Principiile SOLID. Expresii Lambda în C++

Anul Universitar 2023 – 2024, Semestrul II

Programare Orientată pe Obiecte

LABORATOR 11, GRUPA 133

Cuprins

Autor: Wagner Ștefan Daniel

I. Expresii Lambda în C++

1. Introducere la Expresii Lambda

Expresiile lambda, introduse în standardul **C++11**, reprezintă o funcție `anonimă` (definită in-place), care permite scrierea de cod concis și direct în locul unde este necesar.

Acestea sunt deosebit de utile pentru lucrul cu algoritmi din Standard Template Library (STL), permițând personalizarea comportamentului cu minim de sintaxă suplimentară.

2. Sintaxa și Utilizările Expresiilor Lambda

O expresie lambda este definită prin următoarea sintaxă:

```
[captures](parameters) -> returnType {
    // Corpul functiei/expresiei lambda
}
```

- **captures**: controlează ce variabile din scopul înconjurător sunt disponibile în lambda și cum (prin valoare sau referință).
- **parameters**: lista de parametri acceptați de lambda, similar unei funcții normale.
- **returnType**: tipul returnat, care este adesea dedus automat de compilator.
- **body**: corpul lambda, unde este definită logica.

3. Exemplu de Utilizare a Expresiilor Lambda

Expresiile lambda sunt extrem de utile pentru a personaliza comportamentul funcțiilor din STL, cum ar fi **std::sort()** sau **std::for_each()**.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
    std::vector<int> numbers = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5};

    std::cout << "Original vector: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // Sortarea vectorului folosind o expresie lambda
    std::sort(numbers.begin(), numbers.end(),
        [](int a, int b) {
```

```
            return a < b;
        }
    );

    std::cout << "Sorted vector: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // Aplicarea unei operatiuni pe fiecare element
    std::for_each(numbers.begin(), numbers.end(), [](int& n) {
        n *= 2;
    });

    std::cout << "Doubled values: ";
    for (int num : numbers) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

4. Capturi în Lambda: by-value vs. by-reference

Expresiile lambda permit captarea variabilelor din scopul înconjurător fie prin valoare (**[=]**), fie prin referință (**[&]**). Captura prin valoare este sigură dacă lambda este utilizată într-un context unde variabilele capturate nu se modifică în exterior, în timp ce captura prin referință permite modificarea variabilelor capturate.

```
##include <iostream>

int main()
{
    int x = 10;
    auto byValue = [x]() mutable {
        x += 5;
        std::cout << "Inside byValue lambda: " << x << std::endl;
    };
    auto byReference = [&x]() {
        x += 5;
        std::cout << "Inside byReference lambda: " << x << std::endl;
    };
```

```
    // x nu este modificat in afara lambda
    byValue();
    std::cout << "After byValue, x: " << x << std::endl;

    // x este modificat in afara lambda
    byReference();
    std::cout << "After byReference, x: " << x << std::endl;
    return 0;
}
```

5. Expresii Lambda vs Pointeri la Funcții (Function Pointers)

5.1. Pointeri la Funcții (Function Pointers)

Pointerii la funcții sunt metroda tradițională a limbajelor C și C++ care permit transmiterea directă a funcțiilor pentru a fi utilizate ca variabile, deoarece numele unei funcții este o eticheta în memorie (o adresă oarecare care conține zonă de cod executabil), la care procesorul face call/jmp. Acești pointeri sunt instrumente eficiente în design-ul software unde funcțiile pot fi pasate ca argumente fără a necesita un context adițional.

5.2. Expresii Lambda Comparativ cu Pointerii la Funcții

Expresiile lambda permit definirea rapidă și inline a funcțiilor anonime, care pot capta variabile din contextul înconjurător, ceea ce mecanismul de pointeri la funcții moștenit din C nu permite.

Expresiile lambda care nu captează variabile din contextul lor pot fi convertite în pointeri la funcții. Aceasta permite utilizarea lor în contexte tradiționale C care necesită pointeri la funcții. Totuși, o dată ce o expresie lambda captează una sau mai multe variabile, conversia în pointer la funcție devine imposibilă. Acest lucru se datorează faptului că starea capturată trebuie gestionată, iar pointerii la funcții standard nu au capacitatea de a stoca sau accesa această stare.

Pentru cei curioși, stilul C pentru `capturi` este emulat prin trimiterea un pointer către o funcție care așteaptă argumente, precum și acele argumente (capturile), ca parametri către funcția care le va apela. Cel mai generic argument pe care o funcție îl poate primi în C are tipul ***void**** (pointer către orice, o adresă de memorie oarecare), și este responsabilitatea programatorului să facă cast-ul corect la tipul de date așteptat (poate fi chiar și un pointer la o funcție, o structură, sau pointer către un singur int).

Această capacitate de captură le conferă o putere și versatilitate semnificativ mai mare comparativ cu pointerii la funcții

Deși nu voi include exemplul în cadrul laboratorului deoarece este exemplu pur C, vom ilustra utilizarea librăriei POSIX ***pthread.h*** drept exemplu în cadul laboratorului de utilizare a pointerilor la funcții, dar și o mini-introducere în multi-threading. Un exemplu alternativ ar fi să folosim librăria **libcurl** pentru a prelua date dinamic de pe internet, alegem împreună ce exemplu facem 😊.

6. Concluzii finale

Expresiile lambda oferă o modalitate extrem de flexibilă și puternică de a integra **functional programming în C++**, facilitând lucrul cu colecții de date și alte structuri care necesită procesare iterativă sau condiționată.

II. Principiile SOLID

1. Introducere la Principiile SOLID

Principiile SOLID sunt un set de cinci principii de design în programarea orientată pe obiecte, formulate pentru a promova un software mai înțelegibil, flexibil și întreținut.

Respectarea acestor principii poate preveni probleme comune de design, cum ar fi codul greu de înțeles și de modificat.

Respectarea acestor principii, cel mai important (în producție), duce la scalabilitatea codului mai ușoară, și o consistență de stil de scriere și modularizare a codului, într-un limbaj universal.

2. Single Responsibility Principle (SRP) - Principiul Responsabilității Unice

Principiul responsabilității unice afirmă că o clasă ar trebui să aibă un singur motiv pentru a se schimba, adică ar trebui să aibă doar o singură responsabilitate.

Exemplu de SRP:

```
#include <iostream>
#include <fstream>
#include <memory>

// Interfata pentru logging, conform Dependency Inversion Principle (DIP)
// (Avem un exemplu ulterior doar cu DIP)
```

```cpp
class ILogger {
public:
    virtual void Log(const std::string& message) = 0;
    virtual ~ILogger() = default;
};

// Logger concret care scrie in consola
class ConsoleLogger : public ILogger {
public:
    void Log(const std::string& message) override {
        std::cout << "Log: " << message << std::endl;
    }
};

// Logger concret care scrie intr-un fisier
class FileLogger : public ILogger {
    std::ofstream logFile;
public:
    FileLogger(const std::string& filename) {
        logFile = std::ofstream(filename);
        if (!logFile) {
            throw std::runtime_error("Nu s-a putut deschide fisierul de log");
        }
    }

    void Log(const std::string& message) override {
        if (logFile.is_open()) {
            logFile << "Log: " << message << std::endl;
        }
    }

    ~FileLogger() {
        if (logFile.is_open()) {
            logFile.close();
        }
    }
};

// Clasa care gestioneaza procesele de business logic ale utilizatorilor
class UserProcessor {
    std::shared_ptr<ILogger> logger;
public:
    // Constructor cu injectie de dependenta
    UserProcessor(std::shared_ptr<ILogger> logger) : logger(std::move(logger))
{}
```

```
    void AddUser(const std::string& userName) {
        // Aici cod pentru logica de adaugare a unui utilizator
        logger->Log("User added: " + userName);
    }
};

int main()
{
    // Creaza un logger pentru consola si un procesor de utilizatori
    auto consoleLogger = std::make_shared<ConsoleLogger>();
    UserProcessor userProcessor(consoleLogger);
    userProcessor.AddUser("Marian");

    // Creaza un logger pentru fisier si un procesor de utilizatori
    auto fileLogger = std::make_shared<FileLogger>("log.txt");
    UserProcessor userProcessorFile(fileLogger);
    userProcessorFile.AddUser("Ionut");

    return 0;
}
```

3. Open/Closed Principle (OCP) - Principiul Deschis/Închis

OCP afirmă că software-ul ar trebui să fie deschis pentru extensie, dar închis pentru modificare. Acest lucru înseamnă că trebuie să putem adăuga funcționalități noi fără a modifica codul existent.

Exemplu de SRP:

```
#include <iostream>
#include <vector>

class Shape {
public:
    // Metoda pur virtuala, obliga clasele derivate sa o implementeze
    virtual double Area() const = 0;
};
class Rectangle : public Shape {
private:
    double width, height;
public:
    // Constructor pentru initializarea latimii si inaltimii
    Rectangle(double w, double h) : width(w), height(h) {}
```

```
    // Suprascrierea metodei Area specifica pentru dreptunghi
    double Area() const override {
        return width * height;
    }
};
class Circle : public Shape {
private:
    double radius;
public:
    // Constructor pentru initializarea razei
    Circle(double r) : radius(r) {}
    // Suprascrierea metodei Area specifica pentru cerc
    double Area() const override {
        const double PI = 3.1415926;
        return PI * radius * radius;
    }
};

void PrintTotalArea(const std::vector<Shape*>& shapes) {
    double totalArea = 0;
    for (const auto& shape : shapes) {
        // Calculul ariei totale prin adunarea ariilor individuale
        totalArea += shape->Area();
    }
    // Afisarea ariei totale
    std::cout << "Total Area: " << totalArea << std::endl;
}

int main() {
    // Vector pentru stocarea pointerilor catre obiecte de tip Shape
    std::vector<Shape*> shapes;
    // Crearea si adaugarea formelor in vector
    shapes.push_back(new Rectangle(10, 20));
    shapes.push_back(new Circle(10));
    // Afisarea ariei totale a formelor stocate in vector
    PrintTotalArea(shapes);
    // Eliberarea memoriei
    for (auto& shape : shapes) {
        // Eliberarea memoriei pentru fiecare forma
        delete shape;
    }
    // Redimensionarea vectorului la zero elemente
    shapes.clear();
    return 0;
}
```

4. Liskov Substitution Principle (LSP) - Principiul Substituției Liskov

LSP susține că obiectele unei clase derivate trebuie să poată înlocui obiectele clasei de bază fără a afecta corectitudinea aplicației.

Exemplu de LSP:

```
// Exemplul anterior Open Closed Principle cu Shape, Rectangle si Circle
// respecta Liskov Substitution Principle
```

5. Interface Segregation Principle (ISP) - Principiul Separării Interfeței

ISP afirmă că utilizatorii/clienții nu ar trebui să fie forțați să depindă de interfețe pe care nu le utilizează. Acest principiu înseamna împărțitul interfețelor în bucăți cât mai mici, astfel încât nicio subclasă sa nu fie nevoită să implementeze metode care nu au sens/pe care nu le va uitiliza.

Exemplu de ISP:

```
#include <iostream>

class IPrinter {
public:
    virtual void PrintDocument() = 0;
};

class IScanner {
public:
    virtual void ScanDocument() = 0;
};


class SimplePrinter : public IPrinter {
public:
    void PrintDocument() override {
        std::cout << "Printing Document with Simple Printer..." << std::endl;
    }
};

class SimpleScanner : public IScanner {
public:
    void ScanDocument() override {
        std::cout << "Scanning Document with Simple Scanner..." << std::endl;
    }
};
```

```
class MultiFunctionMachine : public IPrinter, public IScanner {
public:
    void PrintDocument() override {
        std::cout << "Printing Document with Multi Functional Machine...\n";
    }
    void ScanDocument() override {
        std::cout << "Scanning Document with Multi Functional Machine...\n";
    }
};

int main()
{
    // Crearea obiectelor pentru fiecare dispozitiv
    SimplePrinter simplePrinter;
    SimpleScanner simpleScanner;
    MultiFunctionMachine multiFunctionalMachine;

    // Demonstrarea principiului Interface Segregation (ISP)
    std::cout << "Using Simple Devices:" << std::endl;
    simplePrinter.PrintDocument();  // Utilizarea imprimantei simple
    simpleScanner.ScanDocument();   // Utilizarea scanerului simplu

    std::cout << "\nUsing MultiFunctional Device:" << std::endl;
    // Utilizarea dispozitivului multifunctional pentru imprimare
    multiFunctionalMachine.PrintDocument();
    // Utilizarea dispozitivului multifunctional pentru scanare
    multiFunctionalMachine.ScanDocument();

    return 0;
}
```

6. Dependency Inversion Principle (DIP) - Principiul Inversării Dependențelor

6.1. Definiție

Principiul Inversării Dependențelor este ultimul dintre cele cinci principii fundamentale ale designului orientat pe obiect (SOLID). DIP subliniază că:

- **Modulele de nivel înalt** nu ar trebui să depindă de modulele de nivel jos. În schimb, ambele ar trebui să depindă de abstracții.
- **Abstracțiile** (interfețe sau clase abstracte) nu ar trebui să depindă de detaliile de implementare ale modulelor de nivel jos, ci doar de definițiile lor abstracte.

6.2. Scopul DIP

Scopul principal al DIP este de a reduce dependențele dintre componentele software, ceea ce duce la un cod mai ușor de gestionat și de modificat. Acest principiu contribuie la realizarea unui sistem mai flexibil și mai modular.

- **Loose coupling - ”dependență scăzută între module”:**

  Implementarea DIP ajută la menținerea unui **loose copuling** între module, permitând fiecărui modul să își îndeplinească funcțiile independent de implementările specifice ale altor module. În cazul loose copuling, componentele/modulele sunt cât mai puțin dependente unul de celălalt, astfel asigurând și **separation of concerns**.

- **Decuplare:**

  DIP favorizează decuplarea componentelor, ceea ce înseamnă că modificările în implementarea unui modul nu ar trebui să afecteze modulele care depind de acesta, atâta timp cât contractele abstracțiilor sunt menținute.

6.3. Exemplu de DIP

Considerăm un sistem software pentru gestionarea plăților într-un magazin online:

- **Clasa de nivel înalt: PaymentProcessor**, care utilizează diverse metode de plată.
- **Clasa de nivel jos: CreditCardPayment**, **PayPalPayment**, etc., fiecare implementând o interfață: **IPaymentMethod**

Fără DIP:

```cpp
#include <iostream>

// Definirea unei clase concrete pentru procesarea platilor cu cardul de credit
class CreditCardPayment {
public:
    void Process(double amount) {
        std::cout << "Processing credit card payment for amount: "
                  << amount << " RON." << std::endl;
    }
};

// Clasa PaymentProcessor care utilizeaza direct un obiect CreditCardPayment
class PaymentProcessor {
    CreditCardPayment creditCardPayment;  // Crearea unei instante a clasei
CreditCardPayment
```

```
public:
    void ProcessPayment(double amount) {
        // Delegarea apelului la metoda Process a clasei CreditCardPayment
        creditCardPayment.Process(amount);
    }
};

int main()
{
    // Crearea unui obiect PaymentProcessor
    PaymentProcessor processor;

    // Procesarea unei plati de 250 RON
    processor.ProcessPayment(250.0);

    return 0;
}
```

În cazul de mai sus, **PaymentProcessor** depinde direct de implementarea **CreditCardPayment**.

Varianta Alternativă cu DIP:

```
#include <iostream>

class IPaymentMethod {
public:
    virtual void Process(double amount) = 0;
};

class CreditCardPayment : public IPaymentMethod {
public:
    void Process(double amount) override {
        // Logica specifica pentru plata cu cardul de credit
        std::cout << "Processing credit card payment for amount: " << amount <<
std::endl;
    }
};

// Alte clase de payment diferite CashPayment, PayPalPayment, etc.

// Implementare concreta sistem de procesare de tranzactii
// Nu conteaza tipul clasei concrete derivate de payment,
// deoarece toate implementeaza o interfata comuna :-)
class PaymentProcessor {
private:
```

```
    IPaymentMethod *paymentMethod;
public:
    PaymentProcessor(IPaymentMethod *method) : paymentMethod(method) {}

    void ProcessPayment(double amount) {
        paymentMethod->Process(amount);
    }
};

int main()
{
    // Implementare concreta a IPaymentMethod pentru credit cards
    CreditCardPayment creditCardPayment;
    PaymentProcessor paymentProcessor(&creditCardPayment);
    // Processing payment-ului folosind metoda de payment injectata cu DI
    paymentProcessor.ProcessPayment(100.0);
    return 0;
}
```

6.4. Exemplu DIP + Dependency Injection:

```
#include <iostream>

// Interfata abstracta pentru accesul la date de orice tip
// Consola, BD, Cloud, Fisier, FTP, SSH, etc...
class IDataAccess {
public:
    // Metoda pur virtuala pentru citirea datelor de orice tip.
    virtual void ReadData() = 0;
};

// Implementare concreta a interfetei IDataAccess pentru accesul la baza de
date.
class DataAccess : public IDataAccess {
public:
    void ReadData() override {
        std::cout << "Reading data from database." << std::endl;
    }
};

// Implementare concreta a interfetei IDataAccess pentru accesul la fisiere.
class FileDataAccess : public IDataAccess {
public:
    void ReadData() override {
        std::cout << "Reading data from a file." << std::endl;
    }
```

```
};

// Clasa BusinessLogic foloseste o instanta de IDataAccess pentru a procesa
date.
class BusinessLogic {
private:
    // Pointer catre interfata IDataAccess pentru dependinta.
    IDataAccess* dataAccess;
public:
    // Constructor care primeste un pointer la interfata IDataAccess.
    BusinessLogic(IDataAccess* da) : dataAccess(da) {}

    // Metoda ce foloseste serviciul IDataAccess pentru a procesa datele.
    void ProcessData() {
        dataAccess->ReadData();
    }
};

int main()
{
    // Implementare concreta a interfetei IDataAccess in doua feluri

    // Crearea unei instante a clasei concrete DataAccess.
    DataAccess dataAccess;
    // Crearea unei instante a clasei concrete FileDataAccess.
    FileDataAccess fileDataAccess;

    // Dependency Injection in clasa de BusinessLogic a clasei DataAccess
    // Se va inregistra ca IDataAccess
    BusinessLogic businessLogicDB(&dataAccess);
    businessLogicDB.ProcessData();  // Afiseaza: Reading data from database.

    // Dependency Injection in clasa de BusinessLogic a clasei FileDataAccess
    // Se va inregistra ca IDataAccess
    BusinessLogic businessLogicFile(&fileDataAccess);
    businessLogicFile.ProcessData();  // Afiseaza: Reading data from a file.

    return 0;
}
```

7. Concluzii finale SOLID

Aceste principii sunt esențiale pentru construirea unui software robust și ușor de întreținut. Implementarea lor corespunzătoare poate reduce semnificativ complexitatea

aplicațiilor, facilitând în același timp testarea și extinderea, lucruri foarte importante in design-ul software.

III. Aplicații Practice și Exemple de Cod

1. Introducere

Această secțiune își propune să demonstreze aplicarea practică a conceptelor discutate anterior prin intermediul unor exemple concrete de cod. Aceste exemple ilustrează cum principiile design-ului, expresiile lambda și pattern-urile de design pot fi utilizate pentru a rezolva probleme specifice în dezvoltarea software în producție.

2. Exemplu de Aplicație - SOLID + Design Patterns

Vom crea o aplicație simplă care folosește mai multe pattern-uri de design pentru a demonstra modularitatea și reutilizabilitatea codului, dar să respecte și toate principiile SOLID.

Scenariu: Aplicație pentru Gestionarea Comenzilor într-un Restaurant

```
#include <iostream>
#include <vector>
#include <memory>

// Aplicatia respecta toate principiile SOLID

// Interfata ICommand - Command Pattern
class ICommand {
public:
    virtual void Execute() = 0;
    // Teoretic =0 dar nu fortez subclasele sa implementeze destructor daca nu
au nevoie
    virtual ~ICommand() {}
};

// Comanda concreta
class Order {
public:
    void PrepareOrder(const std::string& orderDetails) {
        std::cout << "Preparing order: " << orderDetails << std::endl;
    }
};

class OrderCommand : public ICommand {
```

```
private:
    Order *order;
    std::string orderDetails;
public:
    OrderCommand(Order *order, const std::string& details) : order(order),
orderDetails(details) {}
    void Execute() override {
        order->PrepareOrder(orderDetails);
    }
};

// Command Factory
class CommandFactory {
public:
    static std::unique_ptr<ICommand> CreateCommand(Order *order, const
std::string& details) {
        return std::make_unique<OrderCommand>(order, details);
    }
};

// Observer Pattern
class Kitchen {
private:
    std::vector<std::unique_ptr<ICommand>> commands;
public:
    void AddOrder(ICommand *command) {
        commands.emplace_back(command);
    }
    void Notify() {
        for (auto& command : commands) {
            command->Execute();
        }
        commands.clear();
    }
};

// Functia main(), ilustrand Dependency Inversion si Factory Pattern
int main()
{
    Kitchen kitchen;
    Order order;

    auto breakfastOrder = CommandFactory::CreateCommand(&order, "Breakfast");
    auto lunchOrder = CommandFactory::CreateCommand(&order, "Lunch");
```

```
    kitchen.AddOrder(breakfastOrder.get());
    kitchen.AddOrder(lunchOrder.get());

    kitchen.Notify();

    return 0;
}
```

Aplicația ilustrează utilizarea **design pattern-ului Command** pentru gestionarea comenzilor într-un context de bucătărie, demonstrând cum design patterns pot facilita structurarea și organizarea codului conform principiilor **SOLID**. Analiza codului relevă următoarele aspecte notabile în concordanță cu fiecare principiu SOLID:

Principiul Responsabilității Unice (Single Responsibility Principle - SRP):

- **Clasa Order** este responsabilă exclusiv pentru pregătirea comenzilor, concentrându-se pe o singură funcționalitate în sistem, ceea ce este o manifestare clară a SRP.
- **Clasa OrderCommand** servește ca un intermediar între comanda și execuția acesteia, încapsulând atât detaliile comenzii cât și obiectul **Order** care execută comanda. Aceasta separă logic logistica comenzi de implementarea ei, aliniindu-se cu SRP prin segregarea responsabilităților.

Principiul Deschis/Închis (Open/Closed Principle - OCP):

- Sistemul este deschis pentru extensie dar închis pentru modificare. Implementarea **CommandFactory** permite adăugarea de noi tipuri de comenzi fără a modifica codul existent. Aceasta demonstrează flexibilitatea OCP, facilitând extensia funcționalităților fără a afecta componentele stabilite.

Principiul Substituției Liskov (Liskov Substitution Principle - LSP):

- **Interfața ICommand** stabilește un contract cu metoda **Execute()**, care trebuie respectat de toate clasele derivate, cum ar fi **OrderCommand**. Această utilizare a polimorfismului asigură că substituirile de subclase sunt realizate fără a afecta comportamentul sistemului, îndeplinind LSP.

Principiul Separării Interfețelor (Interface Segregation Principle - ISP):

- **ICommand** este o interfață minimală, care expune doar metoda **Execute()**. Acest design previne necesitatea ca clasele implementatoare să suporte metode irelevante, ceea ce corelează direct cu ISP.

Principiul Inversiunii Dependențelor (Dependency Inversion Principle - DIP):

- Componentele de nivel superior, precum **Kitchen**, nu depind de implementările specifice ale comenzilor, ci de abstracțiuni, respectiv interfața **ICommand**. Aceasta reduce cuplajul și crește modularitatea, demonstrând aplicarea DIP.

Utilizarea std::unique_ptr pentru gestionarea instanțelor de **ICommand** ilustrează good practices în C++ modern, ajutând la prevenirea memory leaks și la gestionarea clară a ownership-ului resurselor, ceea ce susține indirect principiile SOLID.

Funcția principală (functia main()) demonstrează Dependency Injection prin utilizarea fabricii pentru crearea obiectelor, care decuplează crearea obiectelor specifice de logicile de utilizare ale acestora. Aceasta ilustrează un nivel avansat de aderență la DIP, facilitând testarea și extensibilitatea.

3. Utilizarea Expresiilor Lambda în STL

Expresiile lambda pot simplifica manipularea colecțiilor de date. Acest exemplu demonstrează utilizarea lor în algoritmi STL pentru a filtra și procesa colecții.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main()
{
    std::vector<int> numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    std::cout << "Original numbers:";
    for (int n : numbers) {
        std::cout << " " << n;
    }
    std::cout << std::endl;
```

```
    // Eliminarea numerelor impare folosind remove_if și lambda
    numbers.erase(std::remove_if(numbers.begin(), numbers.end(),
        [](int x) { return x % 2 != 0; }), numbers.end());

    std::cout << "Even numbers:";
    for (int n : numbers) {
        std::cout << " " << n;
    }
    std::cout << std::endl;

    return 0;
}
```

4. Concluzii Finale Laborator

Exemplele prezentate demonstrează puterea și flexibilitatea design-ului orientat pe obiecte și practicile moderne de programare în C++. Integrând principiile SOLID, expresiile lambda și pattern-urile de design, dezvoltatorii pot crea aplicații mai robuste, mai flexibile și mai ușor de întreținut.

IV. Concluzii la Final de Semestru

Vă mulțumesc tuturor pentru timpul și interesul acordat de-a lungul semestrului!

Împreuna am trecut prin cât de multe concepte OOP C++ am reușit să acoperim (destul de multe, și este un lucru foarte bun).

O să vedeți pe viitor cum vă veți lovi de multe concepte/lucruri învățate în cadrul laboratoarelor, și o sa știti cel puțin de unde sa le apucați, deci munca voastră în mod cert va da roade.

Succes în continuare tuturor în tot ceea ce faceți! 😊