar putea scrie `d.L = -5;` și obiectul ar ajunge într-o stare imposibilă. Cu `private`, singurele modificări posibile sunt cele pe care **clasa** le permite. **Care e diferența față de `struct`?** În C++, `struct` este o clasă cu membrii publici implicit; `class` are membrii privați implicit. Altfel sunt la fel.

3. Încapsulare: obiectul își apără datele



din main nu poți scrie cont.sold = 1000000; poți doar cont.depune(100) — iar depune verifică suma

Încapsulare

Un cont bancar trebuie să nu permită retrageri peste sold. Soldul este privat; singurele căi spre el sunt metodele, iar acestea **verifică**.

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

class Cont
{
private:
    int sold;
public:
    Cont() { sold = 0; }                    // constructor fără parametri
    bool depune(int suma)
    {
        if (suma <= 0) return false;        // sumă invalidă
        sold += suma;
        return true;
    }
    bool retrage(int suma)
    {
        if (suma <= 0 || suma > sold) return false;
        sold -= suma;
        return true;
    }
    int getSold() const { return sold; }    // „get”: citire fără modificare
};

int main()
{
    int n, suma;
    char op;
    Cont c;
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        cin >> op >> suma;
        bool ok = (op == 'D') ? c.depune(suma) : c.retrage(suma);
        if (ok) cout << c.getSold() << '\n';
        else    cout << "fonduri insuficiente\n";
    }
    return 0;
}
```

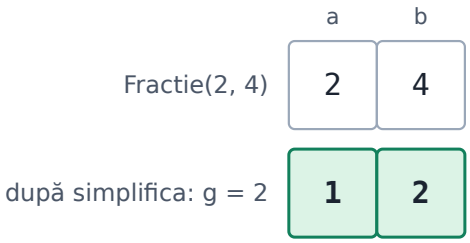
DE CE AȘA?

De ce `const` după `getSold()`? Spune compilatorului (și cititorului) că metoda **nu modifică** obiectul; poate fi apelată și pe obiecte constante. Metodele care doar citesc se declară `const`.
De ce `bool` la depune/retrage? Apelantul află dacă operația a reușit, fără ca clasa să afișeze mesaje (o clasă bine făcută nu are `cout` în interior, decide `main` cum informează utilizatorul).

Metodele `get` (citesc) și `set` (modifică, **cu verificare**) se numesc **accesori**. Nu face `set` pentru orice câmp „de formă”: dacă un `set` nu verifică nimic, atributul putea la fel de bine să fie public.

4. Constructorul: obiectul se naște corect

Constructorul rulează o singură dată, la crearea obiectului, și îl aduce într-o stare **validă**. Poate avea parametri, valori implicite și poate fi mai mulți (supraîncărcare).



obiectul ajunge în main deja corect (1/2), fără ca utilizatorul să-și amintească să simplifice

Constructor

Fracția este exemplul clasic: după constructor, ea este deja simplificată și are numitorul pozitiv.

```
class Fractie
{
private:
    int a, b;                                       // a / b
    static int cmmdc(int x, int y) { return y == 0 ? x : cmmdc(y, x % y); }
public:
    Fractie(int x = 0, int y = 1)                   // valori implicite: Fractie() este 0/1
    {
        if (y < 0) { x = -x; y = -y; }              // semnul stă la numărător
        int g = cmmdc(x < 0 ? -x : x, y);
        if (g == 0) g = 1;
        a = x / g;
        b = y / g;
    }
    int getA() const { return a; }
    int getB() const { return b; }
    Fractie operator+(const Fractie &f) const       // supraîncărcarea lui +
    {
        return Fractie(a * f.b + f.a * b, b * f.b);
    }
    bool operator<(const Fractie &f) const          // supraîncărcarea lui <
    {
        return a * f.b < f.a * b;                   // b > 0, deci sensul se păstrează
    }
};
```

DE CE AȘA?

De ce `static` la `cmmdc`? Nu are nevoie de `a` și `b` ale unui obiect anume; este o funcție auxiliară a clasei. **De ce `operator+`?** Ca să poți scrie `f + g` în loc de `f.aduna(g)`: codul din `main` se citește ca o formulă matematică. **De ce `const Fractie &f`?** Pasezi prin referință (fără copie) și promiți să nu modifici argumentul.

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

class Fractie
{
private:
    int a, b;
    static int cmmdc(int x, int y) { return y == 0 ? x : cmmdc(y, x % y); }
public:
    Fractie(int x = 0, int y = 1)
    {
        if (y < 0) { x = -x; y = -y; }
        int g = cmmdc(x < 0 ? -x : x, y);
        if (g == 0) g = 1;
        a = x / g;
        b = y / g;
    }
    int getA() const { return a; }
    int getB() const { return b; }
    Fractie operator+(const Fractie &f) const { return Fractie(a * f.b + f.a * b, b * f.b);
}
    bool operator<(const Fractie &f) const { return a * f.b < f.a * b; }
};

int main()
{
    int p, q, r, s;
    cin >> p >> q >> r >> s;
    Fractie f(p, q), g(r, s);
    Fractie suma = f + g;
    cout << suma.getA() << '/' << suma.getB() << '\n';
    cout << (f < g ? "mai mic" : "mai mare sau egal");
    return 0;
}
```

Urmărirea pentru `f + g` cu f = 1/2, g = 1/3:

Pas	Calcul
1	numărător: 1·3 + 1·2 = 5
2	numitor: 2·3 = 6
3	`Fractie(5, 6)` : cmmdc(5, 6) = 1 → rămâne 5/6

5. Constructori și destructori: viața unui obiect

Un obiect **se naște** (constructor) la declarare și **moare** (destructor) la ieșirea din blocul `{ }` în care a fost declarat. Destructorul are numele clasei cu `~` în față. El eliberează resursele (memorie alocată cu `new`, fișiere deschise).

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

class Cutie
{
    int id;
public:
    Cutie(int i) : id(i) { cout << "creat " << id << '\n'; }   // listă de inițializare
    ~Cutie()             { cout << "distrus " << id << '\n'; }
};

int main()
{
    Cutie a(1);
    {
        Cutie b(2);
    }                       // aici b iese din bloc: se distruge
    Cutie c(3);
    return 0;               // se distrug c, apoi a (ordinea inversă a creării)
}
```

DE CE AȘA?

De ce `: id(i)` (listă de inițializare)? Inițializează atributul direct, înainte de intrarea în corpul constructorului; este obligatoriu pentru atributele `const` și pentru cele care sunt obiecte fără constructor implicit. **De ce ordine inversă la distrugere?** Un obiect creat mai târziu poate depinde de cele create mai devreme (ca într-o stivă), deci se distruge primul.

6. Structură de date ca clasă: Stiva

Stiva din lecția XI.17 devine o clasă: datele (vectorul și vârful) sunt ascunse, iar utilizatorul vede doar `push`, `pop`, `top`, `empty`.



Stiva

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

class Stiva
{
private:
    int v[1001];
    int n;                                  // câte elemente sunt
public:
    Stiva() { n = 0; }
    bool empty() const { return n == 0; }
    void push(int x) { if (n < 1000) v[n++] = x; }
    int top() const { return v[n - 1]; }    // apelat doar dacă !empty()
    void pop() { if (n > 0) n--; }
};

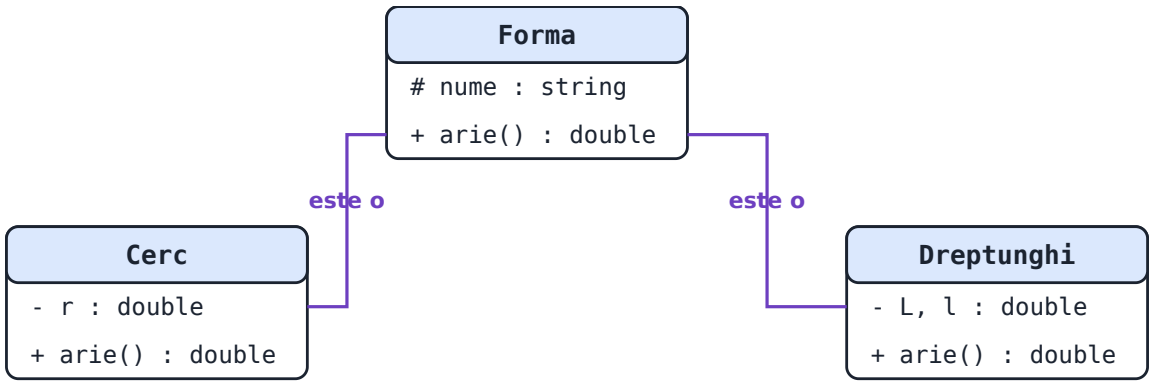
int main()
{
    int n, x;
    Stiva s;
    cin >> n;
    for (int i = 0; i < n; i++) { cin >> x; s.push(x); }
    while (!s.empty()) { cout << s.top() << ' '; s.pop(); }
    return 0;
}
```

DE CE AȘA?

Ce câștigi față de vectorul global? Dacă mâine schimbi stiva să folosească o listă înlănțuită (lecția XI.18), `main` rămâne neschimbat: utilizatorul clasei nu știe (și nu trebuie să știe) cum este făcută. Aceasta este esența încapsulării.

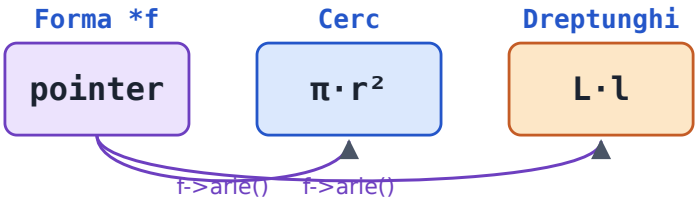
7. Moștenire și polimorfism

Cercul și dreptunghiul sunt „feluri de" formă: au un nume și o arie, dar aria se calculează diferit. Clasa derivată **moștenește** de la cea de bază și **redefinește** ce diferă.



clasele derivate preiau datele și metodele clasei de bază și le pot adăuga sau schimba

Moștenire



cu virtual se alege la rulare metoda clasei obiectului, nu a pointerului

Polimorfism

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

class Forma
{
public:
    virtual double arie() const = 0;        // metodă virtuală pură: fiecare formă o
definește
    virtual ~Forma() {}                     // destructor virtual (obligatoriu cu
moștenire)
};

class Cerc : public Forma
{
    double r;
public:
    Cerc(double raza) : r(raza) {}
    double arie() const override { return 3.14159265 * r * r; }
};

class Dreptunghi : public Forma
{
    double L, l;
public:
    Dreptunghi(double a, double b) : L(a), l(b) {}
    double arie() const override { return L * l; }
};

int main()
{
    int n;
    cin >> n;
    vector<Forma*> forme;
    for (int i = 0; i < n; i++)
    {
        char tip;
        cin >> tip;
        if (tip == 'C') { double r; cin >> r; forme.push_back(new Cerc(r)); }
        else            { double a, b; cin >> a >> b; forme.push_back(new Dreptunghi(a,
b)); }
    }
    cout << fixed << setprecision(2);
    for (Forma *f : forme) cout << f->arie() << '\n';     // același apel, comportament
diferit
    for (Forma *f : forme) delete f;
    return 0;
}
```

DE CE AȘA?

De ce `virtual`? Fără `virtual`, `f->arie()` ar apela metoda clasei **Forma** (tipul pointerului). Cu `virtual`, C++ alege la rulare metoda clasei **obiectului real** (Cerc sau Dreptunghi): asta este polimorfismul. **De ce `override`?** Compilatorul verifică că chiar redefinești o metodă virtuală (o greșeală de scriere în nume nu mai trece neobservată). **De ce `= 0`?** Clasa `Forma` este abstractă: nu poți crea o „formă în general", doar cercuri, dreptunghiuri etc. **De ce destructor virtual?** Ca `delete f` (cu `f` de tip `Forma*`) să distrugă corect și partea derivată.

8. Greșeli frecvente

Greșeala	Efect	Corect
uiți `;` după `}` la sfârșitul clasei	eroare de compilare	`class A { ... };`
accesezi un membru `private` din `main`	eroare de compilare	metodă publică
`Fractie f();` pentru obiect fără parametri	declară o funcție, nu un obiect	`Fractie f;`
`set` care nu verifică nimic	nu ai încapsulat nimic	verifici domeniul valorilor
metodă care doar citește, fără `const`	nu merge pe obiecte constante	`int get() const`
uiți `virtual` în clasa de bază	se apelează metoda de bază	`virtual` + `override`
uiți `delete` după `new`	scurgere de memorie	destructor sau `delete`
inițializezi în constructor atribute `const` cu `=`	eroare	listă de inițializare `: c(valoare)`

9. La BAC și în programare în general

LA BAC

Pentru BAC (Informatică, C++), POO nu face parte din programa de bază. Nu apare la Subiectele I–III. Învățând-o acum, înțelegi `string`, `vector`, `stack`, `queue` din STL (sunt clase!), și te pregătești pentru facultate, olimpiadă (unele probleme se scriu elegant cu clase) și pentru orice limbaj modern (Java, C#, Python).

10. Exersează pe hârtie

1. Scrie clasa `Cerc` cu raza privată, constructor și metodele `arie()` și `circumferinta()` (folosește 3.14).

2. Ce afișează programul?

```
class A { int x; public: A(int v) : x(v) { cout << "A" << x << ' '; } ~A() { cout << "~A"
<< x << ' '; } };
int main() { A a(1); { A b(2); } A c(3); }
```

3. De ce nu compilează `d.L = 5;` în `main`, dacă `L` este `private` în `Dreptunghi`? Cum rezolvi (corect) situația dacă ai nevoie să schimbi lungimea?

4. Adaugă la `Stiva` o metodă `size()` care returnează numărul de elemente și o metodă `clear()`.

5. Scrie clasa `Elev` cu nume și un vector de maxim 20 de note; metoda `adaugaNota(n)` acceptă doar note între 1 și 10 și returnează `true` / `false`; metoda `media()` returnează media notelor.

6. Scrie operatorul `-` pentru `Fractie` (diferența a două fracții).

7. Pentru clasa `Forma` din lecție, scrie clasa `Patrat` derivată din `Forma`, cu latura privată și aria `l * l`.

8. Ce diferență este între `struct` și `class` în C++?

9. (Grilă) Ce afișează `Fractie f(2, 4), g(3, 6); cout << (f < g);`? - a) 0 b) 1 c) 2/4 d) eroare de compilare

10. (Grilă) Care afirmație despre `virtual` este adevărată? - a) face metoda privată - b) permite alegerea metodei după tipul real al obiectului, la rulare - c) face clasa să nu mai poată fi derivată - d) este necesar pentru orice metodă

11. Mai departe (pentru cine vrea mai mult)

Supraîncărcarea lui `<<`. Ca să scrii `cout << f;` în loc de `cout << f.getA() << '/' << f.getB();`:

```
ostream& operator<<(ostream &o, const Fractie &f) { return o << f.getA() << '/' <<
f.getB(); }
```

Compunere. Un obiect conține alt obiect: `class Cerc { Punct centru; double r; };`. Se numește relația „are un” (spre deosebire de moștenire, „este un”). Preferă compunerea când nu ai o relație clară de tip „este un”.

Șabloane (`template`). O singură clasă `Stiva<T>` pentru `int`, `double`, `string`: așa sunt făcute `vector<T>`, `stack<T>`, `queue<T>` din STL.

Principiile SOLID. Reguli de proiectare pentru clase (o clasă = un rol, depinzi de abstracții, nu de implementări). Se vor învăța la facultate, dar cine le știe devreme scrie cod mai curat.

Pointeri inteligenți (`unique_ptr`, `shared_ptr`). Eliberează automat memoria alocată; în C++ modern aproape nu se mai scrie `delete` de mână.

Rezumat

- Clasă = date + metode; obiect = instanță; constructorul îl inițializează, destructorul îl curăță.
- `private` pentru date, `public` pentru interfață: **încapsulare** (obiectul își apără datele).
- Metodele care doar citesc se marchează `const`; operatorii se supraîncarcă pentru cod mai natural.

- **Moștenire** (`: public Baza`) = „este un”; **polimorfism** = `virtual` + `override` , metoda aleasă după tipul real al obiectului.

Probleme de lucru (pbinfo)

Începe cu acestea, în ordine: sunt alese să urmeze pașii din lecție.

- [ ] `#3529` **oop-1** — ușoară
  https://www.pbinfo.ro/probleme/3529/oop-1
- [ ] `#3530` **oop-2** — ușoară
  https://www.pbinfo.ro/probleme/3530/oop-2

Temă (pbinfo)

Trimite soluțiile pe pbinfo și notează punctajul obținut. Bifează ce ai terminat.

- [ ] `#3533` **oop-3** — medie
  Definește clasa Stiva.
  https://www.pbinfo.ro/probleme/3533/oop-3
- [ ] `#3539` **oop-numere-complexe** — medie
  https://www.pbinfo.ro/probleme/3539/oop-numere-complexe

Citește și exersează în plus

- Clase și obiecte C++ https://www.pbinfo.ro/articole/3900/clase-si-obiecte-cpp
- Metode SET și GET https://www.pbinfo.ro/articole/25891/metodele-get-si-set
- Derivarea claselor în C++ https://www.pbinfo.ro/articole/25874/derivarea-claselor-in-cpp
- Polimorfism în C++ https://www.pbinfo.ro/articole/25901/polimorfism-in-cpp
- learncpp.com — Classes and objects — capitolul despre clase, de la zero, cu exemple (engleză)
  https://www.learncpp.com/cpp-tutorial/classes-and-class-members/
- cppreference — Classes — referință oficială: constructori, moștenire, virtual (engleză)
  https://en.cppreference.com/w/cpp/language/classes
- Refactoring Guru — Design Patterns — cum se organizează clase în programe mari, cu desene (engleză) https://refactoring.guru/design-patterns

Răspunsuri

Exersează pe hârtie

1.

```
class Cerc { double r; public: Cerc(double raza) { r = raza; } double arie() { return 3.14 * r * r; } double circumferinta() { return 2 * 3.14 * r; } };
```

2. Se afișează: **A1 A2 ~A2 A3 ~A3 ~A1**. `b` se distruge la ieșirea din bloc, iar `c` și `a` la sfârșitul lui `main`, în ordine inversă a creării.

3. Membrii `private` nu sunt accesibili din afara clasei. Soluție: o metodă publică `void setL(int nou) { if (nou > 0) L = nou; }` care verifică valoarea.

4. `int size() const { return n; }` și `void clear() { n = 0; }`.

5. Atribute: `string nume; int note[20]; int nr;`. `bool adaugaNota(int x) { if (x < 1 || x > 10 || nr == 20) return false; note[nr++] = x; return true; }`. `double media() const { if (nr == 0) return 0; double s = 0; for (int i = 0; i < nr; i++) s += note[i]; return s / nr; }`.

6. `Fractie operator-(const Fractie &f) const { return Fractie(a * f.b - f.a * b, b * f.b); }`

7. `class Patrat : public Forma { double l; public: Patrat(double x) : l(x) {} double arie() const override { return l * l; } };`

8. Doar accesul implicit: în `struct` membrii sunt `public`, în `class` sunt `private`. Altfel sunt identice.

9. `f` = 2/4 = 1/2 și `g` = 3/6 = 1/2 (după simplificare), deci `f < g` este fals și se afișează 0. Răspuns **a**.

10. Răspuns **b**: `virtual` permite alegerea metodei după tipul real al obiectului, la rulare.