Laborator 7

Tema: Moștenire - II & Agregare

Poveste

O primărie dorește să dezvolte un sistem informatic pentru a gestiona datele la nivel de oraș. Sistemul trebuie să stocheze informații despre **locuințele** din oraș (adresă, valoare), taxele și **impozitele** asociate (valori, număr). Fiecare **locuitor** al orașului este caracterizat de nume, dar și de o locuință și de taxele pe care le are de plătit. Astfel, un locuitor moștenește caracteristicile unei locuințe și pe cele ale unui plătitor de impozit. La nivel de **oraș**, sistemul trebuie să gestioneze o listă de locuitori și să aibă un primar desemnat.

Acest scenariu este perfect pentru a aplica conceptele de **moștenire multiplă** (un `Locuitor` este și o entitate ce deține o `Locuinta`, dar are și de plătit un `Impozit`) și **agregare** (un `Oras` este compus dintr-o colecție de `Locuitori`).

Scopul laboratorului

- înțelegerea conceptului de **moștenire multiplă**;
- înțelegerea conceptului de **agregare** (relația "has-a");
- gestionarea memoriei alocate dinamic în contexte de moștenire și agregare;
- apelarea constructorilor din clasele de bază multiple folosind lista de inițializare;
- utilizarea **namespace**-urilor pentru o mai bună organizare a codului.

Structura fișierelor

Se vor crea fișierele sursă și header corespunzătoare pentru clasele de mai jos. Implementarea metodelor se va face în fișierele `.cpp`. **Recomandare:** Înveliți toate declarațiile de clase într-un namespace, de ex. `administratie`.

- **impozit.h** / **impozit.cpp**
- **locuinta.h** / **locuinta.cpp**
- **locuitor.h** / **locuitor.cpp** — clasă derivată din `Impozit` și `Locuinta`;
- **oras.h** / **oras.cpp** — clasă ce folosește agregarea de obiecte `Locuitor`;
- **main.cpp** — testarea funcționalităților.

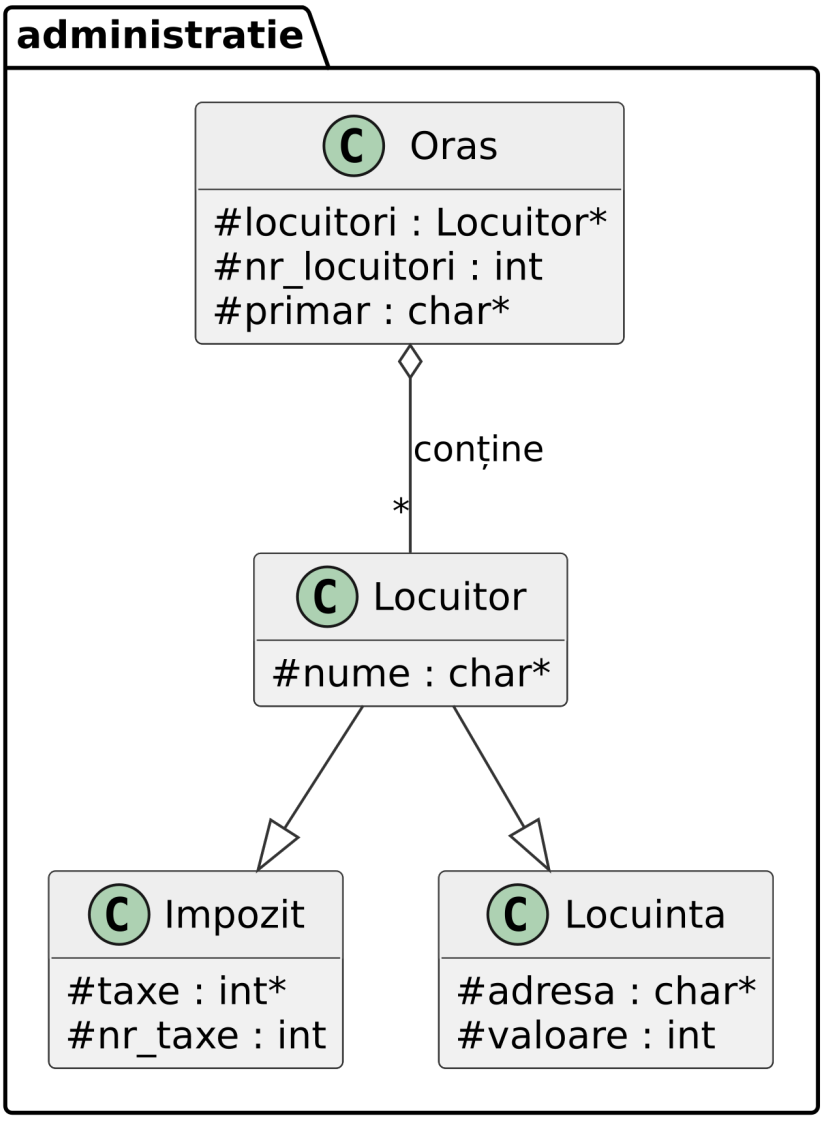

Figura 1: Diagrama UML a ierarhiei de clase

Fișierul `impozit.h`

```h
#pragma once
#include <iostream>

class Impozit {
protected:
    int *taxe;
    int nr_taxe;

public:
    Impozit(int* taxe_init = nullptr, int nr_taxe_init = 0);
    ~Impozit();
    Impozit(const Impozit& other);
    Impozit& operator=(const Impozit& other);

    // Metoda pentru a calcula suma taxelor, utila la sortare
    int getSumaTaxe() const;

    friend std::istream& operator>>(std::istream& in, Impozit& i);
    friend std::ostream& operator<<(std::ostream& out, const Impozit& i);
};
```

Fișierul `locuinta.h`

```h
#pragma once
#include <iostream>
#include <cstring>

class Locuinta {
protected:
    char *adresa;
    int valoare;

public:
    Locuinta(const char* adresa_init = "N/A", int valoare_init = 0);
    ~Locuinta();
    Locuinta(const Locuinta& other);
    Locuinta& operator=(const Locuinta& other);

    friend std::istream& operator>>(std::istream& in, Locuinta& l);
    friend std::ostream& operator<<(std::ostream& out, const Locuinta& l);
};
```

Fișierul `locuitor.h`

```
#pragma once

#include "impozit.h"
#include "locuinta.h"

class Locuitor : public Impozit, public Locuinta {
private:
    char *nume;

public:
    Locuitor(const char* nume_init = "N/A",
             const char* adresa_init = "N/A", int valoare_init = 0,
             int* taxe_init = nullptr, int nr_taxe_init = 0);
    ~Locuitor();
    Locuitor(const Locuitor& other);
    Locuitor& operator=(const Locuitor& other);

    // Functie friend pentru a putea accesa membrii privati/protejati la sortare
    friend void sorteazaLocuitori(Locuitor* locuitori, int n);

    friend std::istream& operator>>(std::istream& in, Locuitor& l);
    friend std::ostream& operator<<(std::ostream& out, const Locuitor& l);
};
```

Fișierul `oras.h`

```h
#pragma once

#include "locuitor.h"

class Oras {
private:
    Locuitor *locuitori;
    int nr_locuitori;
    char *primar;
public:
    Oras(Locuitor* locuitori_init = nullptr, int nr_locuitori_init = 0, const
    ↪  char* primar_init = "N/A");
    ~Oras();
    Oras(const Oras& other);
    Oras& operator=(const Oras& other);

    void schimbaPrimar(const char* nume_nou);
    void adaugaLocuitor(const Locuitor& new_loc, int pozitie);

    friend std::ostream& operator<<(std::ostream& out, const Oras& o);
};
```

Indicații pentru `main.cpp`

```cpp
#include "oras.h" // Include tot ce este necesar

// Daca folositi namespace-uri: using namespace administratie;

int main() {

    // 1. Creati dinamic un vector de cel putin 10 locuitori.
    // Initializati fiecare locuitor cu date relevante (nume, adresa, taxe
    ↪  etc.).
    // Pentru simplitate, hardcodati datele, NU CITITI DE LA TASTATURA.

    // 2. Creati un obiect de tip Oras folosind vectorul de locuitori.
    // Afisati starea initiala a orasului.

    // 3. Sortati vectorul de locuitori din oras in functie de suma totala
    // de taxe platita de fiecare. Puteti implementa o functie globala
    // sau o metoda a clasei Oras pentru aceasta.
    // Afisati orasul dupa sortare pentru a verifica corectitudinea.

    // 4. Schimbati numele primarului din obiectul Oras.
    // Afisati din nou orasul pentru a vedea modificarea.

    // 5. Adaugati un nou locuitor pe o pozitie data (ex: pozitia 3).
    // Vectorul de locuitori se va mari.
    // Afisati starea finala a orasului.

    // 6. Asigurati-va ca toata memoria alocata dinamic este eliberata la final!
    // Valgrind este prietenul vostru.

    return 0;
}
```

Explicații suplimentare

- **Moștenire Multiplă** — Clasa `Locuitor` moștenește membrii publici și protejați din **ambele** clase de bază: `Impozit` și `Locuinta`. Acest lucru permite unui obiect `Locuitor` să aibă atât caracteristici legate de taxe, cât și de locuință.
- **Agregare** — Clasa `Oras` nu moștenește, ci **conține** un vector de obiecte de tip `Locuitor`. Aceasta este o relație de tip "has-a" (un oraș "are" locuitori). Clasa `Oras` este responsabilă pentru managementul ciclului de viață al vectorului de locuitori.
- **Regula celor Trei/Cinci** — Deoarece toate clasele gestionează memorie alocată dinamic (`char*`, `int*`, `Locuitor*`), este esențială implementarea corectă a destructorului, constructorului de copiere și operatorului de atribuire pentru a evita probleme precum memory leaks sau double free.
- **Constructor Chaining (Moștenire Multiplă)** — Constructorul clasei derivate `Locuitor` trebuie să apeleze explicit constructorii ambelor clase de bază pentru a inițializa corect toți membrii moșteniți.

```
Locuitor::Locuitor(...) : Impozit(taxe_init, nr_taxe_init),
↪  Locuinta(adresa_init, valoare_init) {
    // ... initializare membru propriu (nume)
}
```

- **Namespace-uri** — Pentru a evita coliziunile de nume (de ex., dacă ar exista o altă clasă `Oras` într-o altă bibliotecă), este o bună practică să încapsulați clasele într-un namespace.

```
// in fisierele .h
namespace administratie {
    class Impozit { /* ... */ };
    // ... alte clase
}
// in fisierele .cpp
namespace administratie {
    Impozit::Impozit(...) { /* ... */ }
}
```

Cerințe (9p + 1p oficiu)

1. **Implementare Clase de Bază (2p)**
   - (1p) Implementarea corectă (Regula celor Trei, operatori stream) pentru clasa `Impozit`.
   - (1p) Implementarea corectă (Regula celor Trei, operatori stream) pentru clasa `Locuinta`.
2. **Implementare Clasa Derivată (3p)**
   - (1p) Derivarea multiplă corectă și constructor care apelează constructorii din bază.
   - (1p) Implementarea corectă a Regulii celor Trei (destructor, cc, op=) pentru `Locuitor`.
   - (1p) Supraincărcarea operatorilor de stream care reutilizează codul din clasele de bază.
3. **Implementare Clasă Agregat (2p)**
   - (1p) Implementarea corectă a Regulii celor Trei pentru clasa `Oras`, gestionând vectorul de locuitori.
   - (1p) Implementarea metodelor specifice: schimbarea primarului, adăugarea unui nou locuitor.
4. **Testare în main.cpp (2p)**
   - (1p) Crearea unui obiect `Oras` cu cel puțin 10 locuitori și testarea afișării și modificării primarului.
   - (1p) Sortarea vectorului de locuitori și adăugarea unui nou locuitor pe o poziție dată.
5. **(1p) Punct din oficiu.**

Compilare și verificare

```
g++ -g main.cpp impozit.cpp locuinta.cpp locuitor.cpp oras.cpp -o lab7
./lab7
valgrind ./lab7
```

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 7:

`https://classroom.github.com/a/A24trjwB`

(c) După acceptare, veți primi propriul repository, ex. `laborator-7-NumePrenume`.

(d) Clonați repository-ul local:

```
git clone
↪  git@github.com:Laborator-POO-2025-2026/laborator-7-NumePrenume.git
cd laborator-7-NumePrenume
```

(e) După ce ați adăugat fișierele corespunzătoare (`impozit.h`, `impozit.cpp`, `locuinta.h`, `locuinta.cpp`, etc.), urmează comenzile:

```
git add .
git commit -m "Implementare laborator 7 - Mostenire Multipla si Agregare"
git push origin main
```