Laborator 7

Tema: Moștenire - II & Agregare

Poveste

Un producător auto dorește să modeleze componentele și produsele sale. Sistemul trebuie să stocheze informații despre **motoare** (cai putere, tip combustibil) și despre **caroserii** (culoare, număr de uși). Fiecare **mașină** produsă este definită atât de motorul său, cât și de caroseria sa, moștenind caracteristicile ambelor. La nivel de **garaj**, sistemul trebuie să gestioneze o flotă de mașini și să aibă un nume specific.

Acest scenariu este perfect pentru a aplica conceptele de **moștenire multiplă** (o `Masina` este definită de un `Motor` și o `Caroserie`) și **agregare** (un `Garaj` este compus dintr-o colecție de `Masini`).

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. `parc_auto`.

- **motor.h** / **motor.cpp**
- **caroserie.h** / **caroserie.cpp**
- **masina.h** / **masina.cpp** — clasă derivată din `Motor` și `Caroserie`;
- **garaj.h** / **garaj.cpp** — clasă ce folosește agregarea de obiecte `Masina`;
- **main.cpp** — testarea funcționalităților.

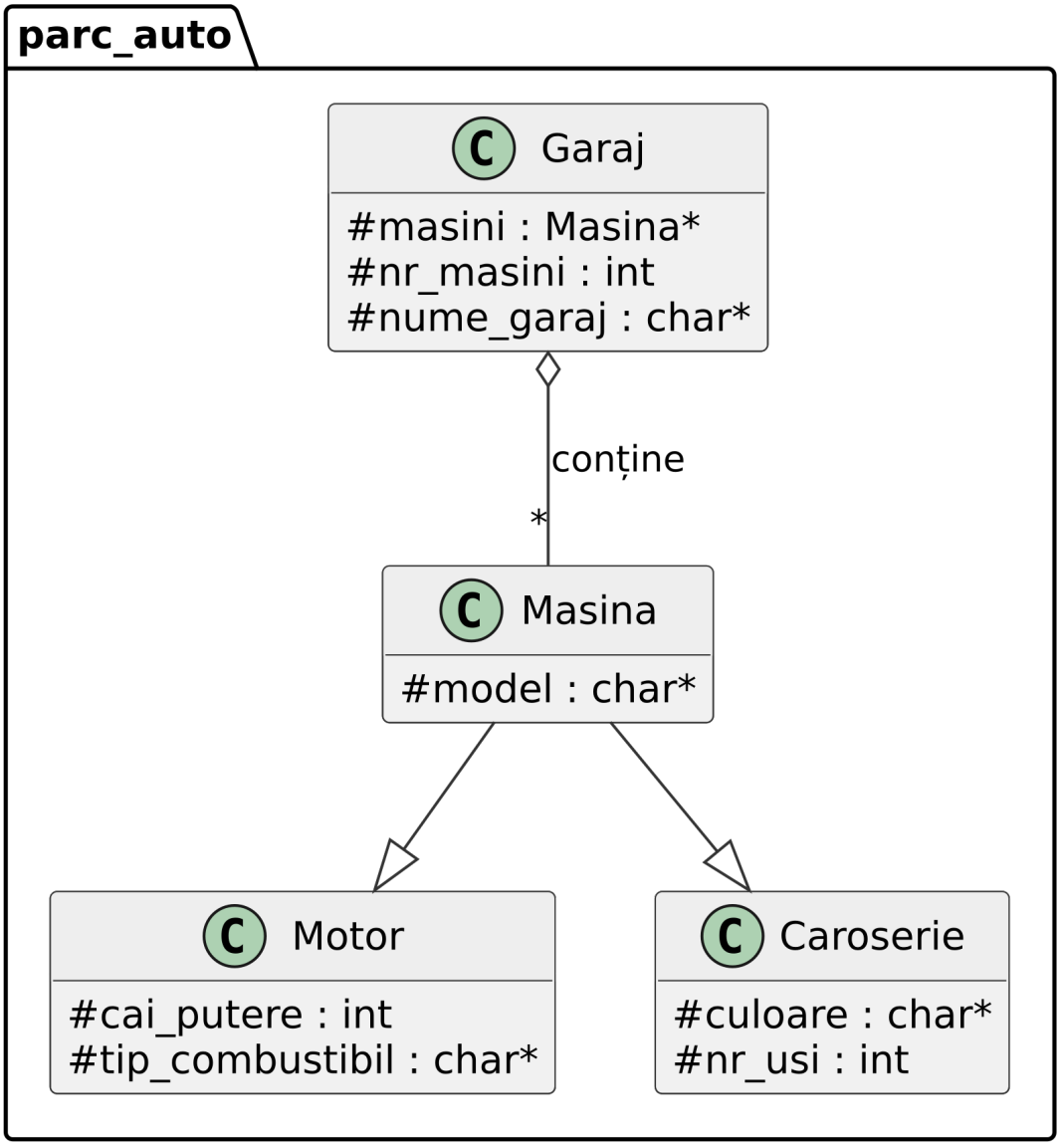

Figura 1: Diagrama UML a ierarhiei claselor.

Fișierul `motor.h`

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

class Motor {
protected:
    int cai_putere;
    char* tip_combustibil;

public:
    Motor(int cp_init = 0, const char* combustibil_init = "N/A");
    ~Motor();
    Motor(const Motor& other);
    Motor& operator=(const Motor& other);

    // Metoda utila la sortare
    int getCaiPutere() const;

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

Fișierul `caroserie.h`

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

class Caroserie {
protected:
    char *culoare;
    int nr_usi;

public:
    Caroserie(const char* culoare_init = "N/A", int usi_init = 0);
    ~Caroserie();
    Caroserie(const Caroserie& other);
    Caroserie& operator=(const Caroserie& other);

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

Fișierul `masina.h`

```h
#pragma once

#include "motor.h"
#include "caroserie.h"

class Masina : public Motor, public Caroserie {
private:
    char *model;

public:
    Masina(const char* model_init = "N/A",
           int cp_init = 0, const char* combustibil_init = "N/A",
           const char* culoare_init = "N/A", int usi_init = 0);
    ~Masina();
    Masina(const Masina& other);
    Masina& operator=(const Masina& other);

    // Functie friend pentru a putea accesa membrii protejati la sortare
    friend void sorteazaMasini(Masina* masini, int n);

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

Fișierul `garaj.h`

```h
#pragma once

#include "masina.h"

class Garaj {
private:
    Masina *masini;
    int nr_masini;
    char *nume_garaj;
public:
    Garaj(Masina* masini_init = nullptr, int nr_masini_init = 0, const char*
    ↪  nume_init = "N/A");
    ~Garaj();
    Garaj(const Garaj& other);
    Garaj& operator=(const Garaj& other);

    void schimbaNumeGaraj(const char* nume_nou);
    void adaugaMasina(const Masina& new_masina, int pozitie);

    friend std::ostream& operator<<(std::ostream& out, const Garaj& g);
};
```

Indicații pentru `main.cpp`

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

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

int main() {

    // 1. Creati dinamic un vector de cel putin 10 masini.
    // Initializati fiecare masina cu date relevante (model, cp, culoare etc.).
    // Pentru simplitate, hardcodati datele, NU CITITI DE LA TASTATURA.

    // 2. Creati un obiect de tip Garaj folosind vectorul de masini.
    // Afisati starea initiala a garajului.

    // 3. Sortati vectorul de masini din garaj in functie de caii putere.
    // Puteti implementa o functie globala sau o metoda a clasei Garaj pentru
    ↪  aceasta.
    // Afisati garajul dupa sortare pentru a verifica corectitudinea.

    // 4. Schimbati numele garajului.
    // Afisati din nou garajul pentru a vedea modificarea.

    // 5. Adaugati o noua masina pe o pozitie data (ex: pozitia 3).
    // Vectorul de masini se va mari.
    // Afisati starea finala a garajului.

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

    return 0;
}
```

Explicații suplimentare

- **Moștenire Multiplă** — Clasa `Masina` moștenește membrii publici și protejați din **ambele** clase de bază: `Motor` și `Caroserie`. Acest lucru permite unui obiect `Masina` să aibă atât caracteristici legate de performanță, cât și de aspect.
- **Agregare** — Clasa `Garaj` nu moștenește, ci **conține** un vector de obiecte de tip `Masina`. Aceasta este o relație de tip "has-a" (un garaj "are" mașini). Clasa `Garaj` este responsabilă pentru managementul ciclului de viață al vectorului de mașini.
- **Regula celor Trei/Cinci** — Deoarece toate clasele gestionează memorie alocată dinamic (`char*`, `Masina*`), 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 `Masina` trebuie să apeleze explicit constructorii ambelor clase de bază pentru a inițializa corect toți membrii moșteniți.

```
Masina::Masina(...) : Motor(cp_init, combustibil_init),
↪  Caroserie(culoare_init, usi_init) {
    // ... initializare membru propriu (model)
}
```

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

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

Cerințe (9p + 1p oficiu)

1. **Implementare Clase de Bază (2p)**
   - (1p) Implementarea corectă (Regula celor Trei, operatori stream) pentru clasa `Motor`.
   - (1p) Implementarea corectă (Regula celor Trei, operatori stream) pentru clasa `Caroserie`.
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 `Masina`.
   - (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 `Garaj`, gestionând vectorul de mașini.
   - (1p) Implementarea metodelor specifice: schimbarea numelui garajului, adăugarea unei noi mașini.
4. **Testare în main.cpp (2p)**
   - (1p) Crearea unui obiect `Garaj` cu cel puțin 10 mașini și testarea afișării și modificării numelui.
   - (1p) Sortarea vectorului de mașini și adăugarea unei noi mașini pe o poziție dată.
5. **(1p) Punct din oficiu.**

Compilare și verificare

```
g++ -g main.cpp motor.cpp caroserie.cpp masina.cpp garaj.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 (`motor.h`, `motor.cpp`, `caroserie.h`, `caroserie.cpp`, etc.), urmează comenzile:

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