

## **Curs 4**

- **C++ Core Guidelines**
- **Clase si obiecte**
- **Clase predefinite: string, vector**
- **Template**

## **Curs 3**

- **Gestiunea memoriei in C**
- **Încapsulare, abstractizare in C**
- **Limbajul de programare C++**
  - **Elemente de limbaj noi (c++ 11/14)**

## C++ Core Guidelines

Editors: [Bjarne Stroustrup](#), [Herb Sutter](#)

<https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>

„**Inside C++, there is a much smaller and cleaner language** struggling to get out.” - B. Stroustrup

Reguli recomandări C++ pentru a crea aplicații mai ușor de întreținut cu mai puține defecte

Propune sa ajute programatorii sa adopte Modern C++ (C++ 11, 14, 17)

- [I.2: Avoid global variables](#)
- [I.5: State preconditions \(if any\)](#)
- [I.7: State postconditions](#)
- [I.10: Use exceptions to signal a failure to perform a required task](#)
- [I.13: Do not pass an array as a single pointer](#)
- [I.22: Avoid complex initialization of global objects](#)
- [I.23: Keep the number of function arguments low](#)
- [I.24: Avoid adjacent unrelated parameters of the same type](#)
- [F.1: "Package" meaningful operations as carefully named functions](#)
- [F.2: A function should perform a single logical operation](#)
- [F.3: Keep functions short and simple](#)
- [F.8: Prefer pure functions](#)

Tema pentru acasă, de citit:

- [Introduction](#)
- [Philosophy](#)

## Clase și obiecte în C++

**Class:** Un tip de dată definit de programator. Descrie caracteristicile unui lucru.

Grupează:

- date – **atribute**
- comportament – **metode**

Clasa este definită într-un fișier header (.h)

Implementarea metodelor se pun într-un fișier .cpp

Sintaxă:

```
//in file rational.h
/**
 * Represent rational numbers
 */
class Rational {
public:
    //methods
    /**
     * Add an integer number to the rational number
     */
    void add(int val);
    /**
     * multiply with a rational number
     * r rational number
     */
    void mul(Rational r);
private:
    //fields (members)
    int a;
    int b;
};
```

```
//in file rational.cpp
/**
 * Add an integer number to the rational number
 */
void Rational::add(int val) {
    a = a + val * b;
}
```

## Definiții de metode

Metodele declarate în clasă sunt definite într-un fișier separat (.cpp)

Se folosește operatorul :: (scope operator) pentru a indica apartenența metodei la clasă

Similar ca și la module se separa declarațiile (interfața) de implementări

```
/**
 * Add an integer number to the rational number
 */
void Rational::add(int val) {
    a = a + val * b;
}
```

Se pot defini metode direct în fișierul header. - **metode inline**

```
class Rational {
public:
    /**
     * Return the numerator of the number
     */
    int getNumerator() {
        return a;
    }
    /**
     * Get the denominator of the fraction
     */
    int getDenominator() {
        return b;
    }
private:
    //fields (members)
    int a;
    int b;
}
```

Putem folosi metode inline doar pentru metode simple (fără cicluri)

Compilatorul inserează (inline) corpul metodei în fiecare loc unde se apelează metoda.

# Obiect

Clasa descrie un nou tip de data.

Obiect - o instanță nouă (o valoare) de tipul descris de clasă

## Declarație de obiecte

<nume\_clasă> <identificator>;

- se alocă memorie suficientă pentru a stoca o valoare de tipul <nume\_clasă>
- obiectul se inițializează apelând constructorul implicit (cel fără parametri)
- pentru inițializare putem folosi și constructori cu parametri (dacă în clasă am definit constructor cu argumente)

```
Rational r1 = Rational{1, 2};
Rational r2{1, 3};
Rational r3;
cout << r1.toFloat() << endl;
cout << r2.toFloat() << endl;
cout << r3.toFloat() << endl;
Rational r4 = Rational(1, 2);
```

## Acces la attribute (câmpuri)

În interiorul clasei

```
int getDenominator() {  
    return b;  
}
```

Când implementăm metodele avem acces direct la attribute

```
int getNumerator() {  
    return this->a;  
}
```

Putem accesa atributul folosind pointerul **this**. Util dacă mai avem variabile cu același nume în metodă (parametru, variabilă locală)

**this**: pointer la instanța curentă. Avem acces la acest pointer în toate metodele clasei, toate metodele membre din clasă au acces la **this**.

Putem accesa attributele și în afara clasei (dacă sunt vizibile)

- Folosind operatorul **'.' object.field**
- Folosind operatorul **'->'** dacă avem o referință (pointer) la obiect **object\_reference->field** is a sau **(\*object reference).field**

# Protecția atributelor și metodelor .

**Modificatori de acces:** Definesc cine poate accesa atributele / metodele din clasă

**public:** poate fi accesat de oriunde

**private:** poate fi accesat doar în interiorul clasei

Atributele (reprezentarea) se declară private

Folosiți funcții (getter/setter) pentru accesa atributele

```
class Rational {
public:
    /**
     * Return the numerator of the number
     */
    int getNumerator() {
        return a;
    }
    /**
     * Get the denominator of the fraction
     */
    int getDenominator() {
        return b;
    }
private:
    //fields (members)
    int a;
    int b;
};
```

## Constructor

**Constructor:** Metoda specială folosită pentru inițializarea obiectelor.

Metoda este apelată când se creează instanțe noi (se declară o variabilă locală, se creează un obiect folosind **new**)

Numele coincide cu numele clasei, nu are tip returnat

Constructorul alocă memorie pentru datele membre, inițializează atributele

<pre>class Rational { public:     Rational(); private:     //fields (members)     int a;     int b; };</pre>	<pre>Rational::Rational() {     a = 0;     this-&gt;b = 1; }</pre>
--	--

**Este apelat de fiecare dată când un obiect nou se creează** – nu se poate crea un obiect fără a apela (implicit sau explicit) constructorul

Orice clasă are cel puțin un constructor (dacă nu se declară unu există un constructor implicit)

Constructorul fără parametri este constructorul implicit (este folosit automat la declararea unei variabile, la declararea unei vector de obiecte)

Într-o clasă putem avea mai mulți constructori, constructorul poate avea parametri.

<pre>class Rational { public:     Rational();     Rational(int a,int b); private:     int a;     int b; };</pre>	<pre>Rational::Rational() {     a = 0;     this-&gt;b = 1; } Rational::Rational(int a,int b):a{a},b{b} { }</pre>
--	--

```
Rational r1 = Rational{1, 2}; //apeleaza constructor cu 2 parametrii  
Rational r2{1, 3}; //apeleaza constructor cu 2 parametrii  
Rational r3; //apeleaza constructor implicit (0 parametrii)
```

Atributele clasei se pot inițializa (member initialization list) adăugând o lista înainte de corpul metodei (expresia între “:” și “{” începutul corpului constructorului)

Varianța cu lista de inițializare are câteva beneficii:

- poate fi mai eficient (inițializează direct atributul - loc de default constructor apoi copiere)
- poate fi singura modalitate de inițializare (ex atribut referință sau const)



## Constructori generați automat de compilator

### Default constructor – constructor fără parametri

Se generează automat doar dacă nu scriem nici un constructor pentru clasa noastră

### Constructor de copiere – constructor care primește un obiect de același tip (prin referință) și creează un obiect nou care este copia parametrului

Constructor folosit când se face o copie a obiectului

- la declarare de variabilă cu inițializare
- la transmitere de parametri (prin valoare)
- când se returnează o valoare dintr-o metodă

```
class Persoana {
    string nume;
    int varsta;
public:
    //constructor default
    Persoana() {
    }
    //constructor de copiere
    Persoana(const Persoana& ot) : nume{ ot.nume }, varsta{ ot.varsta } {
    }
};
```

Constructorul de copiere în general se generează automat. Implementarea default face o copie pentru fiecare atribut al clasei. În cazul în care clasa are atribute alocate dinamic, dacă are pointeri sau necesită o logică specială la copiere atunci trebuie oferit o implementare custom.

= **default** Regulile legate de când se generează automat constructor de copiere / constructor default pot fi surprinzătoare – se poate cere explicit generarea automată

```
class Persoana {
    string nume;
    int varsta;
public:
    //constructor default
    Persoana() = default;
    //constructor de copiere
    Persoana(const Persoana& ot) = default;
};
```

= **delete** În unele cazuri dorim să împiedicăm crearea de copii pentru obiecte

```
class GRASPController {
private:
    vector<Persoana> all;
public:
    GRASPController() = default;
    //nu se mai pot copia obiectele de tip GRASPController
    GRASPController(const GRASPController& ot) = delete;
};
```

## *C++ Core Guidelines – Class*

- [C.1: Organize related data into structures \(structs or classes\)](#)
- [C.2: Use `class` if the class has an invariant; use `struct` if the data members can vary independently](#)
- [C.3: Represent the distinction between an interface and an implementation using a class](#)
- [C.4: Make a function a member only if it needs direct access to the representation of a class](#)
- [C.5: Place helper functions in the same namespace as the class they support](#)
- [C.7: Don't define a class or enum and declare a variable of its type in the same statement](#)
- [C.8: Use `class` rather than `struct` if any member is non-public](#)
- [C.9: Minimize exposure of members](#)

## Obiecte ca parametri de funcții

Se folosește **const** pentru a indica tipul parametrului (in/out,return).

Dacă obiectul nu-și schimbă valoarea în interiorul funcției, el va fi apelat ca parametru **const**

<pre>/**  * Copy constructor  */ Rational(const Rational &amp;ot);</pre>	<pre>Rational::Rational(const Rational &amp;ot) {     a = ot.a;     b = ot.b; }</pre>
--	---

Folosirea **const** permite definirea mai precisă a contractului dintre apelant și metodă  
Oferă avantajul că restricțiile impuse se verifică la compilare (eroare de compilare dacă încercăm să modificăm valoarea/adresa)

Putem folosi **const** pentru a indica faptul ca metoda nu modifică obiectul (se verifică la compilare)

<pre>/**  * Get the <u>nominator</u>  */ int getUp() const; /**  * get the denominator  */ int getDown() const;</pre>	<pre>/**  * Get the <u>nominator</u>  */ int Rational::getUp() const {     return a; } /**  * get the denominator  */ int Rational::getDown() const {     return b; }</pre>
---	---

## Folosirea calificativului **const** - Const Correctness

### Transmitere de parametri/valoare de retur

Pentru clasele definite de noi se aplica același reguli ca și la tipuri built in:

- transmitere prin referință (pas by reference – folosind tipul referință sau pointeri)
- transmitere prin valoare (pass by value – se transmite o copie a obiectului – se creează copie folosind copy constructor)
- Dacă returnăm un obiect prin valoare – se face o copie (copy constructor)

Putem folosi **const** pentru a descrie mai exact efectul funcției asupra parametrilor transmiși.

Dacă parametru nu este modificat în funcție ar trebui folosit **const**

```
/**                                     Rational::Rational(const Rational &ot) {
 * Copy constructor                     a = ot.a;
 *                                     b = ot.b;
 */                                     }
Rational(const Rational &ot);
```

**const** permite descrierea mai precisă a contractului (interfața) între metoda și cel care apelează metoda (codul client)

Restricțiile impuse sunt verificate în timpul compilării (eroare de compilare dacă încercăm să modificăm un obiect transmis prin **const**)

Ar trebui folosit **const** pentru a indica faptul că o metodă a unei clase nu modifică starea obiectului (query method).

```
/**
 * Get the nominator
 */
int getUp() const;
/**
 * get the denominator
 */
int getDown() const;

/**
 * Get the nominator
 */
int Rational::getUp() const {
    return a;
}
/**
 * get the denominator
 */
int Rational::getDown() const {
    return b;
}
```

Dacă într-o funcție avem un parametru formal **const&**, funcția nu poate modifica starea obiectului => poate apela doar metode **const** al obiectului

## Supraîncărcarea operatorilor

C++ permite definirea semanticii pentru operatori asupra tipurilor definite de utilizator.

ex. ce se întâmplă dacă scriu `a+b` unde `a, b` sunt obiecte de un tip definit de utilizator

```
/**
 * Compute the sum of 2 rational numbers
 * a,b rational numbers
 * rez - a rational number, on exit will contain the sum of a and b
 */
void add(const Rational &nr);
/**
 * Overloading the + to add 2 rational numbers
 */
Rational operator +(const Rational& r) const;
/**
 * Sum of 2 rational number
 */
void Rational::add(const Rational& nr1) {
    a = a * nr1.b + b * nr1.a;
    b = b * nr1.b;
    int d = gcd(a, b);
    a = a / d;
    b = b / d;
}

/**
 * Overloading the + to add 2 rational numbers
 */
Rational Rational::operator +(const Rational& r) const {
    Rational rez = Rational(this->a, this->b);
    rez.add(r);
    return rez;
}
```

Lista operatorilor pe care putem să definim:

`+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`, `%`, `%=`, `++`, `--`, `=`, `==`, `<>`, `<=`, `>=`, `!`, `!=`, `&&`, `||`, `<<`, `>>`, `<<=`, `>>=`, `&`, `^`, `|`, `&=`, `^=`, `|=`, `~`, `[]`, `.,` `()`, `->*`, `→`, `new`, `new[]`, `delete`, `delete[]`,

## Operatorul de assignment (=)

Operatorul = (copy assignment operator) înlocuiește conținutul unui obiect a cu o copie a obiectului b (b nu se modifica)

```
Rational& operator=(const Rational& ot) {
    this->a = ot.a;
    this->b = ot.b;
    return *this;
}

Rational& operator=(const Rational& ot) = default;

Rational r1{ 2,3 };
Rational r2;
r2 = r1;
r2.operator=(r1); //acelasi lucru cu linia de mai sus
```

Daca nu definim operatorul de assignment într-o clasa, compilatorul generează o varianta implicita. In cazul in care clasa gestionează memorie alocata dinamic, varianta implicita nu este corecta (nu funcționează cum ne-am aștepta)

```
class Pet {
private:
    char* nume;
    int varsta;
public:
    Pet& operator=(const Pet& ot) {
        if (this == &ot) {
            //sa evitam problemele la a = a;
            return *this;
        }
        char* aux = new char[strlen(ot.nume)+1];
        strcpy(aux, ot.nume);
        //eliberam memoria ocupata
        delete[] nume;
        nume = aux;
        varsta = ot.varsta;
        return *this;
    }
    Pet(const char* n, int varsta) {
        nume = new char[strlen(n) + 1];
        strcpy(nume, n);
        this->varsta = varsta;
    }
}
```

## Abstract datatype – ascunderea reprezentării

Folosind în C struct/module/funcții nu putem defini tipuri abstracte de date care sunt ușor de folosit corect (cel care folosește TAD-ul nostru trebuie să ia în considerare aspecte ce țin de implementare, compilatorul nu ne e de mare ajutor – Ex. nu da eroare dacă uit să apelez funcția `creazaPet`).

Modificând struct `Pet` de la versiunea 1 la versiunea 2 necesită schimbări în întreaga aplicație

Pentru a gestiona corect memoria este nevoie de funcții ajutoare de alocare/deallocare și trebuie să le folosim consistent aceste funcții în toată aplicația. Aceste funcții trebuie folosite peste tot dar sunt doar convenții (reguli pe care trebuie să le ținem minte) de care trebuie să ținem cont, nu sunt urmărite/forțate de compilator.

Dacă folosim clase și ascundem detaliile de implementare (câmpuri private) putem schimba orice detaliu de implementare (reprezentare) fără a schimba codul care folosește clasa.

## Șiruri de caractere in C++

Clasa string este parte biblioteca standard C++

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string s; //create empty string
    //cin >> s;//read a string
    //cout << s;
    getline(cin, s, '\n');//read entire line
    cout << s;
    for (auto c : s) { //iterate characters
        cout << c<<" ";
    }
    string s2{ "asdasd" }; //create string
    s2 = s2 + s; //concatenate strings
    const char* pc = s2.c_str(); //transform to C-String
    return 0;
}
```

```
#include <iostream>
#include <string>
class Persoana {
    std::string nume;
    std::string prenume;
public:
    Persoana(const std::string& n, const std::string& pn)
        :nume{ n }, prenume{ pn } {
    }

    std::string getNume() {
        return this->nume;
    }
};
int main()
{
    Persoana p{ "Ion", "Ion" };
    std::cout << p.getNume();
    return 0;
}
```



## Vector dinamic in C++

Clasa vector face parte din biblioteca standard C++.

Este o lista implementata folosind structura de data vector dinamic.

Este o clasa parametrizata (template), se specifica tipul elementelor continute.

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

int main() {
    //create si initializare vector
    vector<int> v{ 1,2,3,5,78,2 };
    v.push_back(8); //adauga element
    cout << v[4]<<'\n'; //access element dupa index
    //iterare elemente
    for (int el : v) {
        cout << el << ",";
    }
    cout<< '\n' << v.back() << '\n'; //tipareste ultimul element
    v.pop_back(); //sterge ultimul element
    int first = v.front(); //returneaza primul element
    return 0;
}
```

## Template

- permite crearea de cod generic
- in loc sa repetam implementarea unei funcției pentru fiecare tip de date, putem crea o funcție parametrizată după una sau mai multe tipuri
- o metoda convenienta de reutilizare de cod si de scriere de cod generic
- codul C++ se generează automat înainte de compilare, înlocuind parametru template cu tipul efectiv.

Function template:

```
template <class identifier> function_declaration;
```

or

```
template <typename identifier> function_declaration;
```

<pre>int sum(int a, int b) {     return a + b; }  double sum(double a, double b) {     return a + b; }</pre>	<pre>template&lt;typename T&gt; T sum(T a, T b) {     return a + b; }  int sum = sumTemp&lt;int&gt;(1, 2); cout &lt;&lt; sum; double sum2 = sumTemp&lt;double&gt;(1.2, 2.2); cout &lt;&lt; sum2;</pre>
--	--

- T este parametru template (template parameter), este un tip de date , argument pentru funcția sum
- Instantierea template-ului → crearea codului efectiv înlocuind T cu tipul **int**:
- **int** sum = sumTemp<int>(1, 2);

### Class template:

Putem parametriza o clasa după unu sau mai multe tipuri

Template-ul este ca o matriță, înlocuind parametrul template cu un tip de date se obține codul c++, în acest caz o clasa.

Când cream o clasa template tot codul trebuie pus în header (fișierul .h). Implementarea actuală se creează la compilare. Compilatorul înlocuiește (textual) tipul dat ca parametru template și generează partea de implementare.

```
template<typename ElementType>
class DynamicArray {
public:
    /**
     * Add an element to the dynamic array to the end of the array
     * e - is a generic element
     */
    void addE( ElementType r);
    /**
     * Access the element from a given position
     * poz - the position (poz>=0;poz<size)
     */
    ElementType& get(int poz);
    /**
     * Give the size of the array
     * return the number of elements in the array
     */
    int getSize();
    //.....
private:
    ElementType *elems;
    int capacity;
    int size;
};
/**
 * Add an element to the dynamic array
 * r - is a rational number
 */
template<typename ElementType>
void DynamicArray<ElementType>:: addE( ElementType r){
    ensureCapacity(size + 1);
    elems[size] = r;
    size++;
}
//.....
```

### Atribute statice in clasa (campuri/metode).

Atributele statice dintr-o clasa aparțin clasei nu instanței (obiectelor)

Caracterizează clasa, nu face parte din starea obiectelor

Ne referim la ele folosind operatorul scope "::"

Sunt asemănătoare variabilelor globale doar ca sunt definite in interiorul clasei – retine o singura valoare chiar daca am multiple obiecte

Keyword : **static**

```
/**                                     Rational::nrInstances
 * New data type to store rational
numbers
 * we hide the data representation
 */
class Rational {
public:
    /**
     * Get the nominator
     */
    int getUp();
    /**
     * get the denominator
     */
    int getDown();
private:
    int a;
    int b;
    static int nrInstances = 0;
};
```

### Clase/funcții **friend**.

- **friend** permite accesul unei funcții sau clase la câmpuri/metode private dintr-o clasă
- O metodă controlată de a încălca încapsularea
- punând declarația funcției precedată de **friend** în clasă, funcția are acces la membrii privați ai clasei
- Funcția **friend** nu este membră a clasei (nu are acces la `this`), are doar acces la membrii privați din clasă
- O clasă B este **friend** cu class of class A dacă are acces la membri privați ai lui A. Se declară clasa cu cuvântul rezervat **friend** în fața.

### Clasa friend

<pre>class ItLista { public:     friend class Lista; ... </pre>	<pre>template&lt;typename E&gt; class Set {     friend class Set_iterator&lt;E&gt; ; </pre>
---	---

### Funcție friend

```
class List {
public:
    friend void friendMethodName(int param);

```

### Când folosim **friend**

putem folosi la supraîncărcarea operatorilor:

```
class AClass {
private:
    friend ostream& operator<<(ostream& os, const AClass& ob);
    int a;
};
ostream& operator<<(ostream& os, const AClass& ob) {
    return os << ob.a;
}
```

Util și pentru:

```
class AClass {
public:
    AClass operator+(int nr); //pentru: AClass a; a+7
private:
    int a;
    friend AClass operator+(int nr, const AClass& ob); //pentru: AClass a;
    7+a
};
```