AVANSAT · POO ÎN C++ · LECȚIA 7

Excepții, smart pointers și lambda: C++ modern, sigur și expresiv

Durată: 100 min Nivel: avansat / pregătire pentru facultate și industrie Înainte: lecțiile oop-01, oop-02, oop-04

CE VEI ȘTI DUPĂ LECȚIE

- Arunci și prinzi excepții, folosești ierarhia standard și scrii excepții proprii.
- Explici derularea stivei și de ce RAII face codul sigur la excepții; cunoști garanțiile la excepții.
- Folosești unique_ptr, shared_ptr și weak_ptr și știi când se potrivește fiecare.
- Scrii lambda cu capturi prin valoare, prin referință și mutable, și le stochezi în std::function.
- Alegi între excepții, coduri de eroare și std::optional.

1. Trei unelte care schimbă felul în care scrii C++

Excepțiile separă codul „normal” de tratarea erorilor. **Smart pointers** fac ca memoria dinamică să se elibereze singură (RAII aplicat pointerilor). **Lambda** îți permit să scrii o funcție mică exact acolo unde ai nevoie de ea. Împreună, ele formează ceea ce se numește „C++ modern” (C++11 și mai nou).

2. Excepții: throw, try, catch

Când o funcție nu își poate îndeplini sarcina (împărțire la zero, fișier inexistent, index invalid), **aruncă** o excepție cu `throw`. Execuția sare imediat la cel mai apropiat bloc `catch` potrivit, oricât de adânc ar fi apelul.

Variantă de tratare a erorilor	Avantaj	Dezavantaj
cod de eroare întors (`-1`, `false`)	simplu, fără cost ascuns	ușor de ignorat; poluează fiecare apel cu `if`
excepție	nu poate fi ignorată; codul normal rămâne curat	cost când este aruncată; trebuie cod sigur la excepții
`std::optional<T>` (C++17)	„poate lipsi”, explicit în tip	nu spune *de ce* lipsește

Ierarhia standard

`std::exception` este baza; cele mai folosite: `logic_error` (greșeli de program: `invalid_argument`, `out_of_range`, `domain_error`) și `runtime_error` (situații detectabile doar la rulare: `overflow_error`, erori de sistem). Metoda `what()` întoarce mesajul.

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

class EroareMatematica : public runtime_error   // excepție proprie, derivată din cele standard
{
public:
    explicit EroareMatematica(const string &mesaj) : runtime_error(mesaj) {}
};

double imparte(double a, double b)
{
    if (b == 0) throw EroareMatematica("impartire la zero");
    return a / b;
}

double radical(double x)
{
    if (x < 0) throw EroareMatematica("radical din numar negativ");
    return sqrt(x);
}

int main()
{
    vector<int> v = {10, 20, 30};
    int n;
    cin >> n;
    for (int i = 0; i < n; i++)
    {
        string op;
        cin >> op;
        try
        {
            if (op == "div") { double a, b; cin >> a >> b; cout << imparte(a, b) << '\n'; }
            else if (op == "rad") { double x; cin >> x; cout << radical(x) << '\n'; }
            else if (op == "idx") { int k; cin >> k; cout << v.at(k) << '\n'; }   // at() verifică limitele
            else throw invalid_argument("comanda necunoscuta: " + op);
        }
        catch (const EroareMatematica &e) { cout << "eroare matematica: " << e.what() << '\n'; }
        catch (const out_of_range &) { cout << "eroare: index in afara vectorului\n"; }
        catch (const exception &e) { cout << "eroare: " << e.what() << '\n'; }   // orice altă excepție standard
    }
    return 0;
}
```

DE CE AȘA?

De ce prindem prin `const &`? Prinderea prin valoare copiază excepția și poate face **slicing** (o `EroareMatematica` prinsă ca `exception` prin valoare își pierde tipul). **De ce ordinea blocurilor `catch` contează?** Se alege **primul** bloc potrivit, de sus în jos; prinde întâi tipurile **specifice**, apoi pe cele generale. Dacă `catch (const exception&)` ar fi primul, ar prinde tot. **De ce `v.at(k)` și nu `v[k]`?** `at` verifică limitele și aruncă `out_of_range`; `[]` nu verifică (comportament nedefinit).

SFAT

`throw;` (fără argument) într-un `catch` **re-aruncă** excepția curentă mai departe, după ce ai făcut ceva local (de exemplu, ai scris în jurnal). `catch (...)` prinde orice, chiar și excepții care nu derivă din `std::exception`; folosește-l doar la cel mai înalt nivel.

3. Derularea stivei și RAII

Când o excepție părăsește o funcție, **toate obiectele locale** ale funcțiilor traversate sunt distruse, în ordine inversă: se numește **derularea stivei** (*stack unwinding*). De aceea RAII (lecția oop-01) este esențial: resursele ținute de obiecte se eliberează automat, chiar și când apare o eroare.

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

struct Martor
{
    string nume;
    explicit Martor(string n) : nume(n) { cout << "+" << nume << '\n'; }
    ~Martor() { cout << "-" << nume << '\n'; }
};

void nivel2(int x)
{
    Martor m("nivel2");
    if (x < 0) throw runtime_error("valoare negativa");
    cout << "nivel2 terminat\n";
}

void nivel1(int x)
{
    Martor m("nivel1");
    nivel2(x);
    cout << "nivel1 terminat\n";            // nu se mai ajunge aici dacă nivel2 aruncă
}

int main()
{
    int x;
    cin >> x;
    try
    {
        Martor m("main");
        nivel1(x);
    }
    catch (const exception &e)
    {
        cout << "prins: " << e.what() << '\n';
    }
    cout << "program continuat\n";
    return 0;
}
```

ATENȚIE — GREȘELI FRECVENTE

Nu arunca excepții din destructori. Dacă un destructor aruncă în timpul derulării stivei (când o altă excepție este deja „în zbor"), programul se oprește imediat (`std::terminate`). Destructorii sunt implicit `noexcept` .

Garanțiile la excepții

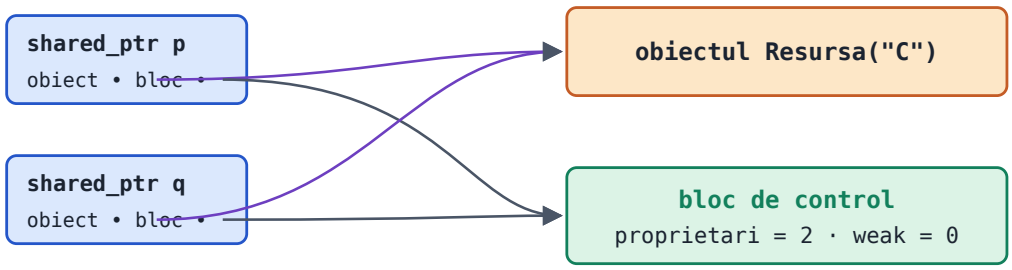
Garanția	Promite	Exemplu
nothrow	funcția nu aruncă niciodată	destructori, `swap` , mutarea marcată `noexcept`

Garanția	Promite	Exemplu
puternică	dacă aruncă, starea rămâne exact cea de dinainte („tot sau nimic”)	`vector::push_back`, copy-and-swap (oop-02)
de bază	dacă aruncă, nu se pierde nimic și obiectele rămân valide (dar starea se poate schimba)	majoritatea operațiilor
fără garanție	se pot pierde resurse sau strica invariantul	cod cu `new` / `delete` manual: de evitat

4. Smart pointers

Un **smart pointer** este un obiect care conține un pointer și face singur `delete` în destructor. Practic, în C++ modern nu mai scrii `new` / `delete` direct; folosești:

Tipul	Proprietate	Copiere	Folosire tipică
`unique_ptr<T>`	**un singur** proprietar	nu (doar mutare)	implicit: obiecte polimorfice, membri dinamici
`shared_ptr<T>`	**mai mulți** proprietari, numărați	da (crește numărătoarea)	obiecte partajate cu viață greu de prevăzut
`weak_ptr<T>`	**niciunul**: doar observă	da	rupe ciclurile; cache-uri; „părintele” unui nod
`T*` simplu	niciunul	da	doar observator, când viața obiectului e sigur mai lungă



la fiecare copie proprietari++, la fiecare distrugere proprietari--; la 0 se distruge obiectul

shared_ptr

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

struct Resursa
{
    string nume;
    explicit Resursa(string n) : nume(n) { cout << "creat " << nume << '\n'; }
    ~Resursa() { cout << "eliberat " << nume << '\n'; }
};

unique_ptr<Resursa> fabrica(const string &n) { return make_unique<Resursa>(n); }

struct Nod                                  // părinte și copil care se cunosc reciproc
{
    string nume;
    shared_ptr<Nod> copil;                  // părintele deține copilul
    weak_ptr<Nod> parinte;                  // copilul doar „cunoaște” părintele: nu îl ține în viață
    explicit Nod(string n) : nume(n) {}
    ~Nod() { cout << "distrus nod " << nume << '\n'; }
};

int main()
{
    cout << "-- unique_ptr --\n";
    {
        unique_ptr<Resursa> a = fabrica("A");
        unique_ptr<Resursa> b = std::move(a);          // proprietatea trece la b; a devine nullptr
        cout << (a ? "a are" : "a gol") << ", b are " << b->nume << '\n';
        vector<unique_ptr<Resursa>> v;
        v.push_back(std::move(b));
        v.push_back(make_unique<Resursa>("B"));
        cout << "in vector: " << v.size() << '\n';
    }                                                  // vectorul moare: eliberează A și B
    cout << "-- shared_ptr --\n";
    {
        shared_ptr<Resursa> p = make_shared<Resursa>("C");
        cout << "proprietari: " << p.use_count() << '\n';
        {
            shared_ptr<Resursa> q = p;                 // al doilea proprietar
            cout << "proprietari: " << p.use_count() << '\n';
        }
        cout << "proprietari: " << p.use_count() << '\n';
    }                                                  // ultimul proprietar dispare: C eliberat
    cout << "-- weak_ptr --\n";
    {
        auto tata = make_shared<Nod>("tata");
        auto fiu = make_shared<Nod>("fiu");
        tata->copil = fiu;
        fiu->parinte = tata;
        if (auto t = fiu->parinte.lock()) cout << "parintele fiului: " << t->nume << '\n';
        cout << "fiu are " << fiu.use_count() << " proprietari\n";
```

```
    }                                                   // fără weak_ptr, ciclul n-ar fi
eliberat niciodată
    cout << "-- gata --\n";
    return 0;
}
```

DE CE AȘA?

De ce `make_unique` / `make_shared` și nu `unique_ptr<T>(new T(...))` ? Sunt mai scurte, nu lasă niciun `new` vizibil și sunt sigure la excepții; `make_shared` alocă obiectul și blocul de control într-o singură alocare. **De ce `weak_ptr` la părinte?** Dacă și copilul ar ține un `shared_ptr` la părinte, s-ar forma un **ciclu**: fiecare ține numărătoarea celuilalt peste 0, iar niciunul nu ar fi eliberat vreodată (scurgere de memorie). `weak_ptr` nu crește numărătoarea; `lock()` întoarce un `shared_ptr` temporar, sau unul gol dacă obiectul a murit. **Afișarea `fiu are 2 proprietari :`** variabila `fiu` și membrul `tata->copil` .

IMPORTANT

Regula de alegere: **`unique_ptr` implicit**; `shared_ptr` doar când proprietatea este cu adevărat împărțită; `weak_ptr` pentru a observa un obiect partajat fără a-l ține în viață; pointer simplu sau referință pentru acces fără proprietate. Un `shared_ptr` are cost: numărătoare atomică și un bloc de control.

5. Lambda

O **lambda** este o funcție anonimă scrisă direct în expresie. Compilatorul generează din ea un obiect-funcție (lecția oop-03).

```
[captura](parametri) mutable -> tip_intors { corp }
```

Captura	Înseamnă
`[]`	nu folosește nimic din exterior
`[x]`	copie a lui `x` , făcută la crearea lambdei
`[&x]`	referință la `x` (vede și modifică originalul)
`[=]` / `[&]`	tot ce folosește, prin valoare / prin referință
`[this]`	membrii obiectului curent (într-o metodă)
`[k = 0]`	variabilă nouă, proprie lambdei (C++14)

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

int main()
{
    int n, prag;
    cin >> n >> prag;
    vector<pair<string, int>> elevi(n);
    for (auto &e : elevi) cin >> e.first >> e.second;

    // 1. lambda ca și comparator: notă descrescător, apoi nume crescător
    sort(elevi.begin(), elevi.end(), [](const auto &a, const auto &b) {
        if (a.second != b.second) return a.second > b.second;
        return a.first < b.first;
    });
    for (const auto &e : elevi) cout << e.first << ' ';
    cout << '\n';

    // 2. captură prin valoare: lambda „ține minte” pragul
    int peste = count_if(elevi.begin(), elevi.end(), [prag](const auto &e) { return
e.second >= prag; });
    cout << peste << '\n';

    // 3. captură prin referință: lambda modifică o variabilă din afară
    int suma = 0;
    for_each(elevi.begin(), elevi.end(), [&suma](const auto &e) { suma += e.second; });
    cout << suma << '\n';

    // 4. lambda cu stare proprie (mutable): un contor
    auto urmatorul = [k = 0]() mutable { return ++k; };
    urmatorul(); urmatorul();
    cout << urmatorul() << '\n';

    // 5. std::function: o tabelă de operații alese după nume
    map<string, function<long long(long long, long long)>> op = {
        {"+", [](long long a, long long b) { return a + b; }},
        {"*", [](long long a, long long b) { return a * b; }},
        {"max", [](long long a, long long b) { return max(a, b); }}};
    string c;
    long long a, b;
    while (cin >> c >> a >> b)
        if (op.count(c)) cout << op[c](a, b) << '\n';
        else cout << "operatie necunoscuta\n";
    return 0;
}
```

DE CE AȘA?

De ce `mutable`? O lambda capturată prin valoare are `operator() const`: nu își poate modifica copiile. `mutable` îi permite să-și schimbe starea proprie (aici, contorul `k`). **Ce este `std::function`?** Un tip care poate păstra **orice** lucru apelabil cu o anumită semnătură (funcție, lambda, functor). Costă puțin (alocare și apel indirect), dar permite tabele de operații, callback-uri și cozi de evenimente (șabloanele Command și Observer, lecția oop-09). **`const auto &a`** în parametrii lambdei face lambda **generică** (un șablon, C++14).

ATENȚIE — GREȘELI FRECVENTE

Captura prin referință a unei variabile locale care moare este o greșeală gravă: dacă lambda este păstrată (într-un `std::function`, un thread, o coadă) și apelată după ieșirea din funcție, referința arată spre memorie eliberată. Când lambda trăiește mai mult decât blocul curent, capturează **prin valoare**.

6. `std::optional`: „poate nu există”

Pentru funcții la care lipsa rezultatului este **normală** (căutarea unui element), o excepție este prea mult, iar o valoare-santinelă (`-1`) este ambiguă. `std::optional<T>` exprimă direct „un `T` sau nimic”:

```
optional<int> cauta(const vector<int> &v, int x)
{
    for (int i = 0; i < (int)v.size(); i++)
        if (v[i] == x) return i;
    return nullopt;                          // nu am găsit
}
// if (auto p = cauta(v, 7)) cout << *p; else cout << "lipsă";
```

7. Greșeli frecvente

Greșeala	Ce se întâmplă	Corect
`catch (exception e)` prin valoare	copie și slicing	`catch (const exception &e)`
`catch` general înaintea celor specifice	cele specifice nu se mai ating	de la specific la general
excepție aruncată din destructor	`std::terminate`	destructori `noexcept`
`new` fără smart pointer și excepție între ele	scurgere de memorie	`make_unique`
`shared_ptr` peste tot	cost și cicluri greu de văzut	`unique_ptr` implicit
ciclu de `shared_ptr`	memorie niciodată eliberată	`weak_ptr` pe una dintre direcții
două `shared_ptr` construite din același pointer brut	dublu `delete`	copiază `shared_ptr`-ul, nu pointerul brut

Greșeala	Ce se întâmplă	Corect
lambda cu `[&]` păstrată după ieșirea din funcție	referințe moarte	capturează prin valoare
excepții pentru control normal (de exemplu, sfârșit de buclă)	cod lent și greu de citit	condiții obișnuite sau `optional`

8. În practică și la interviu

Întrebări clasice: „ce este RAII și cum ajută la excepții?", „diferența `unique_ptr` / `shared_ptr` / `weak_ptr`?", „cum apare o scurgere de memorie cu `shared_ptr`?", „ce garanții la excepții cunoști?", „ce capturează `[=]` într-o metodă?". În proiectele mari, regulile din Core Guidelines interzic aproape complet `new` / `delete` manual în codul de aplicație.

9. Exersează pe hârtie

1. Ce afișează programul cu excepții pentru comenzile `div 9 3`, `idx -1`, `rad 2.25`?

2. De ce un `catch (const exception &e)` pus primul ar prinde și `EroareMatematica`?

3. Pentru programul cu `Martor`, ce s-ar afișa dacă `nivel1` ar avea un `try` / `catch` propriu în jurul apelului `nivel2(x)`, care doar afișează „prins în nivel1"?

4. Ce garanție oferă `vector::push_back` și ce înseamnă pentru vectorul tău dacă alocarea eșuează?

5. De ce nu se poate copia un `unique_ptr`? Cum îl pui într-un `vector`?

6. Ce valori afișează `use_count()`: `auto a = make_shared<int>(5); auto b = a; { auto c = b; cout << a.use_count(); } cout << a.use_count();`?

7. Desenează ciclul care ar apărea dacă `parinte` ar fi `shared_ptr`. De ce nu se mai eliberează?

8. Ce afișează `int x = 1; auto f = [x]() { return x * 10; }; x = 5; cout << f();`? Dar cu `[&x]`?

9. Scrie o lambda care primește un `vector<int>` și întoarce suma pătratelor.

10. Când ai folosi `optional<int>` în loc de o excepție?

10. Mini-grile

1. Excepțiile se prind de preferință: a) prin valoare b) prin referință constantă c) prin pointer d) cu `catch (...)` mereu

2. Când o excepție părăsește o funcție, obiectele ei locale: a) rămân în memorie b) sunt distruse automat (derularea stivei) c) sunt copiate d) provoacă eroare de compilare

3. Pentru un obiect cu un singur proprietar se folosește: a) `shared_ptr` b) `weak_ptr` c) `unique_ptr` d) `auto_ptr`

4. `[&suma]` într-o lambda înseamnă: a) copie a lui `suma` b) referință la `suma` c) `suma` este constantă d) eroare

11. Mai departe (pentru cine vrea mai mult)

- **`std::expected`** (C++23): rezultat sau eroare, explicit în tip.
- **`noexcept` condiționat** și `std::move_if_noexcept`.
- **Ștergători proprii** pentru `unique_ptr` (de exemplu, `fclose` pentru `FILE*`).
- **`enable_shared_from_this`**: un obiect care își dă singur un `shared_ptr`.
- **Lambda recursive**, lambda în `constexpr`, captura `[*this]` (C++17).

Rezumat

- `throw` aruncă, `try` / `catch` prinde; prinde prin `const &`, de la specific la general; excepțiile proprii derivă din `runtime_error` / `logic_error`.
- Derularea stivei distruge obiectele locale ⇒ RAII face codul sigur la excepții; destructorii nu aruncă.
- Garanții: nothrow, puternică, de bază.
- `unique_ptr` (implicit, mutabil), `shared_ptr` (numărătoare), `weak_ptr` (observă, rupe cicluri); `make_unique` / `make_shared`.
- Lambda: `[captură](param) { … }`; prin valoare, prin referință, `mutable`; `std::function` păstrează orice apelabil.
- `optional` pentru „poate lipsi”, excepții pentru erori reale.

Citește și exersează în plus

- learncpp.com — Exceptions — capitolul complet despre excepții (engleză)
  https://www.learncpp.com/cpp-tutorial/the-need-for-exceptions/
- cppreference — Smart pointers (memory) — unique_ptr, shared_ptr, weak_ptr (engleză)
  https://en.cppreference.com/w/cpp/memory
- cppreference — Lambda expressions — sintaxa completă și capturile (engleză)
  https://en.cppreference.com/w/cpp/language/lambda

Răspunsuri

Exersează pe hârtie

1. `3`, apoi `eroare: index in afara vectorului` (`at(-1)`: indexul devine un număr foarte mare, deci în afara limitelor), apoi `1.5`.

2. `EroareMatematica` derivă din `runtime_error`, care derivă din `exception`; primul `catch` compatibil câștigă.

3. `+main`, `+nivel1`, `+nivel2`, `-nivel2`, `prins în nivel1`, `nivel1 terminat`, `-nivel1`, `-main`, `program continuat` (excepția este prinsă în `nivel1`; `main` nu o mai vede).

4. Garanție puternică: dacă alocarea eșuează, vectorul rămâne exact ca înainte (aceleași elemente, aceeași mărime).

5. Proprietatea este unică; două copii ar face `delete` de două ori. În vector se pune prin mutare: `v.push_back(std::move(p));` sau direct `v.push_back(make_unique<T>(...));`.

6. `3`, apoi `2`.

7. tata → (shared) fiu și fiu → (shared) tata: fiecare are numărătoarea cel puțin 1 datorită celuilalt; când variabilele locale dispar, numărătoarele scad la 1, nu la 0, deci niciun obiect nu este distrus.

8. Cu `[x]`: `10` (copia s-a făcut când a fost creată lambda). Cu `[&x]`: `50`.

9. `auto sp = [](const vector<int> &v) { long long s = 0; for (int x : v) s += 1LL * x * x; return s; };`

10. Când lipsa rezultatului este un caz normal, așteptat (căutare fără rezultat, cheie inexistentă), nu o eroare.

Mini-grile

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