AVANSAT · POO ÎN C++ · LECȚIA 2

Copiere și mutare: constructorul de copiere, operatorul =, semantica mutării, regula lui 0/3/5

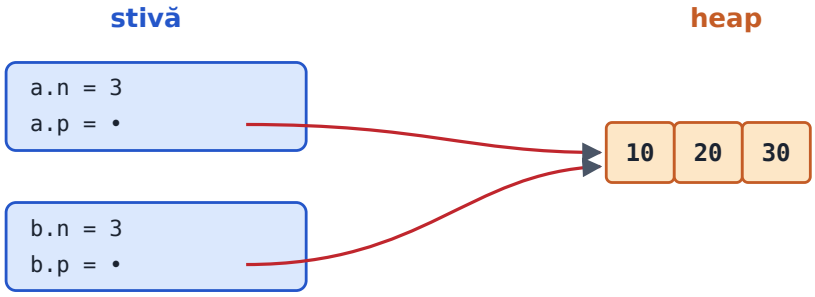
Durată: 90 min Nivel: avansat / pregătire pentru facultate și industrie Înainte: lecțiile oop-01

CE VEI ȘTI DUPĂ LECȚIE

- Explici diferența dintre copierea superficială și cea profundă și de ce prima strică o clasă cu pointeri.
- Scrii constructorul de copiere și operatorul de atribuire prin copiere, cu tratarea auto-atribuirii.
- Deosebești lvalue de rvalue, folosești referințe rvalue (&&) și std::move.
- Scrii constructorul și atribuirea prin mutare, marcate noexcept.
- Aplici regula lui 3, regula lui 5 și, mai ales, regula lui 0.

1. Problema: copierea implicită copiază pointerul

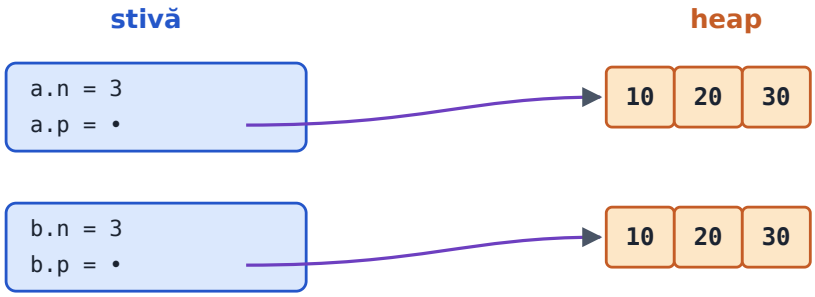
În lecția trecută am interzis copierea clasei `Tablou` cu `= delete`. Să vedem de ce. Dacă nu scrii nimic, compilatorul generează un constructor de copiere care copiază **fiecare membru**: `n` și `p`. Dar `p` este doar o **adresă**: copia arată spre **aceeași** memorie.



b = a copiază pointerul: ambele obiecte arată spre aceeași memorie → delete[] de două ori

Copiere superficială

Consecințe: (1) modificarea unui element prin `b` se vede și în `a`; (2) la sfârșit, ambele destructoare fac `delete[]` pe aceeași adresă: **double free**, comportament nedefinit (de obicei programul cade). Aceasta se numește **copiere superficială** (*shallow copy*). Avem nevoie de **copiere profundă** (*deep copy*): copia primește propria memorie și propriile valori.



b primește memoria lui proprie și o copie a elementelor

Copiere profundă

2. Funcțiile membre speciale

Compilatorul poate genera automat șase funcții. Pe ultimele cinci trebuie să le înțelegi pentru orice clasă care deține o resursă:

Funcția	Semnătura pentru clasa `T`	Când se apelează
constructor implicit	`T()`	`T x;`
destructor	`~T()`	la sfârșitul vieții obiectului
constructor de copiere	`T(const T&)`	`T b = a;`, `T b(a);`, transmitere prin valoare
atribuire prin copiere	`T& operator=(const T&)`	`b = a;` (b exista deja)
constructor de mutare	`T(T&&)`	`T b = std::move(a);`, `T b = funcție();`
atribuire prin mutare	`T& operator=(T&&)`	`b = std::move(a);`, `b = T(…);`

ATENȚIE — GREȘELI FRECVENTE

`T b = a;` **nu** este o atribuire, deși are semnul `=`: `b` se naște acum, deci se apelează **constructorul de copiere**. Atribuirea (`operator=`) are loc doar când obiectul din stânga exista deja.

3. Regula lui 3: copierea profundă

Regula lui 3: dacă o clasă are nevoie de un **destructor** scris de mână (pentru că eliberează o resursă), atunci aproape sigur are nevoie și de **constructorul de copiere** și de **atribuirea prin copiere** scrise de mână.

```
Tablou(const Tablou &alt) : n(alt.n), p(alt.n ? new int[alt.n] : nullptr)
{
    copy(alt.p, alt.p + n, p);             // memorie nouă + copie a valorilor
}

Tablou& operator=(const Tablou &alt)
{
    if (this == &alt) return *this;        // auto-atribuire: a = a
    int *nou = alt.n ? new int[alt.n] : nullptr;
    copy(alt.p, alt.p + alt.n, nou);       // întâi copia nouă...
    delete[] p;                            // ...apoi eliberăm memoria veche
    p = nou;
    n = alt.n;
    return *this;
}
```

DE CE AȘA?

De ce verificăm `this == &alt`? La `a = a`, dacă am face întâi `delete[] p`, am distruge chiar datele pe care vrem să le copiem. **De ce alocăm înainte de `delete[]`?** Dacă `new` eșuează (aruncă o excepție), obiectul rămâne neatins, în starea veche, validă: se numește **garanție puternică** la excepții. **De ce întoarcem `T&`?** Ca să meargă `a = b = c` și ca operatorul să se comporte ca pentru tipurile simple.

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

class Tablou
{
    int n;
    int *p;
public:
    explicit Tablou(int dim) : n(dim), p(new int[dim]()) {}
    ~Tablou() { delete[] p; }
    Tablou(const Tablou &alt) : n(alt.n), p(new int[alt.n])
    {
        copy(alt.p, alt.p + n, p);
    }
    Tablou& operator=(const Tablou &alt)
    {
        if (this == &alt) return *this;
        int *nou = new int[alt.n];
        copy(alt.p, alt.p + alt.n, nou);
        delete[] p;
        p = nou;
        n = alt.n;
        return *this;
    }
    int& operator[](int i) { return p[i]; }
    int marime() const { return n; }
    long long suma() const
    {
        long long s = 0;
        for (int i = 0; i < n; i++) s += p[i];
        return s;
    }
};

int main()
{
    int n;
    cin >> n;
    Tablou a(n);
    for (int i = 0; i < n; i++) cin >> a[i];
    Tablou b = a;                          // copiere profundă
    b[0] = 99;                             // modificăm doar copia
    cout << a[0] << ' ' << b[0] << '\n';
    cout << a.suma() << ' ' << b.suma() << '\n';
    return 0;                              // fiecare își eliberează memoria proprie
}
```

Idiomul copy-and-swap

O variantă scurtă și sigură a atribuirii: parametrul se primește **prin valoare** (deci se face deja o copie), apoi se **schimbă** conținutul cu el. Vechiul conținut pleacă odată cu parametrul, la ieșirea din funcție.

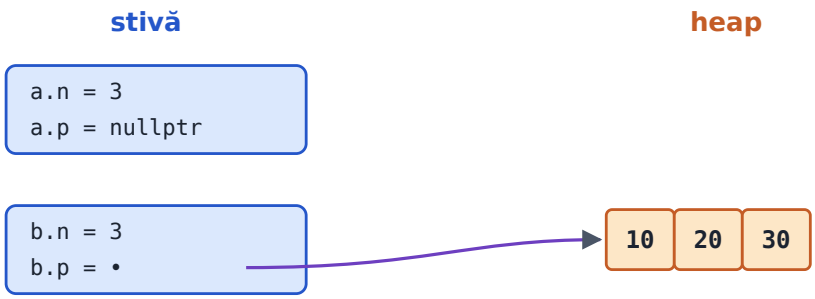
```
Tablou& operator=(Tablou alt)              // alt este deja o copie (sau un obiect mutat)
{
    swap(n, alt.n);
    swap(p, alt.p);
    return *this;                          // vechiul p este eliberat de destructorul lui
alt
}
```

Avantaje: auto-atribuirea funcționează fără test special, garanția puternică vine gratis, iar aceeași funcție servește și pentru copiere și pentru mutare.

4. lvalue, rvalue și referințe `&&`

Termen	Ce este	Exemple
lvalue	are un nume și o adresă; trăiește mai departe	`a`, `v[3]`, `*p`
rvalue	valoare temporară, fără nume, care moare la sfârșitul expresiei	`5`, `a + b`, `Tablou(3)`, `f()`
`T&`	referință la lvalue	se leagă de obiecte cu nume
`const T&`	referință constantă	se leagă de orice (și de temporare)
`T&&`	referință la **rvalue**	se leagă doar de temporare (sau de ce marchezi cu `std::move`)

Ideea mutării: dacă sursa este un temporar care oricum va muri, de ce să-i copiem elementele? Mai bine îi **luăm** memoria (copiem doar pointerul) și o lăsăm goală. Costul scade de la O(n) la O(1).



b preia pointerul lui a; a rămâne gol (nullptr), dar valid și distructibil

Mutare

`std::move(a)` **nu mută nimic** singur: doar transformă `a` într-un rvalue, adică spune „poți să-i iei resursele, nu mai am nevoie de el”. Mutarea propriu-zisă o face constructorul sau atribuirea prin mutare.

5. Regula lui 5: constructorul și atribuirea prin mutare

Programul de mai jos are toate cele cinci funcții speciale și afișează care dintre ele este apelată.

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

class Tablou
{
    int n;
    int *p;
public:
    explicit Tablou(int dim = 0) : n(dim), p(dim ? new int[dim]() : nullptr)
    {
        cout << "constructor(" << n << ")\n";
    }
    ~Tablou() { delete[] p; }

    Tablou(const Tablou &alt) : n(alt.n), p(alt.n ? new int[alt.n] : nullptr)     // 1. copiere
    {
        copy(alt.p, alt.p + n, p);
        cout << "copiere\n";
    }
    Tablou& operator=(const Tablou &alt)                                          // 2. atribuire prin copiere
    {
        cout << "atribuire prin copiere\n";
        if (this == &alt) return *this;
        int *nou = alt.n ? new int[alt.n] : nullptr;
        copy(alt.p, alt.p + alt.n, nou);
        delete[] p;
        p = nou;
        n = alt.n;
        return *this;
    }
    Tablou(Tablou &&alt) noexcept : n(alt.n), p(alt.p)                             // 3. mutare
    {
        alt.n = 0;
        alt.p = nullptr;
        cout << "mutare\n";
    }
    Tablou& operator=(Tablou &&alt) noexcept                                      // 4. atribuire prin mutare
    {
        cout << "atribuire prin mutare\n";
        if (this == &alt) return *this;
        delete[] p;
        p = alt.p;
        n = alt.n;
        alt.p = nullptr;
        alt.n = 0;
        return *this;
    }
    int marime() const { return n; }
};

Tablou creeaza(int k) { return Tablou(k); }
```

```
int main()
{
    Tablou a(3);
    Tablou b = a;
    Tablou c = std::move(a);
    b = c;
    c = Tablou(2);
    Tablou d = creeaza(5);
    cout << a.marime() << ' ' << b.marime() << ' ' << c.marime() << ' ' << d.marime() <<
'\n';
    return 0;
}
```

Linia	Ce se apelează	De ce
`Tablou a(3);`	constructor(3)	obiect nou
`Tablou b = a;`	copiere	`b` se naște dintr-un lvalue
`Tablou c = std::move(a);`	mutare	`std::move` face din `a` un rvalue; `a` rămâne gol
`b = c;`	atribuire prin copiere	`b` exista; `c` este lvalue
`c = Tablou(2);`	constructor(2), apoi atribuire prin mutare	sursa este un temporar
`Tablou d = creeaza(5);`	doar constructor(5)	**copy elision** (C++17): obiectul se construiește direct în `d`
afișarea	`0 3 2 5`	`a` a fost golit de mutare

DE CE AȘA?

De ce `noexcept` la mutare? Când un `vector` se mărește, își mută elementele în memoria nouă. Dacă mutarea ar putea arunca o excepție la mijloc, vectorul ar rămâne stricat; de aceea, `vector` **folosește mutarea doar dacă este marcată** `noexcept`, altfel copiază (mult mai lent). **De ce lăsăm sursa cu** `nullptr`**?** Destructorul ei va rula oricum; `delete[] nullptr` este sigur și nu face nimic. Obiectul mutat rămâne într-o stare **validă dar nespecificată**: îl poți distruge sau îi poți atribui o valoare nouă, dar nu te baza pe conținutul lui.

ATENȚIE — GREȘELI FRECVENTE

După `b = std::move(a);` nu mai folosi valoarea lui `a` (de exemplu `a[0]`) până nu îi dai alta. Și nu pune `std::move` la `return x;` dintr-o funcție: compilatorul face deja copy elision sau mutare automat, iar `std::move` poate chiar împiedica optimizarea.

6. Regula lui 0: cel mai bun cod este cel pe care nu îl scrii

Regula lui 0: dacă membrii clasei își gestionează singuri resursele (`vector`, `string`, `unique_ptr`, alte clase RAII), **nu scrie nicio funcție specială**: cele generate automat copiază, mută și distrug corect, membru cu membru.

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

class Elev                                  // regula lui 0: niciun destructor, nicio
copiere scrisă
{
public:
    string nume;
    vector<int> note;
};

int main()
{
    Elev a{"Ana", {9, 10, 8}};
    Elev b = a;                             // copiere profundă automată: string și vector
se copiază singure
    b.note.push_back(7);
    cout << a.nume << ' ' << a.note.size() << '\n';
    cout << b.nume << ' ' << b.note.size() << '\n';
    Elev c = std::move(a);                  // mutare automată: c preia memoria lui a
    cout << c.note.size() << ' ' << a.note.size() << '\n';
    return 0;
}
```

DE ȘTIUT

Ultimul număr afișat (`0`) este ce face în practică implementarea `vector` după mutare (rămâne gol). Standardul garantează doar că `a` este într-o stare validă, nu exact ce conține.

Rezumatul regulilor

Regula	Când	Ce scrii
regula lui 0	membrii sunt `vector` , `string` , `unique_ptr` etc.	**nimic**: generatele sunt corecte (preferata)
regula lui 3	clasa deține direct o resursă (C++ vechi)	destructor + copiere + atribuire prin copiere
regula lui 5	la fel, în C++ modern	cele 3 + mutare + atribuire prin mutare (`noexcept`)
clasă necopiabilă	resursa nu are sens copiată (fișier, conexiune)	`= delete` la copiere; eventual doar mutare (ca `unique_ptr`)

SFAT

Dacă declari **oricare** dintre destructor, copiere sau atribuire prin copiere, compilatorul **nu mai generează mutarea**: obiectele vor fi copiate acolo unde ar fi putut fi mutate. Încă un motiv pentru regula lui 0 sau pentru a scrie toate cinci (sau `= default` explicit).

7. Transmiterea parametrilor: ce alegi?

Situația	Semnătura
tip mic (`int` , `double` , `char`)	prin valoare: `void f(int x)`
obiect mare, doar citit	`void f(const T &x)`
obiect pe care funcția îl modifică pentru apelant	`void f(T &x)`
funcția păstrează o copie (de ex. constructor care reține un `string`)	prin valoare + `std::move` : `Elev(string n) : nume(std::move(n)) {}`
funcția preia proprietatea	`void f(unique_ptr<T> p)` (lecția oop-07)

8. Greșeli frecvente

Greșeala	Ce se întâmplă	Corect
clasă cu `new` în constructor și fără copiere scrisă	copiere superficială, *double free*	regula lui 3/5 sau `= delete` sau regula lui 0
atribuire fără test de auto-atribuire, cu `delete[]` întâi	`a = a` distruge datele	test `this == &alt` sau copy-and-swap
`delete[] p` înainte de `new` reușit	obiect stricat dacă `new` aruncă	alocă întâi, apoi eliberează
mutare fără a goli sursa	două obiecte dețin aceeași memorie	`alt.p = nullptr`
mutare fără `noexcept`	`vector` copiază în loc să mute	`noexcept`
folosești un obiect după `std::move`	valoare nespecificată	dă-i întâi o valoare nouă
`return std::move(x);`	poate bloca copy elision	`return x;`
confunzi `T b = a;` cu atribuirea	apelezi altă funcție decât crezi	este constructor de copiere

9. În practică și la interviu

„Ce este regula lui 3/5/0?", „ce face `std::move` ?", „de ce `noexcept` la mutare?", „diferența dintre copiere superficială și profundă?" sunt întrebări standard la interviurile de C++. În codul de producție, aproape toate clasele respectă **regula lui 0**; regula lui 5 apare doar în clasele care implementează ele însele o resursă (containere, wrapper-e peste API-uri C, buffer-e).

10. Exersează pe hârtie

1. Ce se apelează la fiecare linie? `T a; T b(a); T c = b; c = a; T d = T(); d = std::move(c);`

2. De ce clasa `Tablou` cu copierea implicită cade la sfârșitul lui `main` după `Tablou b = a;` ?

3. Scrie constructorul de copiere pentru clasa `class Sir { int n; char *s; };` (s are n + 1 caractere, ultimul `'\0'`).

4. Ce se întâmplă la `a = a;` cu atribuirea de mai jos? `T& operator=(const T &x) { delete[] p; n = x.n; p = new int[n]; copy(x.p, x.p + n, p); return *this; }`

5. Care sunt lvalue și care rvalue: `x`, `x + 1`, `v[2]`, `string("ab")`, `f()` (f întoarce `int`), `*p`?

6. Ce face `std::move(a)` singur, fără să fie atribuit nimănui?

7. Pentru `vector<Tablou> v;` și `v.push_back(Tablou(3));`, se apelează copierea sau mutarea? Dar `Tablou t(3); v.push_back(t);`?

8. Clasa `Elev` are membrii `string nume; vector<int> note;`. Ce funcții speciale trebuie să scrii? De ce?

9. De ce `unique_ptr` are mutare dar nu are copiere?

10. Ce garanție oferă idiomul copy-and-swap și de ce?

11. Mini-grile

1. `T b = a;` (b este nou) apelează: a) atribuirea prin copiere b) constructorul de copiere c) constructorul de mutare d) destructorul

2. Regula lui 0 spune că: a) nu folosești pointeri deloc b) dacă membrii își gestionează singuri resursele, nu scrii nicio funcție specială c) clasa nu are constructori d) nu copiezi niciodată obiecte

3. `std::move(x)`: a) mută imediat resursele lui x b) șterge x c) transformă x într-un rvalue, ca să poată fi mutat d) copiază x

4. Constructorul de mutare se marchează `noexcept` pentru că: a) altfel nu se compilează b) `vector` îl folosește la realocare doar dacă nu aruncă excepții c) e mai scurt d) așa cere regula lui 3

12. Mai departe (pentru cine vrea mai mult)

- **Perfect forwarding** (`T&&` într-un șablon și `std::forward`), folosit de `emplace_back`.
- **`emplace_back` vs `push_back`:** construirea direct în container.
- **Obiecte doar mutabile** (*move-only*): `unique_ptr`, `thread`, `ifstream`.
- **Small String Optimization:** de ce mutarea unui `string` scurt nu este mai rapidă decât copierea.
- **Garanțiile la excepții** (de bază, puternică, nothrow): lecția oop-07.

Rezumat

- Copierea implicită copiază membru cu membru: pentru pointeri înseamnă copiere superficială și *double free*.
- **Regula lui 3:** destructor ⇒ și copiere și atribuire prin copiere; **regula lui 5:** plus cele două mutări, `noexcept`.
- **Regula lui 0 (preferată):** folosește membri RAII și nu scrie nimic.
- `T b = a;` este construire, nu atribuire; atribuirea are test de auto-atribuire sau copy-and-swap.

- `std::move` doar marchează un obiect ca rvalue; mutarea fură resursa și lasă sursa goală dar validă.
- C++17 garantează copy elision la `return T(…)`: nicio copiere, nicio mutare.

Citește și exersează în plus

- cppreference — The rule of three/five/zero — regulile, cu exemple (engleză) https://en.cppreference.com/w/cpp/language/rule_of_three
- learncpp.com — Move constructors and move assignment — semantica mutării, explicată treptat (engleză) https://www.learncpp.com/cpp-tutorial/move-constructors-and-move-assignment/
- cppreference — Copy elision — când compilatorul nu mai face nicio copie (engleză) https://en.cppreference.com/w/cpp/language/copy_elision

Răspunsuri

Exersează pe hârtie

1. `T a;` constructor implicit; `T b(a);` copiere; `T c = b;` copiere; `c = a;` atribuire prin copiere; `T d = T();` constructor implicit (copy elision); `d = std::move(c);` atribuire prin mutare.

2. Copia are același pointer; la ieșirea din `main` se distruge întâi `b` (`delete[] p`), apoi `a` face `delete[]` pe aceeași adresă deja eliberată: double free.

3. `Sir(const Sir &x) : n(x.n), s(new char[x.n + 1]) { copy(x.s, x.s + n + 1, s); }` (se copiază și `'\0'`).

4. `delete[] p` șterge chiar datele lui `x` (este același obiect), apoi se copiază din memorie eliberată: comportament nedefinit. Trebuie test `this == &x` sau alocare înainte de ștergere.

5. lvalue: `x`, `v[2]`, `*p`. rvalue: `x + 1`, `string("ab")`, `f()`.

6. Nimic vizibil: doar convertește la rvalue; dacă rezultatul nu este folosit de o mutare, `a` rămâne neschimbat.

7. `v.push_back(Tablou(3))`: mutare (argumentul este temporar). `v.push_back(t)`: copiere (`t` este lvalue; pentru mutare ar trebui `std::move(t)`).

8. Niciuna (regula lui 0): `string` și `vector` se copiază, se mută și se distrug corect singure.

9. `unique_ptr` înseamnă proprietate **unică**: două copii ar însemna doi proprietari care fac `delete` de două ori. Mutarea transferă proprietatea, fără dublură.

10. Garanție puternică: copia se face în parametru, **înainte** de a modifica obiectul; dacă ea eșuează, obiectul curent este neatins. Swap-ul de după nu aruncă excepții.

Mini-grile

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