Laborator 4 – teorie și exemple practice

Anul Universitar 2023 – 2024, Semestrul II

Programare Orientată pe Obiecte

LABORATOR 4, GRUPA 133

CUPRINS

Autor: Wagner Ștefan Daniel

I. Observație generală valabilă pe parcursul semestrului la laboratoarele de POO

Este alegerea voastră dacă veți implementa într-un singur fișier totul (la colocviu de acord, din cauza constrângerilor de timp), însă vă recomand o structură de fișiere cât mai clară, ***.cpp*** (dacă este necesar) și ***.h*** pentru fiecare ***class***/***struct***/**enum**/***union***.

Ulterior, vom discuta și împărțirea pe foldere **src/** și **include/**, cum este standardul în producție și în proiectele open source. Posibil vom ilustra și crearea unei librării statice pentru a justifica mai bine motivul împărțirii în fișiere header și sursă.

II. Sintaxa nouă cu {} pentru inițializare

Inițializarea cu **{}** este o sintaxă introdusă în standardul **C++11**, cunoscută și sub numele de inițializare uniformă sau inițializare directă, care uniformizează și simplifică inițializarea obiectelor.

```
class Vector2D {
private:
    float x;
    float y;
public:
    // Constructor cu initializare uniforma
    Vector2D(float x, float y) : x{x}, y{y} {}
};
Vector2D punct{3.5f, 2.5f}; // Utilizarea initializarii cu {}
```

Această metodă de inițializare previne anumite erori comune, cum ar fi inițializarea ambiguă sau inițializarea incompletă, și este preferată pentru claritate și consistență.

III. Utilizăm *private/protected* pentru metode interne

Este o practică bună în OOP să restricționăm accesul la metodele care sunt menite să fie utilizate doar în interiorul clasei, marcându-le ca ***private*** sau ***protected*** (dacă vrem să fie expusă în subclase – vom vedea la moștenire). Acest lucru îmbunătățește încapsularea.

Utilizatorul clasei nu are niciun motiv să știe de existența metodelor interne, iar dacă acestea ar fi publice există riscul să le apeleze din greșeală crezând că sunt parte din API-ul public și să ”strice” datele interne ale obiectului.

Exemplu simplificat de metodă expusă public care apelează o metodă internă:

```
#include <iostream>
class Calculator {
private:
    int Aduna(int a, int b) { return a + b; } // Metoda interna
public:
    void AfiseazaSuma(int a, int b) // Metoda publica apelata de utilizator
    {
        std::cout << "Suma este: " << Aduna(a, b) << std::endl;
    }
};
```

IV. Cuvântul cheie *static* în diferite contexte

Cuvântul cheie ***static*** poate fi utilizat în diverse contexte în C++, fiecare având implicații diferite asupra stocării și vizibilității. Aveți mai jos ilustrate sensurile semantice ale cuvântului cheie ***static*** în toate contextele:

1. Metode statice în clase

Metodele statice din cadrul unei clase permit utilizatorului să le apeleze fără a avea un obiect de tipul acelei clase. Metodele statice pot acționa numai aupra datelor membre statice și asupra parametrilor primiți. Însă, daca avem un obiect de tipul unei clase cu o metodă statica, putem sa apelăm metoda din interiorul obiectului. De asemenea, putem avea metode membre non-statice care acționează pe date statice.

Important: metodele statice nu conțin parametrul implicit ***this***, deoarece pot fi apelate fără a avea nevoie de un obiect de tipul clasei.

Exemplu simplificat:

```cpp
#include <iostream>
class Exemplu {
public:
    // definitie metoda statica
    static void Mesaj() {
        std::cout << "Aceasta este o functie statica." << std::endl;
    }
};

int main()
{
    Exemplu ex;
    // apel metoda statica din interiorul obiectului
    ex.Mesaj();
    // apel metoda statica fara obiect
    Exemplu::Mesaj();
    return 0;
}
```

2. Date membre statice în clase

În C++, **datele membre statice sunt comune tuturor instanțelor claselor**.

În exemplul de mai jos, variabila ***numarStudenti*** este comună tuturor obiectelor de tip ***Student***, și este de asemenea accesibilă fără existența unui obiect, precum este ilustrat în ***main()***.

O variabilă statică trebuie inițializată explicit în afara clasei (ideal în ***.cpp***), precum în exemplu.

```
#include <iostream>


class Student {
private:
    // Variabila statica comuna tuturor instantelor
    static int numarStudenti;
public:

    static void AdaugaStudenti(int numarStudenti_)
    {
       numarStudenti += numarStudenti_;
    }
    static int GetNumarStudenti(void)
    {
        return numarStudenti;
    }
};

int Student::numarStudenti = 0;

int main()
{
    // cu obiect
    Student student;
    int nrStudenti;
    student.AdaugaStudenti(nrStudenti);
    std::cout << student.GetNumarStudenti() << std::endl;

    // fara obiect
    Student::AdaugaStudenti(nrStudenti);
    std::cout << Student::GetNumarStudenti() << std::endl;

    return 0;
}
```

3. Variabile statice în funcții

În limbajele C/C++, variabilele statice declarate în cadrul funcțiilor se comportă precum variabilele globale (de fapt, ambele se află în aceeași zonă de memorie, segmentul de date, separat de stack și heap), însă scopul lor de vizibilitate este doar în cadrul acelei funcții.

Altfel zis, își mențin valoarea pe parcursul apelului funcțiilor, însă nu sunt vizibile în afara funcției in care au fost definite.

```cpp
#include <iostream>

void contor()
{
    // Valoarea lui cnt persista intre apeluri
    // cnt este initializat doar la primul apel
    // precum o variabila globala, dar vizibila doar
    // in interiorul undei functii.
    static int cnt = 0;
    cnt++;
    std::cout << "Contorul este la: " << cnt << std::endl;
}
```

4. Funcții/Variabile globale statice

Funcțiile și variabilele declarate cu cuvântul cheie static în scopul global sunt vizibile doar în cadrul fișierului **.cpp** curent / assembly-ului curent generat. Dacă sunteți curioși, mai multe informații (deși nu excelente) găsiți aici: https://www.geeksforgeeks.org/what-are-static-functions-in-c/

5. Redirecționarea input-ului din fișiere

Redirecționarea input-ului din fișiere este o tehnică utilă pentru testarea codului cu seturi predefinite de date de intrare. De asemenea, pe parcursul dezvoltării aplicației, nu veți fi nevoiți să introduceti manual datele de fiecare dată cand veti face o schimbare, și va fi mult mai puțin costisitor ca timp. **Obișnuiți-vă cu asta, util pentru colocviu!!**

Exemplu: Pentru a redirecționa input-ul dintr-un fișier numit **date.txt**, către programul vostru compilat numit **test.exe,** puteți folosi următoarea comandă în terminal:

```
C:\Daniel\ore\POO_Laborator\l04> test.exe < date.txt
```

6. Utilizarea containerelor STL (Standard Template Library)

Containerele din Standard Template Library (STL) sunt extrem de utile în C++ deoarece oferă o colecție de clase template gata de utilizat pentru a gestiona colecții de obiecte. Acestea sunt concepute pentru a fi eficiente și ușor de utilizat, reducând cantitatea de cod pe care trebuie să o scrieți și să o testați. Vom discuta în laboratoarele ulterioare detaliat despre template-uri.

Câteva containere importante: ***std::vector<T>, std::list<T>, std::set<T>, std::map<U,V>***

1. **Eficiență și Flexibilitate**: Containerele STL sunt optimizate pentru a oferi performanțe ridicate. De exemplu, **std::vector** oferă acces constant la elemente și redimensionare dinamică eficientă, ceea ce îl face o alegere excelentă pentru o listă de elemente care se modifică în timp.
2. **Reutilizare de Cod**: Folosind containerele STL, puteți evita reinventarea roții pentru structuri de date comune. De exemplu, în loc să implementați propria versiune de listă înlănțuită, puteți folosi **std::list**.
3. **Siguranță Tipului (Type Safety)**: Containerele STL sunt template-uri, ceea ce înseamnă că sunt create pentru a lucra cu orice tip de date, oferind siguranță la nivel de tipuri și evitând erorile comune legate de tipuri incompatibile.
4. **Operații Comune Standardizate**: Fiecare container STL vine cu un set de operații standard, cum ar fi inserarea, ștergerea și accesul la elemente. Acest lucru standardizează modul în care lucrăm cu structuri de date diferite, făcând codul mai ușor de înțeles și de menținut.
5. **Iteratori**: STL oferă iteratori, care permit parcurgerea elementelor unui container într-un mod generic, fără a fi nevoie să cunoașteți detaliile interne ale containerului. Aceasta facilitează scrierea de algoritmi generici care funcționează cu orice tip de container STL.
6. **Compatibilitate cu Algoritmii STL**: Containerele STL pot fi utilizate direct cu algoritmii STL, ceea ce vă permite să efectuați operații complexe, cum ar fi sortarea și căutarea, folosind o singură linie de cod.
7. **Gestionarea Memoriei**: STL se ocupă automat de alocarea și dealocarea memoriei pentru elementele containerelor, reducând riscul de memory leaks și alte erori legate de gestionarea memoriei, și de asemenea elimină necesitatea implementării cc/op=/destructor.

7. Exercițiu lucrat în cadrul laboratorului

Clasa Student

Date membre:

static int idGenerator;
int id;
std::string nume;
int nota;

Metode:

constructor
getters, getters (la nevoie)
cc/op=/destructor ilustrativ (nu este nevoie aici deoarece folosim fie date primitive, fie date membre cu proprii lor cc/op=/destructor, deci cei ***default*** sunt buni.

Clasa Curs

Date membre:

std::vector<Student> studenti;
std::string numeCurs;

Metode:

constructor
op<< pentru afișare
cc/op=/destructor cu mențiuni idem clasei **Student**
getters/setters la nevoie
înscrierea unui student la un curs
returnarea prin copie vectorului de studenti promovați la acel curs
+ ce decidem să mai facem împreună

Fișierul *main.cpp*

std::list<Curs> cursuri
creare cursuri
înscriere studenți la cursuri
afișarea tuturor studenților inscriși la un anumit curs
+ ce decidem să mai facem împreună

8. Important de reținut din laboratorul curent

- compunere și relația între obiecte
- cuvântul cheie ***static*** în cadrul claselor (dar și in alte contexte)
- prelucrarea colecțiilor de obiecte din librăria standard C++ (**STL – Standard Template Library**)

9. Exercițiu individual

De implementat clasa ***Biblioteca*** care conține o colecție de cărți de tipul ***Carte*** și permite operațiuni precum adăugarea sau împrumutul unei cărți, printre altele. Toate funcționalitățile le veți testa în ***main.cpp***.

Structura claselor:

Carte.h

```
#ifndef CARTE_H_
#define CARTE_H_
#include <string>

class Carte {
private:
    std::string titlu;
    std::string autor;
    bool esteImprumutata;
public:
    Carte(const std::string& titlu, const std::string& autor);
    // regula celor 3 nu este necesara, deoarece toate tipurile
    // de date din clasa Carte sunt fie primitive, fie au ele
    // proprii destructori/cc/op=
    // la proiect, totusi, implementati-le chiar daca nu este nevoie
    // ca sa va obisnuiti cu ele
    void Imprumuta(void);
    void Returneaza(void);
    std::string GetTitlu() const;
    std::string GetAutor() const;
    bool GetEsteImprumutata() const;

};

#endif /* CARTE_H_ */
```

Biblioteca.h

```h
#ifndef BIBLIOTECA_H_
#define BIBLIOTECA_H_
#include "Carte.h"
#include <string>
#include <vector>

class Biblioteca {
private:
    std::vector<Carte> carti;
    std::string numeBiblioteca;
public:
    // Vedeti observatia din Carte.h - regula celor 3 - cc/op=/destructor
    Biblioteca(const std::string& numeBiblioteca);
    void AdaugaCarte(const Carte& carte);
    // Alte metode pentru gestionarea cartilor si interactiunea cu utilizatorii
};

#endif /* BIBLIOTECA_H_ */
```

main.cpp

```cpp
#include "Biblioteca.h"
// alte fisiere header necesare

int main()
{
    // aici testati functionalitatile precum am facut la laborator
    return 0;
}
```

Exemple de funcționalități și metode de adăugat:

1. ***CautaCarte(const std::string& titlu)***: Caută o carte după titlu și returnează **true** dacă există în bibliotecă.
2. ***ListaCartiDisponibile(void)***: Afișează toate cărțile care nu sunt împrumutate.
3. ***ImprumutaCarte(const std::string& titlu):*** Permite împrumutarea unei cărți dacă aceasta este disponibilă.
4. ***ReturneazaCarte(const std::string& titlu)***: Permite returnarea unei cărți împrumutate.
5. Orice altă funcționalitate cu sens gândită de voi