AVANSAT · POO ÎN C++ · LECȚIA 1

Clase de la zero: constructori, liste de inițializare, destructor, RAII, this, const, static

Durată: 80 min Nivel: avansat / pregătire pentru facultate și industrie Înainte: lecțiile xi-21, xi-18

CE VEI ȘTI DUPĂ LECȚIE

- Scrii constructori corecți cu liste de inițializare, constructori delegați, = default și = delete.
- Explici ordinea de construire și de distrugere a obiectelor și folosești destructorul pentru RAII.
- Folosești pointerul this și întorci *this pentru apeluri înlănțuite.
- Marchezi corect metodele și parametrii const (const-correctness).
- Folosești membri static pentru date și funcții ale clasei, nu ale unui obiect.

1. De unde pornim

La clasa a XI-a (lecția XI.21) ai scris clase cu date private, metode publice, un constructor și un destructor. Aici mergem mai adânc: **cum** se naște un obiect, **în ce ordine** mor obiectele, cum un obiect își gestionează singur resursele (**RAII**) și cum spui compilatorului ce **nu** are voie să se schimbe (`const`). Sunt lucrurile pe care un programator C++ le folosește zilnic și pe care se întreabă la orice interviu.

DE ȘTIUT

`struct` și `class` sunt **aproape identice** în C++. Singura diferență: membrii unui `struct` sunt implicit `public`, ai unui `class` implicit `private`. Convenția: `struct` pentru date simple, fără reguli (un punct, o muchie); `class` pentru obiecte care își apără datele.

2. Constructorul și lista de inițializare

Constructorul are rolul să aducă obiectul într-o **stare validă** imediat ce există. Există două feluri de a da valori membrilor:

```
class Elev
{
    string nume;
    double nota;
public:
    // varianta 1: atribuire în corpul constructorului
    Elev(string n, double x) { nume = n; nota = x; }
};
```

```
class Elev
{
    string nume;
    double nota;
public:
    // varianta 2: lista de inițializare (recomandată)
    Elev(string n, double x) : nume(n), nota(x) {}
};
```

DE CE AȘA?

De ce lista de inițializare? În varianta 1, `nume` este întâi **construit gol**, apoi **i se atribuie** valoarea: două operații. În varianta 2 este construit direct cu valoarea dorită: una. Mai important, unii membri **nu pot fi atribuiți**, ci doar inițializați: membrii `const`, referințele (`int&`) și membrii care nu au constructor fără parametri. Pentru ei, lista de inițializare este **singura** cale.

Ordinea de inițializare

Membrii se inițializează **în ordinea în care sunt declarați în clasă**, nu în ordinea din listă:

```
class Interval
{
    int lungime;                          // declarat primul: inițializat primul
    int st, dr;
public:
    Interval(int a, int b) : st(a), dr(b), lungime(dr - st) {}   // GREȘIT: dr și st nu au
încă valori
};
```

`lungime` se calculează **înaintea** lui `st` și `dr`, din valori neinițializate. Regula: scrie lista în ordinea declarării (compilatorul te avertizează cu `-Wall`).

Valori implicite, constructori delegați, `= default`, `= delete`, `explicit`

Unealta	Exemplu	La ce folosește
valoare implicită la membru	`int nota = 10;`	valoare folosită dacă constructorul nu spune altceva
constructor delegat	`Elev() : Elev("anonim", 10) {}`	un constructor îl cheamă pe altul: fără cod duplicat
`= default`	`Elev() = default;`	cere compilatorului versiunea generată automat
`= delete`	`Elev(const Elev&) = delete;`	interzice operația (aici, copierea)
`explicit`	`explicit Fractie(int a);`	interzice conversiile implicite surprinzătoare

ATENȚIE — GREȘELI FRECVENTE

Fără `explicit`, un constructor cu un singur parametru devine o **conversie automată**: dacă o funcție cere un `Fractie` și primește `5`, compilatorul construiește în tăcere `Fractie(5)`. Uneori e util, de cele mai multe ori ascunde greșeli. Regula: constructorii cu un parametru se marchează `explicit`, cu excepția cazurilor în care conversia este chiar dorită.

3. Viața unui obiect: ordinea construirii și distrugerii

Un obiect local (pe **stivă**) se construiește când execuția ajunge la declarația lui și se distruge **automat** la ieșirea din blocul `{ … }` în care a fost declarat. Obiectele dintr-un bloc se distrug în **ordine inversă** construirii, exact ca într-o stivă.

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

class Martor
{
    string nume;
public:
    Martor(string n) : nume(n) { cout << "construit " << nume << '\n'; }
    ~Martor() { cout << "distrus " << nume << '\n'; }
};

int main()
{
    Martor a("A");
    {
        Martor b("B");
        Martor c("C");
    }                                   // aici se închide blocul: mor C, apoi B
    Martor d("D");
    return 0;                           // la ieșirea din main mor D, apoi A
}
```

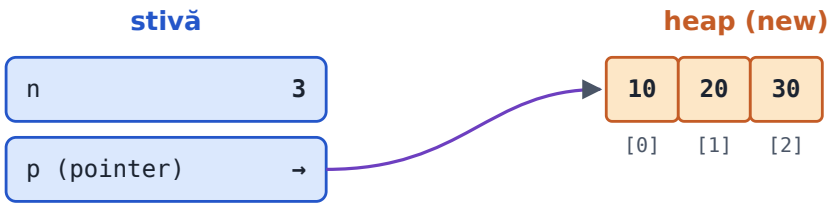
Moment	Pe stivă (de jos în sus)	Afișare
`Martor a`	A	construit A
`Martor b`, `Martor c`	A B C	construit B, construit C
`}`	A	distrus C, distrus B
`Martor d`	A D	construit D
sfârșitul lui `main`	–	distrus D, distrus A

DE CE AȘA?

De ce ordinea inversă? Un obiect construit mai târziu se poate folosi de unul construit mai devreme (de exemplu, `c` poate ține o referință la `b`). Dacă `b` ar muri primul, `c` ar rămâne cu o referință spre ceva mort. Distrugerea în ordine inversă garantează că fiecare obiect moare **înaintea** celor de care depinde. Aceeași regulă se aplică membrilor unui obiect: se construiesc în ordinea declarării și se distrug în ordine inversă.

4. RAII: resursa trăiește cât trăiește obiectul

RAII (*Resource Acquisition Is Initialization*) este cea mai importantă idee din C++: o **resursă** (memorie dinamică, un fișier deschis, o conexiune, un lacăt) este **luată în constructor** și **eliberată în destructor**. Pentru că destructorul este apelat automat la ieșirea din bloc, **orice** cale de ieșire (un `return` timpuriu, o excepție) eliberează resursa. Nu mai există „am uitat `delete`" sau „am uitat `close`".



constructorul face new int[n]; destructorul face delete[] p

Memoria unui obiect Tablou

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

class Tablou
{
    int n;
    int *p;                                 // memoria propriu-zisă este în heap
public:
    explicit Tablou(int dim) : n(dim), p(new int[dim]) {}   // ia resursa
    ~Tablou() { delete[] p; }                                // o eliberează, mereu

    Tablou(const Tablou&) = delete;         // copierea: în lecția oop-02
    Tablou& operator=(const Tablou&) = delete;

    int marime() const { return n; }
    int& operator[](int i) { return p[i]; }                  // acces la element (oop-03)
};

long long suma(Tablou &t)
{
    long long s = 0;
    for (int i = 0; i < t.marime(); i++) s += t[i];
    return s;
}

int main()
{
    int n;
    cin >> n;
    Tablou t(n);                            // new int[n] se face în constructor
    for (int i = 0; i < n; i++) cin >> t[i];
    cout << suma(t);
    return 0;                               // delete[] se face singur, în destructor
}
```

DE CE AȘA?

De ce `= delete` la copiere? Dacă lași compilatorul să copieze un `Tablou`, el copiază **pointerul**, nu elementele: două obiecte ar arăta spre aceeași memorie, iar la sfârșit ambele ar face `delete[]` pe ea (eroare gravă: *double free*). Până învățăm să scriem copierea corect (oop-02), o interzicem: codul care ar copia nici nu se mai compilează. **De ce `explicit`?** Ca `Tablou t = 5;` să nu fie acceptat din greșeală.

IMPORTANT

În C++ modern, aproape niciodată nu scrii `new` și `delete` direct în codul obișnuit: folosești clase RAII gata făcute: `vector`, `string`, `unique_ptr`, `ifstream`, `lock_guard`. Clasa `Tablou` de mai sus este doar ca să vezi **cum** funcționează ele pe dinăuntru.

5. Pointerul `this` și apelurile înlănțuite

În orice metodă, `this` este un pointer către **obiectul pentru care s-a apelat metoda**. Îl folosești când un parametru are același nume ca un membru (`this->nume = nume;`) și când vrei să întorci obiectul

însuși (`return *this;`), ca apelurile să se poată înlănțui.

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

class Text
{
    string s;
public:
    Text& adauga(const string &x)          // întoarce o referință la obiectul curent
    {
        s += x;
        return *this;
    }
    Text& majuscule()
    {
        for (char &c : s)
            if (c >= 'a' && c <= 'z') c = c - 'a' + 'A';
        return *this;
    }
    const string& valoare() const { return s; }   // nu modifică obiectul
    int lungime() const { return s.size(); }
};

int main()
{
    string a, b;
    cin >> a >> b;
    Text t;
    t.adauga(a).adauga(b).majuscule();     // înlănțuire: fiecare apel întoarce t
    cout << t.valoare() << " (" << t.lungime() << ")";
    return 0;
}
```

DE CE AȘA?

De ce `Text&` și nu `Text` ? Cu `Text&` se întoarce **același** obiect; al doilea `adauga` lucrează tot pe `t` . Cu `Text` (fără `&`) s-ar întoarce o **copie**, iar `majuscule()` ar modifica acea copie temporară, nu pe `t` . Așa sunt construite `cout << a << b` (lecția oop-03) și clasele de tip *builder* (lecția oop-08).

6. Const-correctness

`const` este o promisiune verificată de compilator: „aici nu se modifică nimic”. Folosit consecvent, prinde greșeli și documentează codul.

Unde	Exemplu	Înseamnă
variabilă	`const int N = 100;`	nu se mai poate schimba
parametru prin referință	`void f(const string &s)`	f nu copiază s și nici nu îl modifică

Unde	Exemplu	Înseamnă
metodă	`int marime() const`	metoda nu modifică obiectul; se poate apela pe obiecte `const`
valoare întoarsă	`const string& valoare() const`	cine primește referința nu poate modifica interiorul
pointer	`const int *p` / `int * const p`	nu modifici valoarea / nu muți pointerul

ATENȚIE — GREȘELI FRECVENTE

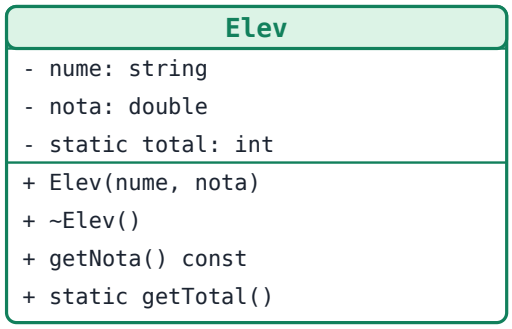
Dacă uiți `const` la o metodă care nu modifică nimic, ea **nu mai poate fi apelată** pe un obiect `const` sau prin referință `const`: `void afis(const Text &t) { t.lungime(); }` nu se compilează dacă `lungime()` nu este marcată `const`. Regula practică: **orice metodă care doar citește se marchează `const`**, iar obiectele mari se transmit prin `const &`.

SFAT

Cuvântul `mutable` permite modificarea unui membru chiar într-o metodă `const`. Se folosește rar, pentru detalii interne care nu schimbă „valoarea logică" a obiectului (de exemplu, un rezultat memorat ca să nu fie recalculat).

7. Membri `static`: ai clasei, nu ai obiectului

Un membru **static** există o singură dată pentru **întreaga clasă**, nu câte unul în fiecare obiect. O metodă statică nu are `this` și se apelează cu numele clasei: `Elev::total()`.



– privat · + public · static = al clasei, nu al unui obiect

Clasa Elev

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

class Elev
{
    string nume_;
    double nota_;
    inline static int vii = 0;            // C++17: membru static inițializat chiar în clasă
public:
    Elev(string n, double x) : nume_(n), nota_(x) { vii++; }
    ~Elev() { vii--; }
    Elev(const Elev&) = delete;           // altfel o copie nu ar crește vii, dar l-ar scădea la distrugere

    const string& nume() const { return nume_; }
    double nota() const { return nota_; }
    static int total() { return vii; }    // fără this: nu are acces la nume_ sau nota_
};

int main()
{
    cout << Elev::total() << '\n';
    {
        Elev a("Ana", 9.5), b("Dan", 8);
        cout << Elev::total() << '\n';
        cout << a.nume() << ' ' << a.nota() << '\n';
    }
    cout << Elev::total() << '\n';        // a și b au murit
    return 0;
}
```

DE ȘTIUT

Înainte de C++17, membrul static se declara în clasă (`static int vii;`) și se **definea** o dată în afara ei, într-un fișier `.cpp` : `int Elev::vii = 0;` . Cu `inline static` (C++17) declarația și definiția stau împreună.

8. Greșeli frecvente

Greșeala	Ce se întâmplă	Corect
lista de inițializare în altă ordine decât declararea	membri calculați din valori neinițializate	aceeași ordine ca în clasă
membru `const` sau referință atribuit în corp	eroare de compilare	lista de inițializare
constructor cu un parametru fără `explicit`	conversii neașteptate	`explicit`
`new` în constructor, fără `delete` în destructor	memorie pierdută (leak)	RAII: eliberează în destructor

Greșeala	Ce se întâmplă	Corect
clasă cu pointer copiată implicit	două obiecte, aceeași memorie, *double free*	`= delete` la copiere sau regula lui 3/5 (oop-02)
metodă care doar citește, fără `const`	nu se poate apela pe obiecte `const`	marchează-o `const`
întorci `Text` în loc de `Text&` la înlănțuire	se modifică o copie	`Text&` și `return *this`
`return *this` dintr-o metodă care întoarce referință la un obiect local	referință spre un obiect mort	întoarce doar membri sau `*this`
metodă `static` care folosește un membru obișnuit	eroare: nu există `this`	membrul static sau metodă obișnuită

9. În practică și la interviu

Întrebările clasice: „ce este RAII?", „de ce lista de inițializare?", „în ce ordine se distrug obiectele?", „ce face `explicit`?", „ce înseamnă o metodă `const`?". Răspunsurile scurte se găsesc în **Rezumat**. În codul real, aceste reguli apar în aproape fiecare clasă: obiecte care își apără invariantul (o stare mereu validă), care nu pierd resurse și care sunt ușor de folosit corect și greu de folosit greșit.

10. Exersează pe hârtie

1. Ce afișează programul de mai jos?

```
#include <iostream>
using namespace std;
struct M { int x; M(int v) : x(v) { cout << "+" << x; } ~M() { cout << "-" << x; } };
int main() { M a(1); { M b(2); M c(3); } M d(4); }
```

2. Clasa `Cerc` are membrul `const double raza;`. De ce nu merge `Cerc(double r) { raza = r; }`? Scrie constructorul corect.

3. Cu `class P { int a; int b; public: P(int x) : b(x), a(b + 1) {} };`, ce valoare are `a` după `P p(5)`? De ce?

4. Ce se întâmplă cu `Tablou t = 5;` dacă constructorul este `explicit`? Dar fără `explicit`?

5. De ce metoda `marime()` din `Tablou` este marcată `const`? Dă un exemplu de cod care nu s-ar compila fără `const`.

6. Ce afișează `t.adauga("x").adauga("y").valoare()` dacă `adauga` întoarce `Text` (fără `&`), pentru un `t` gol? Ce rămâne în `t`?

7. Pentru clasa `Elev` din §7, ce s-ar întâmpla cu `total()` dacă ai șterge linia `Elev(const Elev&) = delete;` și ai scrie `Elev c = a;`?

8. Scrie o clasă RAII `Jurnal` care afișează „început" în constructor și „sfârșit" în destructor. Ce afișează o funcție cu `Jurnal j;` și un `return` la mijloc?

9. O metodă statică poate fi apelată fără niciun obiect creat? Justifică.

10. Ce înseamnă `const int *p` și ce înseamnă `int * const p`?

11. Mini-grile

1. Obiectele locale dintr-un bloc se distrug: a) în ordinea construirii b) în ordine aleatoare c) în ordine inversă construirii d) doar la sfârșitul programului

2. Un membru `const` al unei clase se inițializează: a) în corpul constructorului b) în lista de inițializare (sau cu valoare implicită la declarare) c) cu o metodă set d) nu se poate

3. RAII înseamnă că o resursă este: a) luată în constructor și eliberată în destructor b) alocată global c) eliberată manual de utilizator d) alocată doar pe stivă

4. O metodă `static`: a) are acces la `this` b) nu poate întoarce valori c) este automat `const` d) nu are `this` și se apelează cu numele clasei

12. Mai departe (pentru cine vrea mai mult)

- **Invariantul clasei:** condiția care trebuie să fie mereu adevărată (de exemplu, numitorul unei fracții ≠ 0); constructorul o stabilește, metodele o păstrează.
- **Agregate și inițializare cu acolade** (`Punct p{1, 2};`), *designated initializers* în C++20.
- `constexpr` constructori: obiecte calculate la compilare.
- `friend` (lecția oop-03) și **Pimpl** (ascunderea detaliilor într-o clasă internă).
- **Regula lui 0/3/5:** lecția următoare.

Rezumat

- Lista de inițializare construiește direct membrii; obligatorie pentru `const`, referințe; ordinea reală este ordinea declarării.
- `= default`, `= delete`, constructori delegați, `explicit` pentru constructori cu un parametru.
- Obiectele locale se distrug automat, în ordine inversă construirii.
- **RAII:** resursa se ia în constructor și se eliberează în destructor; în practică folosești `vector`, `string`, `unique_ptr`.
- `this` arată spre obiectul curent; `return *this` cu tip `Clasa&` permite înlănțuirea.
- Metodele care citesc sunt `const`; obiectele mari se transmit prin `const &`.
- Membrii `static` aparțin clasei; metodele statice nu au `this`.

Citește și exersează în plus

- learncpp.com — Constructor member initializer lists — listele de inițializare, pas cu pas (engleză) https://www.learncpp.com/cpp-tutorial/constructor-member-initializer-lists/
- cppreference — RAII — definiția și exemplele standard (engleză) https://en.cppreference.com/w/cpp/language/raii
- C++ Core Guidelines — Classes and class hierarchies — regulile recomandate de autorii limbajului (engleză) https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#S-class

Răspunsuri

Exersează pe hârtie

1. `+1+2+3-3-2+4-4-1`.

2. Un membru `const` nu poate fi **atribuit**, doar **inițializat**; în corpul constructorului membrul există deja. Corect: `Cerc(double r) : raza(r) {}`.

3. `a` este declarat înaintea lui `b`, deci se inițializează primul, din `b + 1`, când `b` nu are încă valoare: `a` are o valoare **nedeterminată** (gunoi). Abia apoi `b = 5`.

4. Cu `explicit`: eroare de compilare (conversia implicită din `int` este interzisă); trebuie `Tablou t(5);`. Fără `explicit`: se compilează și construiește un tablou de 5 elemente.

5. Pentru că doar citește. Fără `const`, funcția `void f(const Tablou &t) { cout << t.marime(); }` nu s-ar compila: pe o referință `const` se pot apela numai metode `const`.

6. Afișează `xy`, dar `t` rămâne `"x"`: primul `adauga` modifică `t` și întoarce o copie; al doilea modifică acea copie temporară.

7. Copierea generată automat nu trece prin constructorul care face `vii++`, dar la distrugerea copiei se execută `vii--`: `total()` ar ajunge mai mic decât numărul real de obiecte (chiar negativ).

8. `struct Jurnal { Jurnal() { cout << "început\n"; } ~Jurnal() { cout << "sfârșit\n"; } };` O funcție cu `Jurnal j;` afișează „început", apoi, la orice `return` (inclusiv cel de la mijloc), „sfârșit": destructorul rulează la ieșirea din funcție, pe orice cale.

9. Da: metoda statică aparține clasei și nu are nevoie de `this`: `Elev::total()` funcționează și când nu există niciun `Elev`.

10. `const int *p`: p arată spre un `int` care **nu se poate modifica prin p** (p se poate muta). `int * const p`: p **nu se poate muta**, dar valoarea spre care arată se poate modifica.

Mini-grile

1. c) în ordine inversă construirii. **2.** b) în lista de inițializare. **3.** a). **4.** d).