Universitatea Tehnică a Moldovei

Facultatea *Calculatoare, Informatică și Microelectronică*

Specialitatea *Tehnologii Informaționale*

Raport

la lucrarea de laborator nr. 2

Tema:*"Supraîncărcarea operatorilor"*

Disciplina: "Programarea orientata pe obiecte"

Varianta 3

A efectuat:	Student grupa TI-231 FR	Apareci Aurica
A verificat:	Asistent universitar	Mititelu Vitalie

Chișinău 2025

Cuprins

1. Cadru teoretic

Tema lucrării: Supraîncărcarea operatorilor

Scopul lucrării:

- Studierea necesității supraîncărcării operatorilor;
- Studierea sintaxei de definire a operatorilor;
- Studierea tipurilor de operatori;
- Studierea formelor de supraîncărcare;

Varianta 3: Să se creeze clasa Vector – vector de tip long, utilizînd memoria dinamică. Să se definească operatorii "+" – adunarea element cu element a vectorilor, "-" – scăderea element cu element a vectorilor, ca funcții prietene. Să se definească "=" – operator de atribuire şi operatorii de comparare: "==", "!=", "<", ">" ca metode ale clasei. Pentru realizarea ultimilor doi operatori să se definească funcția de calcul a modulului elementelor vectorului. Să se supraîncarce operatorii "<<" şi ">>" pentru ieşiri/intrări de obiecte. Clasa trebuie să conțină toți constructorii necesari şi destructorul.

2. Repere teoretice

Supraîncărcarea operatorilor constituie un mecanism specific limbajului C++ care permite redefinirea comportamentului operatorilor prestabiliți, astfel încât aceştia să poată fi utilizați în mod natural şi coerent cu tipurile de date definite de utilizator, în special în contextul claselor. Necesitatea acestei funcționalități decurge din dorința de a extinde expresivitatea limbajului şi de a asigura o interacțiune intuitivă între obiecte, conferindu-le acestora comportamente similare tipurilor fundamentale. Astfel, operatori precum +, -, == sau [] pot fi adaptați pentru a reflecta logica specifică structurii de date implementate.

Supraîncărcarea se realizează prin utilizarea cuvântului cheie operator, urmat de simbolul operatorului dorit, în cadrul unei metode membru sau al unei funcții prietene. Alegerea formei depinde de modul în care se doreşte accesul la membrii interni ai clasei şi de semantica operației.

```
class Complex {
public:
    float re, im;
    Complex operator+(const Complex& other) {
        return Complex(re + other.re, im + other.im);
    }
};
```

Operatorii pot fi supraîncărcați prin două abordări fundamentale, în funcție de natura şi scopul operației dorite. Prima modalitate constă în definirea operatorilor ca metode membre ale clasei, situație în care operatorul acționează asupra instanței curente, reprezentată prin pointerul implicit this, şi cel mult asupra unui operand suplimentar. A doua modalitate implică definirea oper-

atorilor ca funcții prietene (friend) sau funcții globale, context în care ambii operanzi sunt furnizați explicit ca parametri ai funcției. Din perspectiva arității, operatorii se clasifică în *unari*, atunci când acționează asupra unui singur operand – cum este cazul operatorilor unari de negare (-a) sau de incrementare (++a) – și *binari*, atunci când implică doi operanzi, precum în expresiile de adunare (a + b) sau comparație (a == b).

Limbajul C++ oferă suport extins pentru supraîncărcarea operatorilor, însă această facilitate se aplică exclusiv operatorilor deja existenți în limbaj. Cu alte cuvinte, nu este posibilă definirea unor operatori noi, ci doar adaptarea comportamentului celor predefiniți pentru a interacționa în mod specific cu tipurile de date personalizate. Totuși, există anumite excepții importante: operatorii de rezoluție a domeniului (::), acces la membri (. și .*), operatorul ternar (?:), precum și operatorii de preprocesare (# și ##) nu pot fi supraîncărcați, întrucât aceștia sunt interpretați direct de compilator și nu permit personalizare. Pe de altă parte, limbajul permite supraîncărcarea unor operatori avansați, precum ->, [], (), precum și a operatorilor de alocare și dealocare dinamică (new, new[], delete, delete[]), oferind astfel un grad ridicat de flexibilitate în proiectarea claselor complexe.

3. Listingul programului

main.cpp

```
#include <iostream>
#include "MyVector.h"

int main()
{
    int size = 5;
    MyVector MyV(size);
    MyVector MyV2(size);
    cout << "Introduceti 5 elemente ale primului vector:\n";
    cin >> MyV;

    cout << "\nIntroduceti 5 elemente ale celui de-al doilea vector:\n";
    cin >> MyV2;
    system("cls");

    cout << "Primul vector este: ";
    cout << MyV << endl;
    cout << "Al doilea vector este: ";
    cout << MyV2 << endl;

    MyVector sum = MyV + MyV2;
    cout << "Suma celor doi vectori: " << sum << endl;

    MyVector diff = MyV - MyV2;
    cout << "Diferenta celor doi vectori: " << diff << endl;
    cout << "Modulul primului vector: " << MyV.Modulus() << endl;
    cout << "Modulul celui de-al doilea vector: " << MyV2.Modulus() << endl;

    cout << "Comparatii:\n";
    cout << "V1 == V2 -> " << (MyV == MyV2) << endl;
    cout << "V1 != V2 -> " << (MyV != MyV2) << endl;
    cout << "V1 < V2  -> " << (MyV < MyV2) << endl;
    cout << "V1 > V2  -> " << (MyV > MyV2) << endl;

    return 0;
```

MyVector.h

```
#pragma once

#include <iostream>
using namespace std;

class MyVector {
private:
    long* data;
    int size;

public:
    MyVector() : size(0), data(nullptr) {}

    MyVector(int s) : size(s) {
        data = new long[size];
    }

    MyVector(const MyVector& other) : size(other.size) {
        data = new long[size];
        for (int i = 0; i < size; ++i)
            data[i] = other.data[i];
    }

    ~MyVector() {
        delete[] data;
    }

    MyVector& operator=(const MyVector& other) {
        if (this != &other) {
            delete[] data;
            size = other.size;
            data = new long[size];
            for (int i = 0; i < size; ++i)
                data[i] = other.data[i];
        }
        return *this;
    }

    bool operator==(const MyVector& v) const {
        if (size != v.size) return false;
        for (int i = 0; i < size; ++i)
            if (data[i] != v.data[i]) return false;
        return true;
    }

    bool operator!=(const MyVector& v) const {
        return !(*this == v);
    }

    long Modulus() const {
        long sum = 0;
        for (int i = 0; i < size; ++i)
            sum += data[i] * data[i];
        return sqrt(sum);
    }

    bool operator<(const MyVector& v) const {
        return this->Modulus() < v.Modulus();
    }

    bool operator>(const MyVector& v) const {
        return this->Modulus() > v.Modulus();
    }
```

```
    void SetSize(int s) {
        delete[] data;
        size = s;
        data = new long[size];
    }

    int GetSize() const {
        return size;
    }

    void SetData(int index, long value) {
        if (index >= 0 && index < size)
            data[index] = value;
    }

    long GetData(int index) const {
        if (index >= 0 && index < size)
            return data[index];
        return 0;
    }

    friend MyVector operator+(const MyVector& a, const MyVector& b) {
        int minSize = min(a.size, b.size);
        MyVector result(minSize);
        for (int i = 0; i < minSize; ++i)
            result.data[i] = a.data[i] + b.data[i];
        return result;
    }

    friend MyVector operator-(const MyVector& a, const MyVector& b) {
        int minSize = min(a.size, b.size);
        MyVector result(minSize);
        for (int i = 0; i < minSize; ++i)
            result.data[i] = a.data[i] - b.data[i];
        return result;
    }

    friend istream& operator>>(istream& in, MyVector& v) {
        for (int i = 0; i < v.size; ++i) {
            cout << "Element " << i + 1 << ": ";
            in >> v.data[i];
        }
        return in;
    }

    friend ostream& operator<<(ostream& out, const MyVector& v) {
        for (int i = 0; i < v.size; ++i) {
            out << v.data[i] << " ";
        }
        return out;
    }
};
```

Rezultatele testării

```
Introduceti 5 elemente ale primului vector:
Element 1: 1
Element 2: 2
Element 3: 3
Element 4: 4
Element 5: 5

Introduceti 5 elemente ale celui de-al doilea vector:
Element 1: 6
Element 2: 5
Element 3: 4
Element 4: 3
Element 5: 2
```

```
Primul vector este: 1 2 3 4 5
Al doilea vector este: 6 5 4 3 2
Suma celor doi vectori: 7 7 7 7 7
Diferenta celor doi vectori: -5 -3 -1 1 3
Modulul primului vector: 7
Modulul celui de-al doilea vector: 9
Comparatii:
V1 == V2 -> 0
V1 != V2 -> 1
V1 < V2  -> 1
V1 > V2  -> 0
```

4. Concluzii

Lucrarea de laborator a avut ca temă implementarea unei clase numite Vector, care reprezintă un vector de tip long și utilizează alocarea dinamică a memoriei. Sarcina principală a constat în definirea și supraîncărcarea unui set extins de operatori, în conformitate cu principiile programării orientate pe obiect în limbajul C++. Operatorii + și -, destinați adunării și scăderii element cu element a două obiecte de tip Vector, au fost implementați sub forma funcțiilor prietene, pentru a permite accesul la membrii interni ai ambilor operanzi. De asemenea, operatorul de atribuire =, precum și operatorii de comparație ==, !=, < și >, au fost definiți ca metode ale clasei. Pentru implementarea corectă a ultimilor doi operatori de comparație, a fost necesară adăugarea unei funcții auxiliare pentru calculul modulului vectorului, oferind astfel un criteriu numeric obiectiv pentru comparație.

Totodată, pentru a facilita interacțiunea cu fluxurile standard de intrare și ieșire, operatorii << și >> au fost supraîncărcați, asigurând astfel o metodă intuitivă de afișare și citire a obiectelor din clasa Vector. Clasa a fost completată cu toți constructorii necesari – implicit, cu parametru și de copiere – precum și cu un destructor care gestionează eliberarea corectă a memoriei dinamice, prevenind eventualele scurgeri de resurse.

Pe parcursul implementării sarcinii practice, nu au fost întâmpinate dificultăți semnificative. Familiaritatea anterioară cu principiile OOP, precum și experiența acumulată în lucrul cu operatori în alte limbaje orientate pe obiect, cum ar fi C#, au facilitat înțelegerea logicii din spatele supraîncărcării operatorilor în C++. Această activitate a contribuit la consolidarea deprinderilor de implementare a claselor robuste, cu funcționalitate extinsă, demonstrând totodată utilitatea și eleganța oferite de supraîncărcarea operatorilor în dezvoltarea unor interfețe intuitive și expresive.

5. Webobrafie

- https://else.fcim.utm.md/course/view.php?id=663
- https://www.w3schools.com/cpp/cpp_oop.asp
- https://www.programiz.com/cpp-programming/oop,
- https://code.visualstudio.com/docs/cpp/config-linux
- https://chatgpt.com/