Clase si obiecte in C++

Clasa este un tip de data definit de programator care grupeaza

- **atribute** (date)
- **metode** (comportament)

Clasele sunt definite in fisiere header (.h)
Implementarea metodelor sunt puse in fisiere (.cpp)

rational.h

```
//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;
};
```

rational.cpp

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

Pentru a indica apartenenta metodei la clasa se foloseste oferatorul ::
Similar ca si la module se separa declaratia (interfata) de implementare

Se pot defini metode direct in fisierul 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;
```

Se pot folosi metode inline doar pentru metode simple

Obiect

Clasa - un nou tip de data.
Obiectul - o instanta noua de tipul descris de clasa.

Declaratie de obiecte

<class_name> <object_name>

- se aloca memorie pentru a stoca o valoare de tipul <class_name>
- obiectul se initializeaza folosind constructorul implicit (fara parametrii)
- pentru initializare putem folosi constructori cu parametri

Acces la campuri

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

Cand implementam metodele din clasa avem acces la toate atributele clasei

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

Putem accesa campul clasei si folosind pointerul this. Este util daca mai avem alte variabile cu acelasi nume in metoda (parametru, variabila locala)

this : pointer la instanta curenta. Avem acces la acest pointer in toate metodele clasei.

Se pot accesa campurile si in afara clasei daca sunt vizibile folosind :

- operatorul ' . ' , exemplu : object.field
- operatorul ' -> ' daca avem o referinta la obiect, exemplu : object_reference->field sau (*object_reference).field

Protectia atributelor si metodelor

Pentru a proteja atributele si metodele unei clase se folosesc **modificatori de acces** :

- **public** : poate fi accesat de oriunde
- **private** : poate fi accesat doar in interiorul clasei

Atributele se declara private si se folosesc functii pentru a se putea accesa (getter / setter)

```
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

Constructorul este metoda speciala folosita pentru initializarea obiectelor. Metoda este apelata atunci cand se creeaza instante noi si aloca memorie pentru datele membre si initializeaza atributele.

Pentru a creea un constructor scriem o functie publica a carei nume coincide cu numele clasei si care nu are tip returnat.

```
class Rational {
public:
	Rational();
private:
	//fields (members)
	int a;
	int b;
};
```

```
Rational::Rational() {
	a = 0;
	this->b = 1;
}
```

Orice clasa are cel putin un constructor (daca nu se declara unul atunci exista un constructor default).

Intr-o clasa putem avea mai multi constructori, constructorul poate sa aiba si parametrii.

```
class Rational {
public:
	Rational();
	Rational(int a,int b);
private:
	int a;
	int b;
};
```

```
Rational::Rational() {
	a = 0;
	this->b = 1;
}
Rational::Rational(int a,int b):a{a},b{b} {
}
```

```
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 initializa adaugand o lista inainte de corpul metodei. Exemplu mai sus subliniat cu galben.

Constructori generati automat de compilator

Default constructor este constructorul fara parametrii. Se genereaza automat doar daca nu scriem nici un constructor pentru clasa noastra.

Constructor de copiere este constructorul care primeste un obiect de acelasi tip prin referinta si creeaza un obiect nou care este copia parametrului. Acest constructor este folosit cand se face o copie a obiectului.

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

Construcotul de copiere se genereaza automat. Implementarea default a acestui constructor face o copie pentru fiecare atribut al clasei. In cazul in care clasa are atribute alocate dinamic, daca are pointeri sau necesita o logica speciala la copiere atunci trebuie realizata o implementare custom a acestui constructor.

Obiecte ca parametrii de functii

Se foloseste const pentru a indica tipul parametrului (in / out). Daca obiectul nu isi schimba valoarea in interiorul functiei el va fi apelat ca parametru const.

```
/**
* Copy constructor
*
*/
Rational(const Rational &ot);
```

```
Rational::Rational(const Rational &ot) {
	a = ot.a;
	b = ot.b;
}
```

Folosirea const ofera avantajul ca restrictiile impuse se verifica la compilare (eroare de compilare daca incercam sa modificam valoarea/adresa).

Putem folosi const pentru a indica faptul ca metoda nu modifica obiectul (se verifica la compilare).

```
/**
 * 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;
}
```

Folosirea calificativului const - Transmiterea de parametri / valoare de retur

Pentru clasele definite de noi se aplica aceleasi reguli :

- transmitere prin referita (folosind tipul referinta sau pointeri)
- transmitere prin valoare (se transmite o copie a obiectului folosind copy constructor)
- daca returnam un obiect prin valoare se face o copie a lui (folosind copy constructor)

Putem folosi const pentru a descrie efectul unei functii asupra parametrilor transmisi, de exemplu daca parametrul nu este modificat in functie ar trebui sa folosim const. Exemplu :

```
/**
* Copy constructor
*
*/
Rational(const Rational &ot);
```

```
Rational::Rational(const Rational &ot) {
        a = ot.a;
        b = ot.b;
}
```

Restrictiile impuse sunt verificate in timpul compilarii (eroare de compilare daca incercam sa modificam).

Ar trebui folosit const pentru a indica faptul ca o metoda a unei clase nu modifica starea obiectului.

```
/**
 * 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;
}
```

Cand o functie primeste un obiect prin referinta constanta (const&), ea nu face o copie, ci doar acceseaza obiectul fara drept de modificare, se pot apela doar metodele constante ale obiectului.

```
class Test {
public:
    int val;

    void modifica() { val = 100; }   // ❌ NU poate fi apelată în funcții cu `const &`
    int citeste() const { return val; }  // ✅ Poate fi apelată, nu modifică nimic
};

void verifica(const Test& obj) {
    obj.citeste();  // ✅ OK
    // obj.modifica();  ❌ EROARE: funcția nu poate modifica `obj`
}
```

Supraincarcarea operatorilor

C++ permite definirea semanticii pentru operatori asupra obiectelor, lista de operatori pe care putem sa ii definim : +, -, *, /, +=, -=, *=, /=, %, %=, ++, –, =, ==, < >, <=, >=, !, !=, &&, ||, <<, >>, <<=, >>=, &, ^, |, &=, ^=, |=, ~, [] ,, () , ->*, →, new , new[] , delete, delete[]

```
class Numar {
public:
    int valoare;

    // Constructor cu inițializare prin acolade
    Numar(int v) : valoare{v} {}

    // Supraîncărcarea operatorului +
    Numar operator+(const Numar& other) const {
        return Numar{valoare + other.valoare};
    }
};
```

Operatorul de asignare =

Operatorul = inlocuieste continutul unui obiect X cu o copie a obiectului Y.

```
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 intr-o clasa, compilatorul va genera o varianta implicita. In cazul in care clasa gestioneaza memorie alocata dinamic varianta implicita nu este corecta.
Exemplu :

```
class Pet {
private:
      char* nume;
      int varsta;
public:
   Pet& operator=(const Pet& ot) {
      if (this == &ot) {
       //sa evitam problemele la a = a;
            return *this; //verifica auto-atribuirea, daca este cazul atunci
                          returneaza obiectul curent fara sa faca nimic
      }
      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;
   }
```

Siruri de caractere in C++

Clasa string este parte a bibliotecii 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 caracters
            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 este parte din biblioteca standard C++. Este o lista implementata folosind structura de vector dinamic.

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

int main() {
	//creare 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

- il folosim pentru a creea cod generic.
- in loc sa rescriem aceeasi implementare a unei functii pentru un tip de date diferit, putem creea o functie parametrizata dupa una sau mai multe tipuri.
- codul C++ se genereaza automat inainte de compilare, inlocuind parametrul template cu tipul efectiv.

Function template :

template <class identifier> function_declaration

or

template <typename identifier> function_declaration

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

template <typename T> T sum (T a, T b) {
    return a + b;
}

int main () {
    cout << sum <int> (a: 1,  b: 2) << endl;
    cout << sum <double> (a: 2.5,  b: 3.7) << endl;
}
```

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

int sum (int a, int b) {
    return a + b;
}

double sum (double a, double b) {
    return  a + b;
}

int main () {
    cout << sum (a: 1,  b: 2) << endl;
    cout << sum (a: 2.5,  b: 3.7) << endl;
}
```

T este parametrul template, este un tip de date, argument pentru functia sum.

Class template

Putem sa cream o clasa dupa unul sau mai multe tipuri folosind template-ul care inlocuieste parametrul template cu un tip de date. Cand cream o clasa template tot codul trebuie pus in header deoarece la compilare, atunci cand se vede concret ce tip (ElementType) este folosit, pentru a se putea face instantierea sablonului, trebuie ca compilatorul sa poata accesa implamentarea completa a metodelor.

```
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 , keyword : static

Atributele statice dintr-o clasa apartin clasei, nu instantei (obiectelor), acestea caracterizeaza clasa, nu fac parte din starea obiectelor.
Ne referim la ele folosind operatorul " :: ".
Sunt asemanatoare variabilelor globale doar ca sunt definite in interiorul clasei si retin o singura valoare chiar daca exista mai multe obiecte.

```
/**                                    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 / functii friend

- membrii privati a unei clase sunt inaccesibili din afara clasei. Dar uneori exista functii externe sau alte clase care au nevoie de acces intern, fara a fi metode ale clasei.
- friend permite accesul unei functii sau altei clase sa aiba acces direct la membrii privati ai unei clase chiar daca nu face parte din acea clasa.

```cpp
class A;

class B {
public:
    void showA(A a);
};

class A {
private:
    int value = 10;

    // Declară doar metoda lui B ca prietenă
    friend void B::showA(A a);
};

void B::showA(A a) {
    cout << "Value from A: " << a.value << endl;
}
```