# PROGRAMARE ORIENTATĂ PE OBIECTE

# LABORATOR 5

**Anul universitar 2023 – 2024**

**Semestrul II**

## *1 Noțiuni teoretice*

### *1.1 Generalități*

Sunt situații în care necesarul de memorie se poate determina abia în momentul execuției programului. De exemplu, atunci când cantitatea de memorie depinde de datele de intrare furnizate de către utilizator. În aceste situații, programele au nevoie să aloce memorie dinamic, motiv pentru care limbajul C++ integrează operatorii new și delete.

### *1.2 Operatorii new și new[]*

Memoria dinamică este alocată folosind operatorul new. new este urmat de un tip de dată și, dacă este cazul uneia sau mai multor secvențe, de numărul de elemente al acestora cuprins între paranteze drepte []. Operatorul returnează un pointer către începutul blocului de memorie alocat. Sintaxa este:

```
pointer = new tip_data
pointer = new tip_data [numar_de_elemente]
```

Prima expresie este folosită pentru a aloca memorie care conține un singur element de tipul tip_data. A doua instrucțiune este folosită pentru a aloca un bloc (tablou) de elemente de tip tip_data, unde numar_de_elemente este o valoare întreagă reprezentând numărul acestora. De exemplu:

```
int * foo;
foo = new int [5];
```

În acest caz, sistemul alocă dinamic spațiu pentru cinci elemente de tip int și returnează un pointer la primul element al secvenței, pointer care i se atribuie lui foo (variabilă pointer). De aceea, foo pointează acum către un bloc valid de memorie care poate stoca până la cinci elemente de tip int.

Aici, foo este un pointer și, de aceea, primul element către care pointează foo poate fi accesat die cu expresia foo[0] fie cu expresia *foo (cele două sunt echivalente). Al doilea element poate fi accesat fie cu foo[1] fie cu *(foo+1), și așa mai departe...

Există o mare diferență între declararea normală a unui tablou și alocarea dinamică a unui bloc de memorie cu operatorul new. Cel mai important este faptul că dimensiunea unui tablou normal trebuie să fie o expresie constantă, ceea ce înseamnă că dimensiunea trebuie determinată la momentul proiectării programului, înainte de rularea lui, în timp ce alocarea dinamică realizată cu operatorul new permite gestionarea memoriei în timpul execuției, folosind ca dimensiune orice variabilă.

Memoria dinamică solicitată de programul nostru este alocată de către sistem din memoria heap. Totuși, memoria calculatorului este o resursă limitată și poate fi epuizată. De aceea, nu avem există garanție că toate solicitările de alocare dinamică folosind operatorul new vor fi onorate de către sistem.

C++ asigură două mecanisme pentru a verifica dacă alocarea a fost reușită.

Unul presupune gestionarea excepțiilor. Folosind această metodă, o excepție de tipul bad_alloc apare când alocarea eșuează. Excepțiile sunt un instrument puternic C++ pe care îl vom explica mai târziu în aceste lecții. Deocamdată, ar trebui să știți că dacă apare o asemnea exceptie și nu este tratată de un handler specific, execuția programului se încheie.

Aceasta este metoda folosită implcit de către operatorul new și cea folosită într-o declarație ca următoarea:

```
foo = new int [5];  // dacă alocarea eșuează, apare o excepție
```

Cealaltă metodă este cunoscută ca nothrow și când o alocare eșuează, în loc de apariția excepției bad_alloc sau terminarea programului, pointer-ul returnat de new este un pointer nul, iar programul își continuă execuția normală.

Această metodă poate fi precizată printr folosirea unui obiect special numit nothrow, declarat în header <new>, ca argument pentru new:

```
foo = new (nothrow) int [5];
```

În acest caz, dacă eșuează alocarea acestui bloc de memorie, eșecul poate fi detectat verificând dacă foo este pointerul nul:

```
int * foo;
foo = new (nothrow) int [5];
if (foo == nullptr) {
  // eroare la alocarea memoriei. Luăm măsuri.
}
```

Metoda nothrow pare să producă un cod mai puțin eficient decât excepțiile, căci presupune verificarea explicită a obținerii pointerului nul după fiecare alocare. De aceea, în general, este preferat mecanismul excepțiilor, cel puțin pentru alocările critice. Totuși, majoritatea exemplelor noastre vor folosi mecanismul nothrow pentru simplicitatea sa.

### *1.3 Operatorii delete și delete[]*

În cele mai multe cazuri, memoria alocată dinamic este necesară doar pentru o anumită perioadă de timp în cadrul unui program; de îndată ce nu mai este necesară, poate fi eliberată astfel incăt memoria să

redevină disponibilă pentru alte solicitări de memorie dinamică. Pentru aceasta avem operatorul delete, a cărui sintaxă este:

```
delete pointer;
delete[] pointer;
```

Prima instrucțiune eliberează memoria pentru un singur element folosind new, iar a doua eliberează memoria alocată pentru un tablou de elemente folosind new și o dimensiune între paranteze drepte ([]).

Valoarea transmisă ca parametru delete poate fi un pointer către un bloc de memorie alocat anterior cu new sau un pointer nul (în cazul unui pointer nul, delete nu are niciun efect).

```
// rememb-o-matic
#include <iostream>
#include <new>
using namespace std;
int main ()
{
  int i,n;
  int * p;
  cout << "Cate numere vrei sa tastezi? ";
  cin >> i;
  p= new (nothrow) int[i];
  if (p == nullptr)
    cout << "Eroare: memoria nu poate fi alocata!";
  else
  {
    for (n=0; n<i; n++)
    {
      cout << "Tasteaza numar: ";
      cin >> p[n];
    }
    cout << "Ai tastat: ";
    for (n=0; n<i; n++)
      cout << p[n] << ", ";
    delete[] p;
  }
  return 0;
}
Cate numere vrei sa tastezi? 5
Tasteaza numar: 75
Tasteaza numar: 436
Tasteaza numar: 1067
Tasteaza numar: 8
Tasteaza numar: 32
Ai tastat: 75, 436, 1067, 8, 32,
```

Există întotdeauna posibilitatea ca un utilizator să dea pentru i o valoare așa de mare încât sistemul să nu dispună de suficientă memorie pentru a o aloca. De exemplu, când am răspuns cu 1 billion la

întrebarea "Cate numere vrei sa tastezi? ", sistemul meu nu a putut aloca atâta memorie pentru program și am primit pe ecran mesajul pregătit pentru asemenea situații (Eroare: memoria nu poate fi alocata!).

Se consideră o buna practică ca programul să fie scris astfel încât să fie capabil de a gestiona eșecurile în cazul alocării de memorie, fie prin verificarea valorii pointerului (pentru nothrow) fie prin prinderea excepției.

### *1.4 Memoria dinamică în C*

C++ integrează operatorii new și delete pentru alocarea dinamică a memoriei. Dar acești operatori nu erau definiți în limbajul C; în schimb, se folosea o bibliotecă din care se apelau funcțiile malloc, calloc, realloc și free, definite în header-ul <cstdlib> (cunoscut ca <stdlib.h> în C). Funcțiile sunt încă disponibile și în C++ și pot fi folosite pentru alocarea și eliberarea memoriei dinamice.

Să observăm, totuși, că blocurile de memorie alocate cu aceste funcții nu sunt întotdeauna compatibile cu cele returnate de new, deci ele nu pot fi amestecate; fiecare ar trebui gestionat cu propriul set de funcții sau operatori.

### *1.5 Pointeri la clase*

Obiectele pot fi și ele referite prin pointeri: odată declarată, o clasă devine un tip de dată valid, deci poate fi folosită ca tip de dată spre care să pointeze un pointer. De exemplu:

```
Dreptunghi * pdrept;
```

este un pointer către un obiect al clasei Dreptunghi.

Analog structurilor de date obișnuite, membrii unui obiect pot fi accesați direct printr-un pointer folosind operatorul săgeată (->). Iată un exemplu cu câteva posibile combinații:

```
// exemplu de pointer la clase
#include <iostream>
using namespace std;

class Dreptunghi {
  int latime, inaltime;
public:
  Dreptunghi(int x, int y) : latime(x), inaltime(y) {}
  int aria(void) { return latime * inaltime; }
};


int main() {
  Dreptunghi obj (3, 4);
  Dreptunghi * foo, * bar, * baz;
  foo = &obj;
  bar = new Dreptunghi (5, 6);
  baz = new Dreptunghi[2] { {2,5}, {3,6} };
```

```
  cout << "aria lui obj: " << obj.aria() << '\n';
  cout << "aria lui *foo: " << foo->aria() << '\n';
  cout << "aria lui *bar: " << bar->aria() << '\n';
  cout << "aria lui baz[0]:" << baz[0].aria() << '\n';
  cout << "aria lui baz[1]:" << baz[1].aria() << '\n';
  delete bar;
  delete[] baz;
  return 0;
}
```

## *2 Aplicații*

### *2.1 Vector dinamic*

Să se definească o clasă *Vector* care să reprezinte un vector de numere întregi. Clasa trebuie să aibă:

- Două date membru: *m* (numărul de elemente) și *data* (elementele vectorului).
- Un constructor care primește ca parametri numărul de elemente al vectorului și inițializează vectorul cu elemente introduse de la tastatură.
- Un destructor care să elibereze memoria alocată dinamic pentru vector.
- O metodă *adunare* care primește ca parametru un alt vector și returnează un nou vector ce reprezintă rezultatul adunării vectorilor.
- O metodă *afisare* care să afișeze elementele vectorului.

Folosind această clasă, să se creeze două obiecte de tip Vector, să se efectueze adunarea acestora și să se afișeze rezultatul.

### *2.2 Vector static*

Definiți aceeași clasă, dar alocați static vectorul (nu alocare dinamică).

# 3 Rezolvări

## 3.1 Vector dinamic

```
#include <iostream>
using namespace std;

class Vector {
private:
    int* data;
    int m;
    static const int MAX_SIZE = 100;

public:

    Vector(int size) : m(size) {
        if (size > MAX_SIZE) {
            cerr << "Dimensiunea vectorului este prea mare" << endl;
            exit(1);
        }
        data = new int[size];
        cout << "Introduceti " << size << " elemente:\n";
        for (int i = 0; i < size; i++) {
            cin >> data[i];
        }
    }

    // constructor implicit
    Vector() : data(NULL), m(0) {}

    // constructor de copiere (deep copy)
    Vector(const Vector& other) {
        m = other.m;
        data = new int[m];
        for (int i = 0; i < m; ++i) {
            data[i] = other.data[i];
        }
    }

    // destructor
    ~Vector() {
        delete[] data;
    }

    // metoda de adunare
    Vector adunare(const Vector& v) const {
        if (m != v.m) {
            cerr << "Vectorii au dimensiuni diferite!" << endl;
            exit(1);
        }

        Vector rezultat;
        rezultat.m = m;
        rezultat.data = new int[m];
```

```
        for (int j = 0; j < m; j++) {
            rezultat.data[j] = data[j] + v.data[j];
        }

        return rezultat;
    }

    // afisare
    void afisare() const {
        cout << "Elementele vectorului: ";
        for (int i = 0; i < m; i++) {
            cout << data[i] << " ";
        }
        cout << '\n';
    }
};

int main() {
    int size;
    cout << "Introduceti dimensiunea vectorului: ";
    cin >> size;

    Vector v1(size);
    Vector v2(size);
    Vector v_add = v1.adunare(v2); // sau Vector v_add(v1.adunare(v2);
    cout << "Rezultatul adunarii: ";
    v_add.afisare();

    return 0;
}
```

### *3.2 Vector static*

```
#include <iostream>

using namespace std;

class Vector
{

private:
    int m;
    static const int MAX_SIZE = 100;
    int data[MAX_SIZE];

public:

    Vector (int m1): m(m1)
    {
        cout << "Introduceti elementele: " << "\n";
        for(int i = 0; i < m; i++)
        {
```

```
            cin >> data[i];
        }
    }

    Vector(): m(0) {}

    Vector adunare(Vector& v)
    {
        Vector rezultat;

        rezultat.m = m;

        for(int i = 0; i < m; i++)
        {
            rezultat.data[i] = data[i] + v.data[i];
        }

        return rezultat;
    }

    void afisare()
    {
        cout << "Elementele vectorului sunt: \n";
        for (int i = 0 ; i < m; i++)
        {
            cout << data[i] << " ";
        }
        cout << "\n";
    }

};

int main()
{
    int dim;
    cout << "Dimensiunea vectorului este: ";
    cin >> dim;

    Vector v1(dim);
    Vector v2(dim);

    Vector v3;

    v3 = v1.adunare(v2);

    v3.afisare();

    return 0;
}
```

## *4 Teme*

Denumiți proiectele realizate de voi cu denumirea problemei și numele vostru. Exemplu: ***matrice_Bold Nicolae***.

### *4.1 Matrice*

Să se definească o clasă *Matrice* care să reprezinte o matrice de numere întregi. Clasa trebuie să aibă:

- Trei date membru: *m* (numărul de linii), *n* (numărul de coloane) și *data* (elementele matricii).
- Un constructor care primește ca parametri numărul de linii și de coloane ale matricei și inițializează matricea cu elemente introduse de la tastatură.
- Un destructor care să elibereze memoria alocată dinamic pentru matrice.
- O metodă *adunare* care primește ca parametru o altă matrice și returnează o nouă matrice ce reprezintă rezultatul adunării matricelor.
- O metodă *afisare* care să afișeze elementele matricei.

Folosind această clasă, să se creeze două obiecte de tip Matrice, să se efectueze adunarea acestora și să se afișeze rezultatul.

Indicații:

1. Folosiți ca model problema Vector dinamic.
2. Matricea poate fi privită ca un vector de vectori.
3. Atributul clasei care conține elementele matricii poate fi declarat astfel:

```
int** data;
```

4. Alocarea dinamică a matricii se poate face folosind următorul cod:

```
data = new int*[m];
for (int i = 0; i < m; i++) {
     data[i] = new int[n];
}
```

5. Dezalocarea matricii se poate face folosind următorul cod:

```
~Matrice() {
        for (int i = 0; i < m; i++) {
            delete[] data[i];
        }
        delete[] data;
    }
```