Laborator 4 – teorie și exemple practice

Anul Universitar 2023 – 2024, Semestrul II

Programare Orientată pe Obiecte

LABORATOR 4, GRUPA 133

CUPRINS

I. **Observație generală valabilă pe parcursul semestrului la laboratoarele de POO**
II. **Sintaxa nouă cu {} pentru inițializare**
III. **Utilizăm *private/protected* pentru metode interne**
IV. **Redirecționarea input-ului din fișiere**
V. **Utilizarea containerelor STL (Standard Template Library)**
VI. **Exercițiu lucrat în cadrul laboratorului**
VII. **Important de reținut din laboratorul curent**
VIII. **Exercițiu individual**

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***.

În cazul claselor mici, împărțirea pe fișiere poate mai mult să incurce. De asemenea, dacă aveți mai multe clase mici care au legătură între ele, puteți sa le puneți în același fisier. Consistența contează.

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, care 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.

Metodele interne sunt metode ”ajutătoare” pe care le folosim pentru a construi API-ul public al clasei, adică metodele pe care dorim sa le expunem utilizatorului clasei noastre.

Metodele la care utilizatorul ar trebui sa aiba acces (care sunt de fapt cele pentru care ne folosește clasa), rămân în continuare publice.

Utilizatorul clasei nu are niciun motiv să știe de existența metodelor interne (cele pe care le apelăm doar noi în cadrul clasei).

Mai mult, dacă metodele interne ar fi publice există riscul ca utilizatorul să le apeleze din greșeală crezând că sunt parte din API-ul public (cele pe care dorim să le expunem) și să ”strice” datele interne ale obiectului. De asemenea, nu mai există libertatea de a face orice modificare în partea publică a clasei fără să ”încalci contractul”.

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

```cpp
#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. 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
```

V. 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.

Motive pentru care vrem să folosim containere STL:

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. În producție, se pot defini custom allocators (**vedeți https://github.com/mcmarius/poo/blob/master/obs.md**).

Câteva containere importante pe care le veți utiliza: ***std::vector<T>, std::list<T>, std::set<T>, std::map<U,V>, std::stack<T>,*** etc**.**

O referință pentru restul container-elor disponibile în librăria standard (deși nu garantez că este cea mai bună referință): https://en.cppreference.com/w/cpp/container

Exercițiu lucrat în cadrul laboratorului

Clasa Student

Date membre:

int numarMatricol;
std::string nume;
int nota;
+ ce decidem să mai facem împreună

Metode:

constructor
getters, setters (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. Mai multe discuții despre =***default*** (și =***delete***), în laboratoarele urmatoare.

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

VI. Important de reținut din laboratorul curent

- compunere și relația între obiecte
- prelucrarea colecțiilor de obiecte din librăria standard C++ (**STL – Standard Template Library**)

VII. Exercițiu individual

De implementat clasa ***Biblioteca*** care conține ca atribut 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