AVANSAT · POO ÎN C++ · LECȚIA 4

Moștenire și polimorfism: virtual, clase abstracte, vtable, override, destructor virtual

Durată: 90 min Nivel: avansat / pregătire pentru facultate și industrie Înainte: lecțiile oop-01, oop-02, xi-21

CE VEI ȘTI DUPĂ LECȚIE

- Construiești ierarhii de clase cu moștenire publică și explici ordinea constructorilor și destructorilor.
- Folosești funcții virtuale, override și final pentru polimorfism la rulare.
- Scrii clase abstracte (interfețe) cu funcții pur virtuale.
- Explici de ce destructorul unei baze polimorfice trebuie să fie virtual și ce este object slicing.
- Înțelegi mecanismul vtable și costul unui apel virtual.

1. Moștenirea: relația „este un”

O clasă **derivată** preia datele și metodele unei clase **de bază** și adaugă sau modifică ce are nevoie. Relația trebuie să fie **„este un”**: un câine *este un* animal, un cerc *este o* formă. Dacă relația este „are un” (o mașină *are un* motor), folosești **compunere** (un membru), nu moștenire (lecția oop-05).

```
class Animal { /* ... */ };
class Caine : public Animal { /* ce are în plus un câine */ };
```

Specificatorii de acces

În bază	Din clasa derivată	Din afară (cu moștenire `public`)
`public`	da	da
`protected`	da	nu
`private`	**nu**	nu

Moștenirea `public` păstrează interfața: tot ce era public în bază rămâne public. Moștenirile `protected` și `private` ascund interfața bazei și se folosesc rar (pentru „implementat cu ajutorul lui”, unde de obicei compunerea este mai clară).

SFAT

Folosește `protected` cu măsură: datele `protected` pot fi stricate de orice clasă derivată, deci invariantul bazei nu mai este apărat. De multe ori e mai bine ca datele să rămână `private`, iar baza să ofere metode `protected` pentru derivate.

2. Construcție și distrugere într-o ierarhie

Un obiect `Caine` **conține** o parte de `Animal`. La construire, **întâi** se construiește partea de bază (prin constructorul ales în lista de inițializare), **apoi** membrii derivatei, apoi corpul constructorului derivatei. La distrugere, ordinea este **inversă**.

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

class Animal
{
protected:
    string nume;                          // vizibil în clasele derivate, nu și din afară
public:
    explicit Animal(string n) : nume(n) { cout << "Animal(" << nume << ")\n"; }
    virtual ~Animal() { cout << "~Animal(" << nume << ")\n"; }
    virtual string sunet() const { return "..."; }
    string prezentare() const { return nume + " face " + sunet(); }   // apel virtual din
bază
};

class Caine : public Animal
{
    int varsta;
public:
    Caine(string n, int v) : Animal(n), varsta(v) { cout << "Caine(" << nume << ")\n"; }
    ~Caine() override { cout << "~Caine(" << nume << ")\n"; }
    string sunet() const override { return "ham"; }
};

void prinValoare(Animal a) { cout << "prin valoare: " << a.sunet() << '\n'; }
void prinReferinta(const Animal &a) { cout << "prin referinta: " << a.sunet() << '\n'; }

int main()
{
    Caine c("Rex", 3);
    cout << c.prezentare() << '\n';
    prinReferinta(c);
    prinValoare(c);                       // slicing: se copiază doar partea de Animal
    Animal *p = new Caine("Lord", 5);
    delete p;                             // destructor virtual: întâi ~Caine, apoi
~Animal
    return 0;
}
```

Linie	Afișare	Explicație
`Caine c("Rex", 3);`	Animal(Rex), Caine(Rex)	întâi baza, apoi derivata
`c.prezentare()`	Rex face ham	`prezentare` este din `Animal`, dar `sunet()` e virtual: se alege versiunea lui `Caine`

Linie	Afișare	Explicație
`prinReferinta(c)`	prin referinta: ham	referința la bază păstrează tipul real
`prinValoare(c)`	prin valoare: ..., ~Animal(Rex)	**slicing**: parametrul este un `Animal` nou, copiat doar din partea de bază; moare la ieșirea din funcție
`new Caine("Lord", 5)` și `delete p`	Animal(Lord), Caine(Lord), ~Caine(Lord), ~Animal(Lord)	destructorul virtual distruge corect prin pointerul la bază
sfârșitul lui `main`	~Caine(Rex), ~Animal(Rex)	ordine inversă construirii

3. Funcții virtuale: polimorfism la rulare

Fără `virtual`, funcția apelată se alege după **tipul declarat** al variabilei (la compilare). Cu `virtual`, se alege după **tipul real** al obiectului (la rulare), atunci când apelul se face printr-un **pointer** sau o **referință** la bază. Aceasta se numește **polimorfism dinamic** (sau legare târzie, *late binding*).

Apel	`sunet()` fără `virtual`	`sunet()` cu `virtual`
`Caine c; c.sunet();`	ham	ham
`Animal &r = c; r.sunet();`	...	**ham**
`Animal *p = &c; p->sunet();`	...	**ham**
`Animal a = c; a.sunet();` (copie)	...	... (slicing)

`override` și `final`

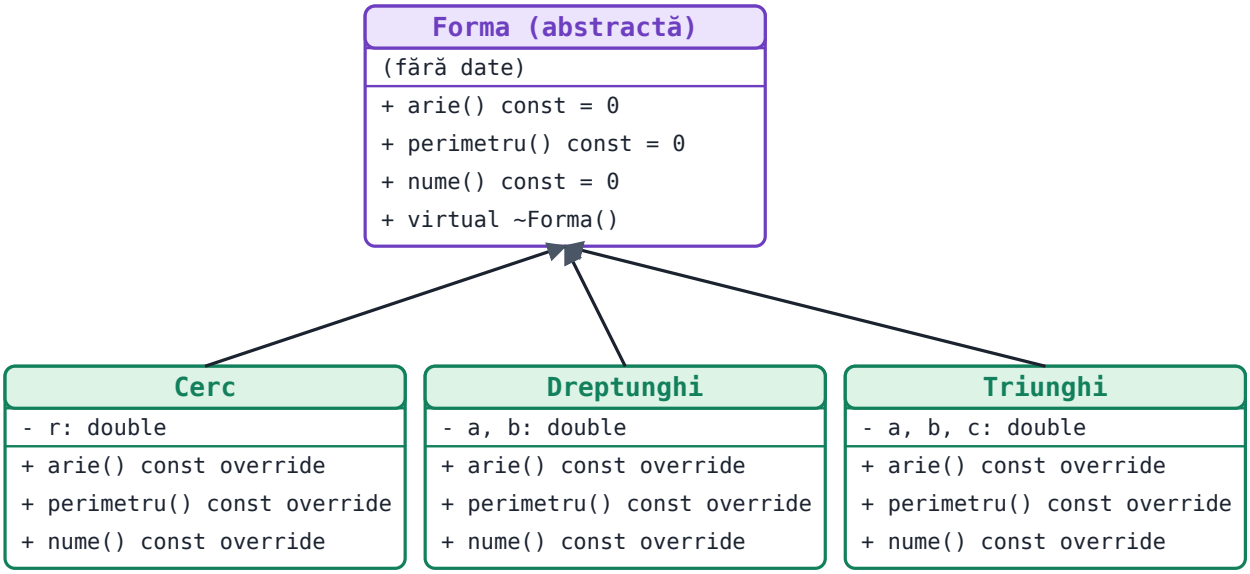
- `override` (după semnătură) cere compilatorului să verifice că funcția **chiar suprascrie** o funcție virtuală din bază. Dacă greșești semnătura (de exemplu, uiți `const`), primești eroare, în loc de o funcție nouă, nevirtuală, care nu este niciodată apelată prin bază.
- `final` interzice suprascrierea mai departe (la o funcție) sau derivarea (la o clasă: `class Cerc final : public Forma`).

> **ATENȚIE — GREȘELI FRECVENTE**
>
> **Suprascriere** (*override*) ≠ **supraîncărcare** (*overload*). Suprascrierea înlocuiește o funcție virtuală din bază, cu **aceeași** semnătură. Supraîncărcarea înseamnă mai multe funcții cu același nume și **parametri diferiți**. Dacă în derivată declari `sunet(int)`, aceasta **ascunde** toate `sunet` din bază (*name hiding*): `c.sunet()` nu se mai compilează fără `using Animal::sunet;`.

4. Clase abstracte și interfețe

O funcție **pur virtuală** (`= 0`) nu are implementare în bază: spune doar „orice derivată trebuie să aibă această funcție”. O clasă cu cel puțin o funcție pur virtuală este **abstractă**: nu se pot crea obiecte din

ea, doar pointeri și referințe. O clasă abstractă fără date, doar cu funcții pur virtuale, joacă rolul unei **interfețe**.



săgeata merge de la clasa derivată la clasa de bază („este o”)

Ierarhia de forme

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

class Forma                                 // clasă abstractă: interfața comună
{
public:
    virtual double arie() const = 0;        // pur virtuală: fiecare formă o definește
    virtual double perimetru() const = 0;
    virtual string nume() const = 0;
    virtual ~Forma() = default;             // destructor virtual: obligatoriu într-o bază
polimorfică
};

class Cerc : public Forma
{
    double r;
public:
    explicit Cerc(double raza) : r(raza) {}
    double arie() const override { return M_PI * r * r; }
    double perimetru() const override { return 2 * M_PI * r; }
    string nume() const override { return "cerc"; }
};

class Dreptunghi : public Forma
{
    double a, b;
public:
    Dreptunghi(double x, double y) : a(x), b(y) {}
    double arie() const override { return a * b; }
    double perimetru() const override { return 2 * (a + b); }
    string nume() const override { return "dreptunghi"; }
};

class Triunghi : public Forma
{
    double a, b, c;
public:
    Triunghi(double x, double y, double z) : a(x), b(y), c(z) {}
    double perimetru() const override { return a + b + c; }
    double arie() const override            // formula lui Heron
    {
        double s = perimetru() / 2;
        return sqrt(s * (s - a) * (s - b) * (s - c));
    }
    string nume() const override { return "triunghi"; }
};

int main()
{
    int n;
    cin >> n;
    vector<unique_ptr<Forma>> forme;        // pointeri la bază: pot arăta spre orice formă
```

```
    for (int i = 0; i < n; i++)
    {
        char tip;
        cin >> tip;
        if (tip == 'C') { double r; cin >> r; forme.push_back(make_unique<Cerc>(r)); }
        else if (tip == 'D') { double a, b; cin >> a >> b;
forme.push_back(make_unique<Dreptunghi>(a, b)); }
        else { double a, b, c; cin >> a >> b >> c; forme.push_back(make_unique<Triunghi>(a,
b, c)); }
    }
    cout << fixed << setprecision(2);
    double total = 0;
    const Forma *maxim = nullptr;
    for (const auto &f : forme)
    {
        cout << f->nume() << ' ' << f->arie() << '\n';   // apel virtual: se alege funcția
clasei reale
        total += f->arie();
        if (!maxim || f->arie() > maxim->arie()) maxim = f.get();
    }
    cout << "total " << total << '\n';
    if (maxim) cout << "cea mai mare: " << maxim->nume() << '\n';
    return 0;
}
```

DE CE AȘA?

De ce `vector<unique_ptr<Forma>>` și nu `vector<Forma>`? Un `vector<Forma>` nici nu se compilează (clasa este abstractă), iar un vector de obiecte de bază ar face slicing. Polimorfismul funcționează doar prin pointeri sau referințe; `unique_ptr` (lecția oop-07) este pointerul care face singur `delete`, deci nu pierdem memorie. **Beneficiul:** bucla din `main` nu știe ce forme există. Dacă adaugi `Hexagon`, nu modifici nimic din bucla de calcul: doar scrii clasa nouă (principiul deschis/închis, lecția oop-05).

5. Destructorul virtual

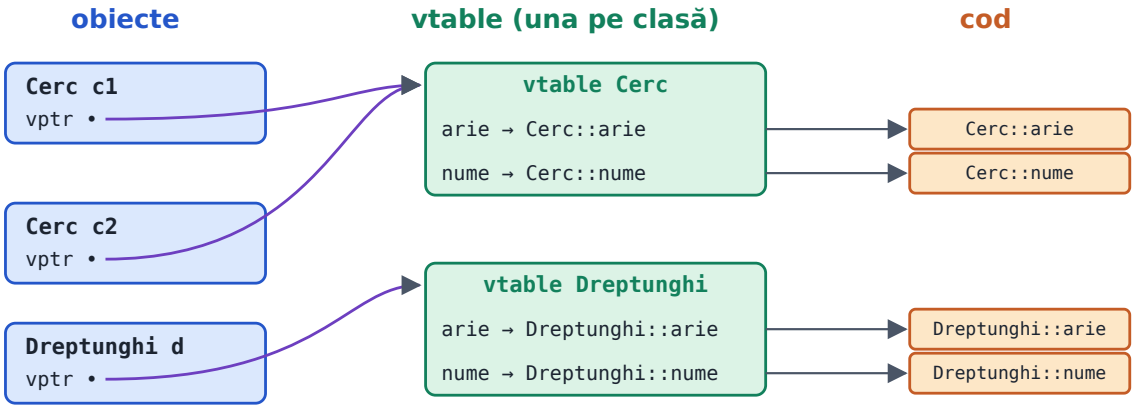
Dacă ștergi un obiect derivat printr-un pointer la bază (`Animal *p = new Caine(...); delete p;`) și destructorul bazei **nu** este virtual, se apelează doar `~Animal()`: partea de `Caine` nu este distrusă, iar standardul spune că este **comportament nedefinit**. Regula:

IMPORTANT

Orice clasă de bază cu funcții virtuale trebuie să aibă destructorul `virtual` (sau `protected` și nevirtual, dacă nu vrei ștergere prin bază). Cel mai simplu: `virtual ~Baza() = default;`.

6. Cum funcționează: vtable și vptr

Compilatorul construiește, pentru fiecare clasă cu funcții virtuale, o **tabelă de funcții virtuale** (*vtable*): adresele versiunilor care se folosesc pentru acea clasă. Fiecare obiect primește un pointer ascuns, **vptr**, către tabela clasei sale. Un apel `p->arie()` devine: citește `vptr` din obiect → citește adresa din tabelă → sare acolo.



f->arie() citește vptr-ul obiectului, apoi adresa din vtable: alegerea se face la rulare

vtable

Aspect	Cost
memorie	un pointer (8 octeți) în fiecare obiect + o tabelă pe clasă
timp	două citiri din memorie în plus pe apel; funcția nu mai poate fi făcută *inline*
când contează	în bucle de milioane de apeluri pe obiecte mici (atunci se folosesc șabloane, oop-06)

> **DE ȘTIUT**
> În constructorul și destructorul bazei, obiectul „este" încă (sau din nou) doar de tipul bazei: un apel virtual de acolo **nu** ajunge la versiunea derivatei. Nu apela funcții virtuale din constructori.

7. Object slicing și `dynamic_cast`

Slicing: când copiezi un obiect derivat într-o variabilă de tipul bazei (`Animal a = c;`, sau parametru prin valoare), se copiază doar partea de bază; datele și comportamentul derivatei se pierd. Evitare: transmite prin `const Baza&` sau prin pointer.

`dynamic_cast` verifică la rulare dacă un pointer la bază arată de fapt spre un anumit tip derivat:

```
if (auto c = dynamic_cast<Cerc*>(f)) cout << "este cerc";   // nullptr dacă nu este Cerc
```

Dacă ai nevoie des de `dynamic_cast`, de obicei înseamnă că o funcție virtuală lipsește din interfață.

8. Moștenire multiplă (pe scurt)

C++ permite mai multe clase de bază: `class Amfibian : public Masina, public Barca`. Problema clasică este **diamantul**: dacă `Masina` și `Barca` moștenesc amândouă din `Vehicul`, un `Amfibian` are **două** părți de `Vehicul`. Soluția este **moștenirea virtuală** (`class Masina : virtual public Vehicul`). În practică, moștenirea multiplă se folosește aproape exclusiv pentru **interfețe** (clase fără date): `class Fisier : public Citibil, public Scriibil`.

9. Greșeli frecvente

Greșeala	Ce se întâmplă	Corect
destructor nevirtual într-o bază polimorfică	`delete` prin bază: comportament nedefinit, scurgeri	`virtual ~Baza() = default;`
lipsește `override`, semnătură ușor diferită	funcție nouă, nevirtuală, niciodată apelată prin bază	scrie mereu `override`
obiecte derivate într-un `vector<Baza>`	slicing	`vector<unique_ptr<Baza>>`
parametru `Baza` prin valoare	slicing	`const Baza&`
apel virtual în constructor	se apelează versiunea bazei	evită; folosește o funcție de inițializare separată
`protected` pentru toate datele	derivatele pot strica invariantul	date `private`, metode `protected`
moștenire pentru reutilizare de cod fără „este un”	ierarhii fragile	compunere (oop-05)
supraîncărcare în derivată ascunde bază	`c.f()` nu se compilează	`using Baza::f;`

10. Exersează pe hârtie

1. În ce ordine se apelează constructorii și destructorii pentru `class A {…}; class B : public A {…}; class C : public B {…};` la `C x;`?

2. Fără `virtual`, ce afișează `Animal *p = new Caine("Rex", 3); cout << p->sunet();`? Dar cu `virtual`?

3. Care este diferența dintre o funcție virtuală și una pur virtuală? Se poate crea un obiect `Forma`?

4. O clasă derivată scrie `double arie() override { … }` (fără `const`), iar baza are `virtual double arie() const = 0;`. Ce se întâmplă?

5. Explică de ce `delete p` (cu `Animal *p = new Caine…`) este periculos dacă `~Animal()` nu este virtual.

6. Ce afișează `prinValoare(c)` pentru un `Caine` și de ce apare `~Animal(Rex)` imediat după?

7. Câți octeți în plus are un obiect cu funcții virtuale, pe un sistem de 64 de biți? De ce?

8. Adaugă clasa `Patrat` în ierarhie. Ar fi mai bine să derive din `Dreptunghi` sau direct din `Forma`? (Gândește-te la o metodă `setLatime` a dreptunghiului.)

9. Ce este problema diamantului și cum se rezolvă?

10. Când folosești `final` pe o clasă?

11. Mini-grile

1. Polimorfismul dinamic funcționează când apelul se face prin: a) obiect copiat b) pointer sau referință la bază, cu funcție virtuală c) funcție statică d) orice variabilă

2. O clasă cu o funcție `virtual void f() = 0;` este: a) abstractă b) finală c) statică d) template

3. `override` ajută pentru că: a) face funcția mai rapidă b) compilatorul verifică faptul că funcția suprascrie una virtuală din bază c) face funcția statică d) interzice moștenirea

4. Object slicing apare când: a) folosești pointeri b) folosești `override` c) clasa e abstractă d) copiezi un obiect derivat într-o variabilă de tipul bazei

12. Mai departe (pentru cine vrea mai mult)

- **NVI (Non-Virtual Interface):** metode publice nevirtuale care apelează metode private virtuale.
- **CRTP:** polimorfism la compilare, fără vtable (lecția oop-06).
- **`std::variant` + `std::visit`:** alternativă la ierarhii pentru un set închis de tipuri.
- **Principiul substituției Liskov** (lecția oop-05): derivata trebuie să poată înlocui baza fără surprize.
- **Type erasure** (`std::function`): polimorfism fără moștenire publică.

Rezumat

- Moștenirea publică modelează „este un”; baza se construiește prima și se distruge ultima.
- `virtual` + pointer/referință la bază ⇒ se apelează funcția tipului real (legare la rulare).
- Scrie mereu `override` ; `final` oprește suprascrierea/derivarea.
- `= 0` face funcția pur virtuală și clasa abstractă; interfață = clasă abstractă fără date.
- Baza polimorfică are destructor virtual; altfel `delete` prin bază este nedefinit.
- Slicing: copierea în tipul bazei pierde derivata; colecții de obiecte polimorfice = `vector<unique_ptr<Baza>>` .
- Mecanism: vptr în obiect + vtable pe clasă; cost mic, dar real.

Citește și exersează în plus

- learncpp.com — Virtual functions and polymorphism — capitolul despre funcții virtuale, pas cu pas (engleză) https://www.learncpp.com/cpp-tutorial/virtual-functions/
- cppreference — virtual function specifier — reguli exacte, override și final (engleză) https://en.cppreference.com/w/cpp/language/virtual
- C++ Core Guidelines — Class hierarchies — când și cum folosești ierarhiile (engleză) https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#Sh-domain

Răspunsuri

Exersează pe hârtie

1. Construire: `A()`, `B()`, `C()`. Distrugere: `~C()`, `~B()`, `~A()`.

2. Fără `virtual`: `...` (versiunea din `Animal`, după tipul pointerului). Cu `virtual`: `ham`.

3. Una virtuală are implementare în bază, care poate fi suprascrisă; una pur virtuală (`= 0`) nu are implementare și **trebuie** suprascrisă. Nu: `Forma` este abstractă, nu se pot crea obiecte din ea.

4. Eroare de compilare: cu `override`, compilatorul vede că nu există o funcție virtuală `arie()` non-`const` în bază. (Fără `override`, ar fi fost o funcție nouă, iar `Dreptunghi` ar fi rămas abstractă.)

5. Se apelează doar `~Animal()`; standardul spune că este comportament nedefinit. Practic, destructorul lui `Caine` nu rulează, iar resursele lui (memorie, fișiere) nu sunt eliberate.

6. „prin valoare: ...” : parametrul este un `Animal` nou, construit prin copiere doar din partea de bază (slicing), deci `sunet()` din `Animal`. La ieșirea din funcție parametrul moare: `~Animal(Rex)`.

7. 8 octeți (pointerul `vptr`), plus eventual umplutură pentru aliniere. Obiectul trebuie să știe ce tabelă de funcții virtuale folosește.

8. Direct din `Forma`. Un pătrat derivat din dreptunghi încalcă principiul substituției: codul care face `d.setLatime(5)` și se așteaptă ca înălțimea să rămână aceeași ar fi surprins de un pătrat care își schimbă ambele laturi.

9. Când o clasă moștenește din două clase care au aceeași bază, conține două copii ale bazei (ambiguitate). Se rezolvă cu moștenire virtuală (`: virtual public Baza`), care păstrează o singură copie.

10. Când clasa nu este gândită pentru a fi extinsă (de exemplu, invariantul ar putea fi stricat de o derivată), sau ca să permiți compilatorului să elimine apelurile virtuale (devirtualizare).

Mini-grile

1. b). **2.** a) abstractă. **3.** b). **4.** d).