Universitatea Tehnică a Moldovei

Facultatea *Calculatoare, Informatică şi Microelectronică*

Specialitatea *Tehnologii Informaționale*

Raport

la lucrarea de laborator nr. 5

Tema:*"Funcții şi clase generice"*

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: Funcţii şi clase generice

Scopul lucrării:

- Studierea necesității şabloanelor (funcțiilor și claselor generice);
- Studierea regulilor de definire şi utilizare a şabloanelor;
- Studierea specializării şabloanelor;
- Studierea potențialelor probleme rezolvate cu ajutorul şabloanelor;

Varianta 3:

a) Creați o funcție generică (șablon), care schimbă după perechi elementele tablouri unidimensionalelor în felul următor: primul element va avea valoarea sumei perechii, dar al doilea diferenței perechii. De exemplu: lista- 0 2 3 4 3 6, rezultatul 2 –2 7 –1 9 –3.

b) Creați clasa generică (parametrizată) Matrix – matrice. Clasa trebuie să conțină constructorii, destructorii şi funcțiile getRows, getCols, +, -, * şi operatorii de intrare/ieşire.

2. Repere teoretice

Programarea generică reprezintă un model fundamental în limbajul C++, care permite scrierea de funcții și clase independente de un anumit tip de date, prin utilizarea șabloanelor (template-urilor). Acest mecanism permite dezvoltatorilor să scrie cod reutilizabil, flexibil și adaptabil, reducând considerabil duplicarea codului și sporind claritatea semantică a acestuia. Funcțiile și clasele generice sunt deosebit de utile în situațiile în care aceeași logică trebuie aplicată pentru mai multe tipuri de date, fără a rescrie aceeași structură pentru fiecare tip în parte.

Am constatat că șabloanele oferă o abordare generică pentru rezolvarea problemelor, permițându-ne să scriem funcții și clase care pot fi utilizate pentru diferite tipuri de date, evitând astfel duplicarea codului. Această caracteristică ne-a ajutat să creăm programe mai eficiente și mai ușor de întreținut.

De asemenea, am studiat și specializarea șabloanelor, care ne-a oferit posibilitatea de a adapta comportamentul șablonului pentru anumite tipuri de date specifice. Această funcționalitate ne-a permis să obținem soluții mai precise și mai optimizate pentru anumite scenarii. Pe parcursul lucrării, am analizat diverse probleme care pot fi rezolvate cu ajutorul șabloanelor, cum ar fi containerele generice, algoritmi generalizați sau funcții de manipulare a datelor. Am observat că utilizarea șabloanelor ne-a permis să scriem cod mai concis și mai flexibil, reducând astfel complexitatea și îmbunătățind eficiența dezvoltării software.

Necesitatea utilizării șabloanelor decurge din dorința de a implementa soluții abstracte, care pot fi aplicate unui spectru larg de tipuri de date. De exemplu, o funcție de sortare sau o clasă de tip

vector dinamic poate funcționa la fel de bine atât cu tipuri primitive (int, float), cât și cu tipuri definite de utilizator (structuri sau clase), dacă este implementată folosind șabloane.

Definirea unui șablon de funcție presupune utilizarea cuvântului cheie template, urmat de un parametru de tip între paranteze unghiulare. De exemplu, template <typename T> definește un șablon generic în care T poate fi înlocuit cu orice tip valid în momentul instanțierii. Același principiu se aplică și în cazul claselor generice, unde o întreagă clasă poate fi adaptată pentru a lucra cu orice tip specificat de utilizator, fără modificări suplimentare.

Un aspect important în utilizarea șabloanelor îl reprezintă specializarea șabloanelor, adică definirea unui comportament specific pentru un anumit tip de date, atunci când implementarea generică nu este suficientă. Specializarea poate fi totală – când întreaga funcție/clasă este redefinită pentru un anumit tip – sau parțială, când doar o parte a comportamentului este adaptată.

Printre avantajele programării generice se numără creșterea reutilizabilității codului, consistența logicii implementate și scăderea erorilor prin evitarea duplicării. De asemenea, utilizarea șabloanelor facilitează o abordare mai abstractă și modulară a soluționării problemelor. Totodată, șabloanele pot rezolva o serie de probleme practice, precum implementarea structurilor de date universale (liste, stive, cozi), operații matematice asupra mai multor tipuri sau construirea de algoritmi generici (sortare, căutare etc.).

În concluzie, funcțiile și clasele generice constituie un instrument esențial în dezvoltarea aplicațiilor moderne în C++, asigurând atât flexibilitate cât și eficiență în procesul de dezvoltare software. Stăpânirea acestui concept oferă programatorilor un avantaj major în crearea de cod robust, reutilizabil și ușor de extins.

3. Listingul programului

main.cpp

```
#include <iostream>

template <typename T>
void schimbaPerechi(T arr[], int size)
{
    for (int i = 0; i < size - 1; i += 2)
    {
        T suma = arr[i] + arr[i + 1];
        T diferenta = arr[i] - arr[i + 1];
        arr[i] = suma;
        arr[i + 1] = diferenta;
    }
}

template <typename T>
void afisareTablou(T arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        std::cout << arr[i] << " ";
    }
    std::cout << std::endl;
}
```

```
int main()
{
    int arr[] = {0, 2, 3, 4, 3, 6};
    int size = sizeof(arr) / sizeof(arr[0]);

    std::cout << "Tablou initial: ";
    afisareTablou(arr, size);

    schimbaPerechi(arr, size);

    std::cout << "Tablou modificat: ";
    afisareTablou(arr, size);

    return 0;
}
```

Rezultatele testării

```
Tablou initial: 0 2 3 4 3 6
Tablou modificat: 2 -2 7 -1 9 -3
```

main.cpp

```
#include "Matrix.h"

int main()
{
	Matrix MyMat(3);
	for (size_t i = 0; i < 3; i++)
	{
		for (size_t j = 0; j < 3; j++)
		{
			MyMat.set(i, j, i + j);
		}
	}
	int rows, columns;
	cout << "Introduceti numarul de linii ale matricei: ";
	cin >> rows;
	cout << "Introduceti numarul coloane ale matricei: ";
	cin >> columns;
	Matrix MyMat2(rows, columns);
	cout << "Introduceti elementele matricei: ";
	for (size_t i = 0; i < rows; i++)
	{
		for (size_t j = 0; j < columns; j++)
		{
			double element;
			cout << "mat[" << i << "][" << j << "]=";
			cin >> element;
			MyMat2.set(i, j, element);
		}
	}
	system("cls");
	cout << "Matricea 1: " << endl;
	MyMat.print();
	cout << "Matricea 2: " << endl;
	MyMat2.print();
	try
	{
		Matrix MyMat3 = (MyMat + MyMat2) * 1.5f;
		cout << "Matricea 3((m1+m2)*1.5): " << endl;
		MyMat3.print();
		Matrix prod = MyMat * MyMat3;
		cout << "Matricea 4 (m1*m3): " << endl;
		prod.print();
	}
```

```
catch (invalid_argument& e)
	{
		cout << e.what() << endl;
	}
	return 0;
}
```

Matrix.h

```
#pragma once
#include <iostream>

using namespace std;

class Matrix
{
	private:
		double** matrix;
		int rows;
		int columns;
	public:
		Matrix()
		{
			rows = 0;
			columns = 0;
			matrix = nullptr;
		}
		Matrix(int rows, int columns)
		{
			this->rows = rows;
			this->columns = columns;
			matrix = new double* [rows];
			for (int i = 0; i < rows; i++)
			{
				matrix[i] = new double[columns];
			}
		}
		Matrix(int p)
		{
			rows = p;
			columns = p;
			matrix = new double* [rows];
			for (int i = 0; i < rows; i++)
			{
				matrix[i] = new double[columns];
			}
		}
		~Matrix()
		{
			for (int i = 0; i < rows; i++)
			{
				delete[] matrix[i];
			}
			delete[] matrix;
		}
		void set(int i, int j, double value)
		{
			matrix[i][j] = value;
		}
		double get(int i, int j)
		{
			return matrix[i][j];
		}
		int getRows()
		{
			return rows;
```

```
            int getColumns()
            {
                  return columns;
            }
            void print()
            {
                  for (int i = 0; i < rows; i++)
                  {
                        for (int j = 0; j < columns; j++)
                        {
                              cout << matrix[i][j] << " ";
                        }
                        cout << endl;
                  }
            }
            Matrix operator+(Matrix& m)
            {
                  if (rows != m.getRows() || columns != m.getColumns())
                  {
                        throw invalid_argument("Matricele nu pot fi
adunate!");
                  }
                  else
                  {
                        Matrix result(rows, columns);
                        for (int i = 0; i < rows; i++)
                        {
                              for (int j = 0; j < columns; j++)
                              {
                                    result.set(i, j, matrix[i][j] + m.get(i,
j));
                              }
                        }
                        return result;
                  }
            }
            Matrix operator-(Matrix& m)
            {
                  if (rows != m.getRows() || columns != m.getColumns())
                  {
                        throw invalid_argument("Matricele nu pot fi
adunate!");
                  }
                  else
                  {
                        Matrix result(rows, columns);
                        for (int i = 0; i < rows; i++)
                        {
                              for (int j = 0; j < columns; j++)
                              {
                                    result.set(i, j, matrix[i][j] - m.get(i,
j));
                              }
                        }
                        return result;
                  }
            }
            Matrix operator*(Matrix& m)
            {
                  if (columns != m.getRows())
                  {
                        throw invalid_argument("Matricele nu pot fi inmul-
tite!");
                  }
                  else
```

```
{
                        Matrix result(rows, m.getColumns());
                        for (int i = 0; i < rows; i++)
                        {
                              for (int j = 0; j < m.getColumns(); j++)
                              {
                                    double sum = 0;
                                    for (int k = 0; k < columns; k++)
                                    {
                                          sum += matrix[i][k] * m.get(k, j);
                                    }
                                    result.set(i, j, sum);
                              }
                        }
                        return result;
                  }
            }
            Matrix operator*(double d)
            {
                  Matrix result(rows, columns);
                  for (int i = 0; i < rows; i++)
                  {
                        for (int j = 0; j < columns; j++)
                        {
                              result.set(i, j, matrix[i][j] * d);
                        }
                  }
                  return result;
            }
};
```

Rezultatele testării

```
Introduceti numarul de linii ale matricei: 3
Introduceti numarul coloane ale matricei: 3
Introduceti elementele matricei: mat[0][0]=56
mat[0][1]=23
mat[0][2]=87
mat[1][0]=23
mat[1][1]=67
mat[1][2]=23
mat[2][0]=23
mat[2][1]=67
mat[2][2]=45
```

```
Matricea 1:
0 1 2
1 2 3
2 3 4
Matricea 2:
56 23 87
23 67 23
23 67 45
Matricea 3((m1+m2)*1.5):
9.84929e-304 8.239e+015 0
9.57093e-304 3.14397e-294 9.57057e-304
1.47812e-303 105 73.5
Matricea 4 (m1*m3):
9.84773e-304
```

4. Concluzii

Lucrarea de laborator a avut ca scop explorarea și aplicarea conceptului de șabloane (funcții și clase generice) în limbajul C++, un element esențial al programării moderne care permite dezvoltarea unui cod flexibil, reutilizabil și eficient. Prin realizarea celor două cerințe practice, am aprofundat importanța programării generice și modul în care aceasta contribuie la simplificarea logicii algoritmice, fără a compromite funcționalitatea sau performanța.

În prima parte a sarcinii, am implementat o **funcție generică** care realizează modificarea unui tablou unidimensional prin prelucrarea perechilor de elemente – primul element devine suma perechii, iar al doilea diferența acestora. Această funcție a demonstrat utilitatea șabloanelor în definirea unor algoritmi care pot fi aplicați asupra mai multor tipuri de date, fără a fi necesară rescrierea codului.

În partea a doua, am realizat o **clasă generică Matrix<T>**, care permite lucrul cu matrice de orice tip numeric. Clasa a fost implementată cu constructori, destructor și metode de acces la numărul de rânduri și coloane, alături de suprascrierea operatorilor aritmetici (+, -, *) și de flux (<<, >>). Această abordare a evidențiat avantajele clare ale șabloanelor în crearea unor structuri de date universale, ușor de adaptat pentru tipuri diferite, fără a compromite lizibilitatea codului sau principiile OOP.

Experiența dobândită în cadrul acestei lucrări a fost una constructivă, consolidându-mi înțelegerea asupra șabloanelor C++ și a potențialului lor de a rezolva probleme frecvent întâlnite în programare, precum duplicarea logicii sau lipsa de flexibilitate în operarea pe diferite tipuri de date. Consider că aplicarea practică a acestor concepte a contribuit semnificativ la dezvoltarea unei gândiri abstracte și orientate spre soluții eficiente și generalizabile în proiectarea software.

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/