Laborator 6

Tema: Moștenire - I ⮯

Poveste

O universitate dorește să dezvolte un sistem simplu pentru a gestiona datele persoanelor din campus. Sistemul trebuie să stocheze informații de bază pentru orice persoană, cum ar fi numele și vârsta. Pentru studenți, care sunt un tip special de persoană, sistemul trebuie să memoreze și informații suplimentare, precum facultatea și media. Acest scenariu este perfect pentru a aplica conceptul de **moștenire**, unde clasa `Student` va moșteni atributele și comportamentul clasei de bază `Persoana`.

Scopul laboratorului

- înțelegerea conceptului de moștenire și a relației "is-a" (un Student "este o" Persoană);
- implementarea unei clase de bază (`Persoana`) și a uneia derivate (`Student`);
- gestionarea memoriei alocate dinamic pentru atribute de tip `char*`;
- implementarea corectă a constructorilor, destructorului, constructorului de copiere și operatorului de atribuire (Regula celor Trei);
- apelarea constructorului din clasa de bază folosind lista de inițializare a membrilor;
- utilizarea membrilor `protected` pentru a oferi acces claselor derivate;
- suprascrierea operatorilor de citire/afișare pentru ierarhia de clase.

Structura fișierelor

Se vor crea fișierele sursă și header corespunzătoare pentru clasele de mai jos. Implementarea metodelor se va face în fișierele `.cpp`.

- **persoana.h** / **persoana.cpp** — declararea și implementarea clasei de bază;
- **student.h** / **student.cpp** — clasa derivată;
- **main.cpp** — testarea funcționalităților, inclusiv cu un șir de obiecte alocat dinamic.

Fișierul `persoana.h`

```h
#ifndef PERSOANA_H
#define PERSOANA_H
#include <iostream>

class Persoana {
protected:
    char* nume;
    int varsta;

public:
    // Constructor
    Persoana(const char* nume_init = "Anonim", int varsta_init = 0);
    // Destructor
    ~Persoana();
    // Constructor de copiere
    Persoana(const Persoana& other);
    // Operator de atribuire
    Persoana& operator=(const Persoana& other);

    // Operatori de stream
    friend std::istream& operator>>(std::istream& in, Persoana& p);
    friend std::ostream& operator<<(std::ostream& out, const Persoana& p);
};

#endif
```

Fișierul `student.h`

```h
#ifndef STUDENT_H
#define STUDENT_H

#include "persoana.h"

class Student : public Persoana {
private:
    char* facultate;
    float medie;

public:
    // Constructor
    Student(const char* nume_init = "Anonim", int varsta_init = 0, const char*
    ↪  fac_init = "N/A", float medie_init = 5.0);
    // Destructor
    ~Student();
    // Constructor de copiere
    Student(const Student& other);
    // Operator de atribuire
    Student& operator=(const Student& other);

    // Operatori de stream
    friend std::istream& operator>>(std::istream& in, Student& s);
    friend std::ostream& operator<<(std::ostream& out, const Student& s);
};

#endif
```

Fișierul `main.cpp` (model)

```cpp
#include "student.h" // Includem doar student.h, care la randul sau include
↪  persoana.h
#include <iostream>
using namespace std;

int main() {

    // TODO 1: Testarea clasei Persoana
    // Persoana p1("Popescu Ion", 30);
    // Persoana p2;
    // cout << "Persoana p1: " << p1 << endl;
    // cout << "Introduceti date pentru a doua persoana (Nume Prenume Varsta):"
    ↪  << endl;
    // cin >> p2;
    // cout << "Persoana p2: " << p2 << endl;

    // TODO 2: Testarea clasei Student
    // Student s1("Ionescu Vasile", 21, "ETTI", 8.75);
    // cout << "Student s1: " << s1 << endl;

    // TODO 3: Testarea constructorului de copiere si a operatorului de
    ↪  atribuire
    // Student s2 = s1; // Constructor de copiere
    // Student s3;
    // s3 = s1; // Operator de atribuire
    // cout << "Student s2 (copie s1): " << s2 << endl;
    // cout << "Student s3 (atribuire s1): " << s3 << endl;

    // TODO 4: Testarea cu un sir de studenti alocat dinamic
    // Student* studenti = new Student[5];
    // studenti[0] = Student("Popa Lucian", 22, "ACS", 9.00);
    // studenti[1] = Student("Berca Ioana", 23, "FILS", 8.23);
    // tot asa

    // cout << "\n--- Afisare lista studenti ---" << endl;
    // for (int i = 0; i < n; i++) {
    //     cout << studenti[i] << endl;
    // }

    // TODO 5: Nu uitati sa eliberati memoria!
    // delete[] studenti;

    return 0;
}
```

Explicații suplimentare

- **Moștenire publică (`public`)** — Clasa `Student` moștenește toți membrii publici și protejați ai clasei `Persoana`. Membrii `protected` din bază (`nume`, `varsta`) sunt direct accesibili în metodele clasei derivate `Student`.
- **Regula celor Trei** — Deoarece clasele noastre gestionează direct memorie (prin `char*`), trebuie să respectăm "Regula celor Trei": dacă o clasă are nevoie de un destructor, un constructor de copiere sau un operator de atribuire definit explicit, atunci probabil are nevoie de toate trei.
  - **Destructorul**: eliberează memoria alocată cu `new[]`. Ex: `delete[] nume;`
  - **Constructorul de copiere**: alocă memorie nouă și copiază conținutul, nu doar pointerul (deep copy).
  - **Operatorul de atribuire**: previne auto-atribuirea, eliberează memoria veche, apoi alocă memorie nouă și copiază conținutul (deep copy).
- **Constructor Chaining** — Constructorul clasei derivate trebuie să apeleze constructorul clasei de bază pentru a inițializa membrii moșteniți. Acest lucru se face prin lista de inițializare a membrilor.

```
Student::Student(...) : Persoana(nume_init, varsta_init) {
    // ... initializare membri proprii (facultate, medie)
}
```

- **Reutilizarea codului** — În operatorii de citire/afișare din `Student`, putem apela versiunile din clasa de bază folosind o conversie explicită:

```
// in operatorul << pentru Student
out << (Persoana&)student;
// sau
operator<<(out, (Persoana&)student);
```

Cerințe (9p + 1p oficiu)

1. **Implementare Clasa Persoana (4p)**
   - (1p) Constructor care alocă dinamic memorie pentru `nume`.
   - (1p) Destructor care eliberează memoria.
   - (1p) Constructor de copiere și operator de atribuire (implementare deep copy).
   - (1p) Supraincărcarea operatorilor de stream `«` și `»`.
2. **Implementare Clasa Student (3p)**
   - (1p) Derivarea corectă, constructor care apelează constructorul din bază și gestionează alocarea pentru `facultate`.
   - (1p) Destructor, constructor de copiere și operator de atribuire.
   - (1p) Supraincărcarea operatorilor de stream care reutilizează codul din clasa de bază.
3. **Testare în main.cpp (2p)**
   - (1p) Testarea corectă a constructorilor, destructorilor și operatorilor pentru obiecte individuale.
   - (1p) Testarea cu un șir de obiecte alocat dinamic (creare, citire, afișare, dealocare).
4. **(1p) Punct din oficiu.**

Compilare și verificare

```
g++ -g main.cpp persoana.cpp student.cpp -o lab6
./lab6
valgrind ./lab6
```

Bonus GitHub Classroom

Studenții pot obține până la **3p bonus** dacă încarcă toate laboratoarele până la finalul semestrului, respectiv **1p bonus** dacă încarcă mai mult de jumătate dintre ele.

Pași pentru încărcarea laboratorului:

(a) Accesați cursul de pe OCW – Programare Orientată pe Obiecte, secțiunea **Resurse** → **Bonus GitHub laborator**.

(b) Acolo veți găsi linkul **GitHub Classroom** pentru Laboratorul 6:

`https://classroom.github.com/a/ZHG6I0Xp`

(c) După acceptare, veți primi propriul repository, ex. `laborator-6-NumePrenume`.

(d) Clonați repository-ul local:

```
git clone
↪  git@github.com:Laborator-POO-2025-2026/laborator-6-NumePrenume.git
cd laborator-6-NumePrenume
```

(e) După ce ați adăugat fișierele `persoana.h`, `persoana.cpp`, `student.h`, `student.cpp` și `main.cpp`, urmează comenzile:

```
git add .
git commit -m "Implementare laborator 6 - Mostenire"
git push origin main
```

Atenție: laboratoarele copiate între studenți sunt considerate nevalide, iar niciunul dintre cei doi participanți nu va primi bonusul. Repository-urile sunt monitorizate automat de sistemul GitHub Classroom.