# Laborator V

## *FUNCŢII MEMBRU, SUPRAÎNCĂRCAREA OPERATORILOR, POINTERUL `this`, MODIFICATORUL `friend`*

### *1. Scopul lucrării*

Lucrarea de laborator are ca scop ilustrarea conceptului de supraîncărcare a operatorilor, utilitatea pointerului implicit **`this`** şi a câtorva elemente de stil de programare.

### *2. Supraîncărcarea operatorilor*

Operaţia de supraîncărcare a operatorilor este similară cu supraîncărcarea funcţiilor. Operatorii predefiniţi pentru tipuri de date predefinite sunt trataţi de compilator ca funcţii cu parametri operanzii. Diferenţa apare în momentul apelului care, formal, este diferit de apelul de funcţie.

Supraîncărcarea operatorilor este o formă de relaxare a modului de utilizare a claselor. Se elimină astfel construcţii speciale, programatorul putând să se concentreze asupra algoritmilor.

Prototipul unui operator supraîncărcat este:

```
class Clasa
{
...
tip_returnat  operator simbol_operator(lista  cu parametrii  formali);
...
}
```

Definiţia operatorului:

```
tip_returnat Clasa::operator simbol_operator(lista  cu parametrii formali)
{
instrucţiuni
}
```

### *3. Pointerul `this`*

Pointerul **`this`** este un pointer implicit asociat automat de compilator cu fiecare obiect de un tip **`class`** definit. Pointerul **`this`** are tipul clasei obiectului şi valoarea adresei la care este memorat obiectul.

Pointerul **`this`** este transferat implicit funcţiei membru care este apelată folosind un obiect. Acest pointer este utilizat implicit de compilator. Deşi este corectă utilizarea explicită a acestui pointer, este bine ca utilizarea lui explicită să rămână o soluţie pentru cazurile în care aceasta este indispensabilă.

Formal, prototipul unei funcţii membru şi apelul acesteia poate fi văzut astfel:

```
class Clasa
{
...
tip_returnat funcţie_membru(Clasa  *this, lista cu parametrii formali);
...
}
...
Clasa ObiectClasa; //Se apelează constructorul implicit
...
ObiectClasa.funcţie_membru(&ObiectClasa, lista cu parametri reali);
//Apelul funcţiei membru
...
```

în loc de:

```
class Clasa
{
...
tip_returnat funcţie_membru(lista  cu parametrii formali);
...
}
...
Clasa ObiectClasa; //Se apelează constructorul implicit
...
ObiectClasa.funcţie_membru(lista cu parametri reali);
//Apelul funcţiei membru ...
```

Bineînţeles, prima variantă are ca rol ilustrarea conceptului şi nu va fi folosită ca atare în programe.

### *4. Friend*

Obiectele declarate **friend** ocolesc mecanismul de încapsulare propriu claselor. Această ocolire poate fi extrem de benefică în câteva cazuri, în altele poate fi indispensabilă, iar în multe altele poate conduce la mari neplăceri. Dacă se poate, este de dorit să se evite folosirea membrilor **friend**.

Membrii declaraţi **friend** pentru o clasă au acces direct la toţi membrii acesteia. Funcţiile declarate **friend** nu au acces la membrii clasei prin pointerul **this** ca funcţiile membru, deoarece nu sunt asociate direct cu un obiect, ci mai degrabă li se permite accesul la membrii clasei de tipul căreia este obiectul.

### *5. Program*

Codul care urmează reprezintă definiţia pentru un tip de date matrice de întregi *MatrixInt2*.

```
/***********************************************
matrixcpp.hpp
***********************************************/
```

```
#include <iostream>
#include <iomanip>
#include <conio.h>
#include <ctype.h>
using namespace std;

#ifndef _MATRIXCPP3_INCLUDED_
#define _MATRIXCPP3_INCLUDED_

/*
//Nu este nevoie de ele decat in versiuni mai vechi ...
enum Boolean
{
    false = 0,
    true = 1
};
*/

enum Error
{
    error = 0,
    success = 1
};

enum ObjState
{
    Destroyed = 0,
    Constructed = 1,

};
```

```
//Declaratia de clasa ...
class MatrixInt2
{
private:
    int State; //Starea curenta a obiectului ...
    int **Matrix;
    int NrLin;
    int NrCol;
protected:
    void Copy(MatrixInt2 const &); //Realizeaza copierea ...
public:
    MatrixInt2(void); //Constructor implicit ...
    MatrixInt2(MatrixInt2 const &); //Constructor de copiere ...
    MatrixInt2(int, int); //Constructor ...
    ~MatrixInt2(void); //Destructor ...
    void ClearAndDestroy(void);
    int SetUp(int, int); //Returneaza starea de eroare ...

    void Print(ostream & = cout, int = 3) const; //Tipareste matricea la consola ...
    friend ostream & operator <<(ostream &, MatrixInt2 const &);

    int & operator()(int, int); //Operator de acces aleatoriu ...
    MatrixInt2 & operator =(MatrixInt2 const &); //Operator de atribuire ...
    MatrixInt2 & operator +=(MatrixInt2 const &); //Operator de adunare ...
    MatrixInt2 & operator *=(int); //Operator de inmultire cu scalar intreg ...
    MatrixInt2 operator +(MatrixInt2 const &); //Operator de adunare ...
    MatrixInt2 operator *(int); //Operator de inmultire ...

    int operator ==(MatrixInt2 const &); //Operator relational ...

    int GetNrLin(void) {return NrLin;}
    int GetNrCol(void) {return NrCol;}
    int ** GetMatrix(void) {return Matrix;}
};

/*Testeaza daca valoarea pointerului este nullptr.
Daca da returneaza True, altfel returneaza False.*/
int NULLMemTest(void *);

/*Testeaza daca valoarea pointerului este nullptr.
Daca da returneaza True si afiseaza un mesaj, altfel returneaza False.*/
int AllocError(void *);

#endif
```

```
/*********************************************
consmatr.cpp
*********************************************/
```

```
#include "matrixcpp.hpp"

//Constructor implicit ...
MatrixInt2::MatrixInt2(void)
{
    State = Constructed;
    Matrix = nullptr;
    NrLin = NrCol = 0;
}
```

```
//Constructor de copiere ...
MatrixInt2::MatrixInt2(MatrixInt2 const &M)
{
    State = Constructed;
    Matrix = NULL;
    NrLin = NrCol = 0;
    *this = M; //Se foloseste operatorul de atribuire ...
}

MatrixInt2::MatrixInt2(int NL, int NC)
{
    State = Constructed;
    Matrix = NULL;
    NrLin = NrCol = 0;
    SetUp(NL, NC);
}
```

```
/*********************************************
destmatr.cpp
*********************************************/
```

```
#include "matrixcpp.hpp"

//Elibereaza memoria alocata dinamic pentru o matrice.
void MatrixInt2::ClearAndDestroy(void)
{
    if(!State)
        return;

    //Se testeaza daca memoria pentru matrice a fost alocata ...
    if(NULLMemTest(Matrix))
    {
        NrLin = NrCol = 0;
        State = Destroyed;
        return;
    }

    int **MatrixTemp=Matrix;
    //Se sterge memoria pe linii ...
    for(int contlin = 0; contlin < NrLin; contlin++)
    {
        //Se testeaza daca a fost alocata memorie pentru linia curenta ...
        if(NULLMemTest(*MatrixTemp++))
        delete []MatrixTemp; //Se sterge memoria pentru linia curenta ...
                break;
            //*MatrixTemp++;
    }
    delete []Matrix;
    Matrix = NULL;
    NrLin = NrCol = 0;
    State = Destroyed;
}

MatrixInt2::~MatrixInt2(void)
{
    ClearAndDestroy();
}
```

```
/*********************************************
memerror.cpp
*********************************************/
```

```
#include "matrixcpp.hpp"

/*Testeaza daca valoarea pointerului este nullptr.
Daca da returneaza True, altfel returneaza False.*/
int NULLMemTest(void *Ptr)
{
    if(!Ptr)
        return true;
    return false;
}

/*Testeaza daca valoarea pointerului este nullptr.
Daca da returneaza True si afiseaza un mesaj, altfel returneaza False.*/
int AllocError(void * Ptr)
{
    if(NULLMemTest(Ptr))
    {
        cout << "Eroare la alocarea memoriei ...\n";
        return true;
    }
    return false;
}
```

```
/*********************************************
mfctmatr.cpp
*********************************************/
```

```
#include "matrixcpp.hpp"

//Relizeaza o copie a matricii ...
void MatrixInt2::Copy(MatrixInt2 const &M)
{
    NrLin = M.NrLin;
    NrCol = M.NrCol;
    State = Constructed;
    if(!State)
    {
        Matrix = nullptr;
        return;
    }
    if(!SetUp(NrLin, NrCol))
        return;
    int **MatrixTemp=M.Matrix;
    for(int contorlin = 0; contorlin < NrLin; contorlin++)
        for(int contorcol = 0; contorcol < NrCol; contorcol++)
                Matrix[contorlin][contorcol] = MatrixTemp[contorlin][contorcol];
}

//Aloca dinamic memorie pentu o matrice cu NrLin linii si NrCol coloane.
//Returneaza error sau success.
int MatrixInt2::SetUp(int NrLin, int NrCol)
{
    if(State != Destroyed)
        ClearAndDestroy();
    State = Constructed;

    MatrixInt2::NrLin = NrLin;
    MatrixInt2::NrCol = NrCol;
```

```
    /*Se aloca memorie pentru pointeri la linii ...*/
    Matrix = new int *[NrLin]; //Se foloseste operatorul new (specific C++)
    if(AllocError(Matrix))
    {
        ClearAndDestroy();
        return error;
    }

    int ** MatrixTemp=Matrix;
    //Initializarea folosind sintaxa de la C++ (constructor de copiere) ...
    for(int contorlin = 0; contorlin < NrLin; (contorlin++, MatrixTemp++))
    {
        /*Se aloca memorie pentru o linie ...*/
        *MatrixTemp = new int[NrCol];
        if(AllocError(*MatrixTemp))
        {
            /*In caz de eroare la alocare se sterge memoria deja alocata ...*/
            ClearAndDestroy();
            return error;
        }
    }
    return success;
}

/*********************************************
opermatr.cpp
*********************************************/

#include "matrixcpp.hpp"

int & MatrixInt2::operator()(int i, int j)
{
    if(!State || i >= NrLin || i < 0 || j >= NrCol || j < 0 )
    {
        cout << "Eroare la apelarea operatorului (,)\n";
        return State;
    }
    return Matrix[i][j];
}

MatrixInt2 & MatrixInt2::operator =(MatrixInt2 const &M) //
{
    if(!M.State)
    {
        ClearAndDestroy();
        return *this;
    }
    ClearAndDestroy(); //Este distrusa vechea matrice ...
    Copy(M); //Vechea matrice este inlocuita cu o copie a argumentului ...
    return *this;
}

MatrixInt2 & MatrixInt2::operator +=(MatrixInt2 const &M)
{
    if(!M.State) //Matricea M nu exista ...
        return *this;
    int MustChange = 0;
    int NrLinTemp, NrLinTempPrim;
    int NrColTemp, NrColTempPrim;
    NrLinTemp = NrLinTempPrim = M.NrLin;
    NrColTemp = NrColTempPrim = M.NrCol;
    if(NrLin > NrLinTemp)
```

```
    {
        NrLinTemp = NrLin;
        MustChange = 1;
    }
    if(NrCol > NrColTemp)
    {
        NrColTemp = NrCol;
        MustChange = 1;
    }
    if(MustChange) //Daca dimensiunile matricii originale nu sunt cele adecvate ..
    {
        MatrixInt2 MatrixInt2Temp(*this);
        //Se realizeaza o copie temporara a obiectului curent ...
        ClearAndDestroy(); //Se distruge obiectul curent ...
        if(!SetUp(NrLinTemp, NrColTemp)) //Se redimensioneaza obiectul curent ...
                return *this;
        //Se refac valorile matricii originale in noua matrice (mai mare) ...
        int **MatrixTemp=MatrixInt2Temp.Matrix;
        for(int contorlin = 0; contorlin < NrLin; contorlin++)
            for(int contorcol = 0; contorcol < NrCol; contorcol++)
                    Matrix[contorlin][contorcol] = MatrixTemp[contorlin][contorcol];
        //Se initializeaza cu 0 valorile din zonele bordate ...
        for(int contorlin = 0; contorlin < NrLin; contorlin++)
            for(int contorcol = NrCol; contorcol < NrColTemp; contorcol++)
                Matrix[contorlin][contorcol] = 0;
         for(int contorlin = NrLin; contorlin < NrLinTemp; contorlin++)
              for(int contorcol = 0; contorcol < NrColTemp; contorcol++)
                Matrix[contorlin][contorcol] = 0;
    }
    int **MTemp=M.Matrix;
    for(int contorlin = 0; contorlin < NrLinTempPrim; contorlin++)
        for(int contorcol = 0; contorcol < NrColTempPrim; contorcol++)
                Matrix[contorlin][contorcol] += MTemp[contorlin][contorcol];
    return *this;
}

MatrixInt2 & MatrixInt2::operator *=(int Scalar)
{
    for(int contorlin = 0; contorlin < NrLin; contorlin++)
        for(int contorcol = 0; contorcol < NrCol; contorcol++)
                Matrix[contorlin][contorcol] *= Scalar;
    return *this;
}

MatrixInt2 MatrixInt2::operator +(MatrixInt2 const &M)
{
    if(!M.State)
        return *this;
    MatrixInt2 MatrixResult(*this);
    //Matrice rezultat initializata cu valoarea matricii curente ...
    MatrixResult += M; //Operator binecunoscut ...
    return MatrixResult; //Se intoarce rezultatul adunarii ...
}

MatrixInt2 MatrixInt2::operator *(int Scalar)
{
    MatrixInt2 MatrixInt2Temp(NrLin, NrCol);
    int **MatrixTemp=MatrixInt2Temp.Matrix;
    for(int contorlin = 0; contorlin < NrLin; contorlin++)
        for(int contorcol = 0; contorcol < NrCol; contorcol++)
               MatrixTemp[contorlin][contorcol] = Matrix[contorlin][contorcol] * Scalar;
    return MatrixInt2Temp;
}
```

```
int MatrixInt2::operator ==(MatrixInt2 const &M)
{
    if(NrLin != M.NrLin && NrCol != M.NrCol)
        return false;

    int **MatrixTemp=M.Matrix;
    for(int contorlin = 0; contorlin < NrLin; contorlin++)
        for(int contorcol = 0; contorcol < NrCol; contorcol++)
                if(Matrix[contorlin][contorcol] != MatrixTemp[contorlin][contorcol])
                        return false;
    return true;
}
```

```
/*********************************************
prntmatr.cpp
*********************************************/
```

```
#include "matrixcpp.hpp"

/*Tipareste la consola matricea de intregi specificata.*/
void MatrixInt2::Print(ostream &StrOut, int Width) const
{
    StrOut << "Sa afisez matricea ? (y/n)";
    int Test;
    do
    {
        Test = toupper(cin.get());
    }
    while(Test != 'Y' && Test != 'N');

    if(Test == 'Y')
    {
        StrOut << "Matricea este :\n";
        for(int contorlin = 0; contorlin < NrLin; contorlin++)
        {
            for(int contorcol = 0; contorcol < NrCol; contorcol++)
                    StrOut << setw(Width) << Matrix[contorlin][contorcol];
            StrOut << endl;
        }
    }
}

ostream & operator <<(ostream &StrOut, MatrixInt2 const &M)
{
    M.Print(StrOut,8);
    return StrOut;
}
```

```
/*********************************************
mainmatr.cpp
*********************************************/
```

```
#include "matrixcpp.hpp"

void main(void)
{
    MatrixInt2 Matrix1, Matrix2;

    int NrLin, NrCol;

    cout << "Introduceti numarul de linii: ";
    cin >> NrLin;

    cout << "Introduceti numarul de coloane: ";
    cin >> NrCol;

    Matrix1.SetUp(NrLin, NrCol);

    Matrix1.Print(); //Matricea nu a fost initializata ...

    for(int contlin = 0; contlin < Matrix1.GetNrLin(); contlin++)
        for(int contcol = 0; contcol < Matrix1.GetNrCol(); contcol++)
                Matrix1(contlin, contcol) = contlin + contcol;
    Matrix1.Print();
    Matrix2 = Matrix1;
    cout << "Relatia intre cele doua matrici este: " << (Matrix2 == Matrix1) << endl;
    Matrix2.Print();
    MatrixInt2 Matrix3(Matrix2);
    Matrix1.ClearAndDestroy();
    Matrix3.Print();
    Matrix2 += Matrix3;
    Matrix2.Print();
    Matrix3 *= 100;
    Matrix3.Print(cout, 5);
    MatrixInt2 Matrix4;
    Matrix4 = Matrix2 + Matrix3;
    Matrix4.Print(cout, 5);
    MatrixInt2 Matrix5;
    Matrix5 = Matrix4 * 100;
    Matrix5.Print(cout, 8);
    cout << Matrix5 << "Gata ... pentru moment ..." << endl;
    Matrix3 = Matrix4 = Matrix5 + Matrix2;
    cout << Matrix3 << Matrix4 << Matrix5 + Matrix2 << "Gata ..." << endl;
}
```

***6. Temă***

Rulaţi programul pas cu pas şi urmăriţi funcţionarea acestuia. Observaţi comportarea funcţiilor şi operatorilor pentru alocarea dinamică de memorie. Pentru clasa *MatrixInt2* constataţi o redundanţă în exprimarea stării obiectului. Eliminaţi această redundanţă.