Curs 3 – Clase și obiecte

- **Alocare dinamica - TAD Lista**
- **Limbajul de programare C++**
- **Programare orientată obiect**

Curs 2 – Programare modulară în C

- **Funcții – Test driven development, Code Coverage**
- **Module – Programare modulara, TAD**
- **Gestiunea memoriei in C/C++**

Gestiunea memoriei in C / C++

Memoria ocupată de o aplicație in timpul rulării este împărțita pe segmente de memorie:

Segment	Ce conține	Gestiune in C/C++
Stack (call stack)	Variabile locale, transmitere parametrii, Informații despre funcții care au fost apelate dar încă nu sau terminat.	Gestionat automat si eficient. Structura de stiva (LIFO). La fiecare apel de metoda se pune pe stivă un Stack Frame La terminarea metodei se elimină ultimul stack frame.
Heap (free segment, dynamic memory)	Variabile alocate dinamic	Gestionat manual de programator folosind: `malloc`, `calloc`, `realloc`, `free`.
Data segment (initialized data segment)	Variabile globale si statice inițializate din program	
Bss segment (uninitialized data segment)	Variabile globale si statice neinițializate (memorie inițializată cu 0)	
Code segment (text segment)	Instrucțiuni cod mașina (programul compilat)	Read only in general

Stack vs Heap

Stack avantaje:

- Gestionat automat (domeniu de vizibilitate/ciclu de viată pentru variabile)
- Structura de date tip stiva – LIFO – Last in first out
- Eficient
    - se gestionează doar un pointer la capul stivei (operații simple de aritmetica de pointeri)
    - Zona compacta de memorie, frecvent in memoria cache al procesorului (L1,L2,L3 cache)
    - Fiecare fir de execuție are stack propriu (nu este nevoie de sincronizare)

Stack dezavantaje:

- Memoria pentru variabilele de pe stiva este eliberata la terminarea funcției (blocului {}) in care am declarat variabila
- Dimensiune limitata (default 1Mb pe windows)
- Memoria aferenta valorilor din stack in general trebuie specificat (cunoscut) la compilare

Heap avantaje:

- Permite alocarea memoriei in timpul rulării programului.
- Putem avea memorie alocata fără a fii dealocata automat la terminarea funcției (blocului{})
- Dimensiunea memoriei este specificat dinamic (valori cunoscute doar in timpul execuției)
- Permite definirea de structuri de date mai complicate (înlănțuire, arbore, graf)
- Dimensiune mult mai mare (limitata doar de sistemul de operare)

Heap dezavantaje:

- Gestiunea (alocare/de-alocare) este făcut de programator. Se poate ușor ajunge la memory leak (memorie alocata dar niciodată eliberata).
- Ineficient (comparativ cu stack)
    - folosește o structura interna de gestiune mult mai complicata decât o stiva.
    - Este partajat intre firele de execuție ale programului
    - Poate conduce la fragmentarea memoriei

Alocare dinamica

Folosind funcțiile **malloc**(size) și **free**(pointer) programatorul poate aloca/elibera memorie pe Heap – zonă de memorie gestionat de programator

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
	//allocate memory on the heap for an int
	int *p = malloc(sizeof(int));

	*p = 7;
	printf("%d \n", *p);
	//Deallocate
	free(p);
    //allocate space for 10 ints (array)
	int *t = malloc(10 * sizeof(int));
	t[0] = 0;
	t[1] = 1;
	printf("%d \n", t[1]);
	//dealocate
	free(t);
	return 0;
}
/**
 * Make a copy of str
 * str - string to copy
 * return a new string
 */
char* stringCopy(char* str) {
	int len = strlen(str) + 1; // +1 for the '\0'
	char* newStr = malloc(sizeof(char) * len); // allocate memory
	strcpy(newStr, str); // copy string
	return newStr;
}
```

Programatorul este responsabil cu dealocarea memoriei
OBS: Pentru fiecare `malloc` trebuie sa avem exact un `free`

Memory management

void* malloc(int num); - alocă **num** byte de memorie, memoria este neinițializată

void* calloc(int num, int size); - alocă **num***size** memorie, inițializează cu 0

void* realloc(void* address, int newsize);-redimensionare memorie alocată

void free(void* address); - eliberează memoria (este disponibila pentru următoarele alocări)

Memory leak

Programul aloca memorie dar nu dealoca niciodată, memorie irosită

```
int main() {
	int *p;
	int i;
	for (i = 0; i < 10; i++) {
		p = malloc(sizeof(int));
		//allocate memory for an int on the heap
		*p = i * 2;
		printf("%d \n", *p);
	}
	free(p); //deallocate memory
	//leaked memory - we only deallocated the last int
	return 0;
}
```

Memory leak detection

Memory leak – memorie care nu a fost dealocata. Alocat pe heap (cu malloc) dar niciodată dealocat (free).

Memory leak detection – identificarea unui Memory leak.
Chiar dacă zona de memoria ne-dealocata este mica, in timp aceste mici zone se aduna si cauzează probleme serioase (Out of memory error).
Cu cat complexitatea aplicației este mai mare, cu atât găsirea acestor probleme este mai dificilă.

Există diferite instrumente care asistă programatorul in găsirea de Memory Leak.
In Visual Studio putem folosi CRT Library:
https://msdn.microsoft.com/en-us/library/x98tx3cf(v=vs.140).aspx

Adăugați in programul vostru, la început in aceasta ordine:

```
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>
```

Unde vreți sa vedeți daca aveți memory leak (in general la sfârșitul funcției main):

```
_CrtDumpMemoryLeaks();
```

Metoda `_CrtDumpMemoryLeaks()` tipărește toate alocările pentru care nu s-a executat free()

void*

O funcție care nu returnează nimic

```
void f() {
}
```

Nu putem avea variabile de tip `void` dar putem folosi pointer la void - `void*`

```
#include <stdio.h>
#include <stdlib.h>

int main() {
      void* p;
      int *i=malloc(sizeof(int));
      *i = 1;
      p = i;
      printf("%d /n", *((int*)p));
      long j = 100;
      p = &j;
      printf("%ld /n", *((long*)p));
        free(i);
      return 0;
}
```

Se pot folos `void*` pentru a crea structuri de date care funcționează cu orice tip de elemente

Probleme: verificare egalitate între elemente de tip `void*` , copiere elemente

TAD – Tip abstract de date (ADT – Abstract data types)

TAD:

- Definește domeniul de valori
- Definește operațiile posibile (interfața)
- Definiția operațiilor este independent de implementare (abstract)
- Ascunde implementarea.

TAD implementat in C

<adt.h> + <adt.c>
interfața implementare

Interfața (header) conține declarații de funcții, fiecare funcție este specificata independent de implementare.

Codul client (cel care folosește TAD) nu are acces la detalii de implementare

Putem folosi diferite structuri de date pentru implementare

Este valabil pentru orice concept implementat in cod: separam interfața de detalii de implementare.

Vector dinamic

```c
typedef void* Element;

typedef struct {
	Element* elems;
	int lg;
	int capacitate;
} VectorDinamic;

/**
 *Creaza un vector dinamic
 * v vector
 * post: vectorul e gol
 */
VectorDinamic * creazaVectorDinamic();


/**
 *Initializeaza vectorul
 * v vector
 * post: vectorul e gol
 */
VectorDinamic * creazaVectorDinamic() {
  VectorDinamic *v =
malloc(sizeof(VectorDinamic));
  v->elems = malloc(INIT_CAPACITY *
sizeof(Element));
  v->capacitate = INIT_CAPACITY;
  v->lg = 0;
  return v;
}
/**
 * Elibereaza memoria ocupata de vector
 */
void distruge(VectorDinamic *v) {
	int i;
	for (i = 0; i < v->lg; i++) {
//!!!!functioneaza corect doar daca
//elementele din lista NU refera
// memorie alocata dinamic
		free(v->elems[i]);
	}
	free(v->elems);
	free(v);
}
```

```c
/**
 * Adauga un element in vector
 * v - vector dinamic
 * el - elementul de adaugat
 */
void add(VectorDinamic *v, Element el);

/**
 *Returneaza elementul de pe pozitia data
 * v - vector
 * poz - pozitie, poz>=0
 * returneaza elementul de pe pozitia poz
 */
Element get(VectorDinamic *v, int poz);


/**
 * Aloca memorie aditionala pentru vector
 */
void resize(VectorDinamic *v) {
  int nCap = 2*v->capacitate;
  Element* nElems=
malloc(nCap*sizeof(Element));
	//copiez din vectorul existent
  int i;
  for (i = 0; i < v->lg; i++) {
		nElems[i] = v->elems[i];
  }
  //dealocam memoria ocupata de vector
  free(v->elems);
  v->elems = nElems;
  v->capacitate = nCap;
}
/**
 * Adauga un element in vector
 * v - vector dinamic
 * el - elementul de adaugat
 */
void add(VectorDinamic *v, Element el) {
	if (v->lg == v->capacitate) {
		resize(v);
	}
	v->elems[v->lg] = el;
	v->lg++;
}
```

Pointer la funcții

```
void(*funcPtr)(int);  // function returns void has an int parameter
int(*funcPtr2)(int,int); // function returns int has two int
parameters
```

```
void f(int a) {
      printf("%d\n", a);
}
```

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

```
int main() {
  void(*funcPtr)(int);
  funcPtr = f;
  funcPtr(6);
  return 0;
}
```

```
int main() {
   int(*funcPtr2)(int,int);
   funcPtr2 = sum;
   int c = funcPtr2(6,3);
   printf("%d\n", c);
   return 0;
}
```

```
void func() {
      printf("func() called...");
}
int main() {
      void (*fp)(); // Define a function pointer
      fp = func; // Initialise it
      (*fp)(); // Dereferencing calls the function
      void (*fp2)() = func; // Define and initialize
      (*fp2)(); // call
}
```

Vector dinamic generic – pointer la funcții

Pentru o lista generica (funcționează cu orice fel de elemente) putem avea nevoie diferite funcții generice care oferă diferite operații pe elementele respective

Putem folosi pointer la funcții în structurile de date generice

```
typedef elem (*CopyFctPtr)(elem);

typedef int (*EqualsFctPtr)(elem, elem);
```

```
typedef void* ElemType;
//function type for dealocating an element
typedef void(*DestroyF)(ElemType);

typedef struct {
	ElemType* elems;
	int lg;
	int capacitate;
} MyList;
```

```
/*
Dealocate list
*/
void destroy(MyList* l, DestroyF dealocate) {
	//free elements
	for (int i = 0; i < l->lg; i++) {
		dealocate(l->elems[i]);
	}
	//free list
	free(l->elems);
	l->lg = 0;
}
```

```
void testCopyList() {
	MyList l = createEmpty();
	add(&l, createPet("a", "b", 10));
	add(&l, createPet("a2", "b2", 20));
	MyList l2 = copyList(&l);
	assert(size(&l2) == 2);
	Pet* p = get(&l2, 0);
	assert(strcmp(p->type, "a") == 0);
	destroy(&l, destroyPet);
	destroy(&l2, destroyPet);
}
```

Urmașul limbajului C apărut în anii 80, dezvoltat de **Bjarne Stroustrup**
Bibliografie:

B. Stroustup, The C++ Programming Language
B. Stroustup, A Tour of C++

ISO standard din 1998 – isocpp.com

Limbajul C++

- compatibil cu C
- multiparadigmă, suportă paradigma orientat obiect (clase, obiecte, polimorfism, moștenire)
- tipuri noi – bool, referință
- spații de nume (namespace)
- șabloane (templates)
- excepții
- bibliotecă de intrări/ieșiri (IO Streams)
- STL (Standard Template Library)

Evoluție:

C with Classes (1979 Aduce concepte din Simula: clase, clase derivate)
C++ (1983 funcții virtuale, supraîncărcarea operatorilor)
C++ 98 (devine standard ISO – clase abstracte, metode statice/const)
C++ 11 (C++ 0x – auto, lambda, rvalue, move, constexpr, etc)
C++ 14 (C++ 1y – fix/upgrade template, lambda, constexpr, etc)
C++ 17 (C++ 1z – string_view, optional, file system, etc)
C++ 20 – standardul curent (module, corutine,stl2)
C++ 23 – in procesul de standardizare

Tipuri de date predefinite

int, long, double, char, bool, void, etc.

Conversii intre tipuri

C++ este un limbaj puternic tipizat, in majoritatea cazurilor este nevoie de o conversie explicita type-casting când dorim sa interpretam o valoare in mod

`char c = 23245;`	Conversie implicita – **de evitat pe cat posibil** poate cauza probleme (owerflow, trunc) In general compilatorul da warning Se poate întâmpla la inițializare, assignment, la transmiterea de parametrii
`int a = static_cast<int>( 7.5);`	Conversie explicita Verificat la compilare Eroare de compilare daca conversia este imposibila (intre tipuri incompatibile)

`char c = (char)2000;`
 	
`//functional notation`
`char c = char(2000);`
	C-style cast. Elimina warningurile cauzate de conversii periculoase
	Este de evitat fiindcă poate cauza probleme mai ales daca tipurile nu sunt compatibile

Tipul bool - domeniu de valori: adevărat (`true`) sau fals (`false`)

```
/** Verifica daca un numar e prim
 * nr numar intreg
 * return true daca nr e prim*/
bool ePrim(int nr) {
     if (nr <= 1)  return false;
     for (int i = 2; i < nr - 1; i++) {
          if (nr % i == 0)
               return false;
     }
     return true;
}
```

data_type &reference_name;

```
int y = 7;
int &x = y; //make x a reference to, or an alias of, y
```

Dacă schimbăm x se schimbă și y și invers, sunt doar două nume pentru același locație de memorie (alias)
Tipul referință este similar cu tipul pointer:

- sunt pointeri care sunt automat dereferențiate când folosim variabile
- nu se poate schimba adresa referită

```
/**
 *  C++ version
 *  Sum of 2 rational number
 */
void sum(Rational nr1, Rational nr2, Rational &rez) {
     rez.a = nr1.a * nr2.b + nr1.b * nr2.a;
     rez.b = nr1.b * nr2.b;
     int d = gcd(rez.a, rez.b);
     rez.a = rez.a / d;
     rez.b = rez.b / d;
}
```

```
/**
 *  C version
 * Sum of 2 rational number
 */
void sum(Rational nr1, Rational nr2, Rational *rez) {
     rez->a = nr1.a * nr2.b + nr1.b * nr2.a;
     rez->b = nr1.b * nr2.b;
     int d = gcd(rez->a, rez->b);
     rez->a = rez->a / d;
     rez->b = rez->b / d;
}
```

Declarare/Inițializare de variabile

Initializare variabile la declarare

`int b { 7 };`
 	
`int c = { 7 };`
	Universal form
	varianta de preferat in modern in C++
	Evită problemele legate de conversii prin care se pierde precizie (narrowing)

`int a = 7;`	Varianta “clasică” moștenită din C
`int d;//gresit`	Varianta greșita, compilează dar variabila este neinițializată (are o valoare aleatoare)

Nu folosiți variabile neinițializate. Preferați varianta cu {}

`auto`

Când definim o variabilă putem sa nu specificam explicit tipul (compilatorul deduce tipul din expresia de inițializare.

```
auto a = 7;//a e int
double b{7.4};
double c{1.4};
auto d = b+c;//d e double
```

`auto` este util pentru:

- a evita scrierea de nume lungi de tipuri
- a evita repetiția
- scriere de cod generic

Const

const semnalează compilatorului ca nu dorim sa schimbam valoarea variabilei

```
const int nr = 100;
```

Daca încercăm sa schimbăm valoarea lui nr rezulta o eroare la compilare

Este util pentru:

- a comunica ce face funcția (descrie mai precis interfața)
  - r1,r2 au fost transmise ca referință (pentru a evita copierea) dar sunt declarate **const** astfel este clar ca aceste valori nu se modifica in interiorul funcției
- compilatorul ajuta la evitarea unor greșeli (compilatorul verifică si dă eroare dacă se încearcă modificarea lui r1 sau r2)
- poate oferi posibilități de optimizare pentru compilator

```
typedef struct{
    int a;
    int b;
} Rationa;

void add(const Rationa& r1, const Rationa& r2, Rationa&
rez){
    ...
}
```

Folosiți **const** pentru a exprima idea de imutabil (nu se modifica)
Folosiți **const** **peste tot** unde are sens:

- compilatorul v-a ajuta in prinderea de bug-uri
- codul este mai ușor de înțeles de alții (codul exprima mai bine intenția programatorului)
- adăugați **const** de la început (e mai greu sa adaugi apoi)

Const Pointer

const type*

```
int j = 100;
const int* p2 = &j;
```

Valoarea nu se poate schimba folosind pointerul. Se poate schimba adresa referită

```
const int* p2 = &j;
cout << *p2 << "\n";
p2 = &i;//change the memory address (valid)
cout << *p2 << "\n";
*p2 = 7;//change the value (compiler error)
cout << *p2 << "\n";
```

type * const

```
int * const p3 = &j;
```

Valoarea se poate schimba folosind acest pointer dar adresa de memorie referită nu se poate schimba

```
int * const p3 = &j;
cout << *p2 << "\n";
//change the memory address (compiler error)
p3 = &i;
cout << *p3 << "\n";
//change the value (valid)
*p3 = 7;
cout << *p3 << "\n";
```

const type* const

```
const int * const p4 = &j;
```

Atât adresa cât și valoarea sunt constante

Range for

```
int a[] = {0, 1, 2, 3, 4, 5};
for (auto v:a) {
     cout << v << "\n";
}
```

```
for (auto v:{0,1,2,3,4,5} ) {
     cout << v << "\n";
}
```

Semantica: Pentru fiecare element din a, de la primul element pană la ultimul, copiază in variabila v;
Range for poate fi folosit cu orice secvența de elemente

Daca vrem sa evitam copierea valorii din vectorul a in variabila v putem folosi **auto**&.
Daca vream sa evitam copierea dar vrem si sa nu modificam elementele **const auto**&

```
for (auto& v : a) {
     ++v;
}
```

```
for (const auto& v : a) {
     cout << v << "\n";
}
```

IO library in C++ - definit in <iostream>

cin - corespunde intrări standard (stdin), tip **istream**

cout – corespunde ieșirii standard (stdout) , tip **ostream**

cerr - corespunde stderr, tip **ostream**

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

void testStandardIOStreams() {
	//prints Hello World!!! to the console
	cout << "Hello World!!!\n";
	int i = 0;
	cin >> i; //read an int from the console
	cout << "i=" << i << "\n"; // printsto the console
	//write a message to the standard error stream
	cerr << "Error message";
}
```

- Operația de scriere se realizează folosind operatorul **“<<“,** insertion operator
- citirea de realizează folosind operatorul “>>“, extraction operator

Paradigma de programare orientată-object

Este metodă de proiectare și dezvoltare a programelor:

- Oferă o abstractizare puternică și flexibilă
- Programatorul poate exprima soluția în mod mai natural (se concentrează pe structura soluției nu pe structura calculatorului)
- Descompune programul într-un set de obiecte, obiectele sunt elementele de bază
- Obiectele interacționează pentru a rezolva problema, există relații între clase
- Tipuri noi de date modelează elemente din spațiul problemei, fiecare obiect este o instanța a unui tip de data (clasă)

Un obiect este o entitate care:

- are o stare
- poate executa anumite operații (comportament)

Poate fi privit ca și o combinație de:

- date (atribute)
- metode

Concepte:

- obiect
- clasă
- metodă (mesaj)

Proprietăți:

- abstractizare
- încapsulare
- moștenire
- polimorfism

Caracteristici:

Încapsulare:

- capacitatea de a grupa date și comportament
    - controlul accesului la date/funcții,
        - ascunderea implementării
        - separare interfață de implementare

Moștenire

- Refolosirea codului

Polimorfism

- comportament adaptat contextului
    - în funcție de tipul actual al obiectului se decide metoda apelată în timpul execuției

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)

Sintaxă:

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

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 in 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 instanța noua (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 parametrii)
- pentru inițializare putem folosi si 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() << "\n";
cout << r2.toFloat() << "\n";
cout << r3.toFloat() << "\n";
```

Acces la atribute (câmpuri)

În interiorul clasei

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

Când implementăm metodele avem acces direct la atribute

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

Putem accesa atributul folosind pointerul **this**. Util daca 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 atributele 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 declara 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

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

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

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)

Intr-o clasă putem avea mai mulți constructori, constructorul poate avea parametrii.

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

Constructor cu parametrii

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

```
Rational r2(1, 3);
```

Constructori - Listă diferită de parametrii

Obiecte ca parametrii 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**

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

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

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 încercam să modificăm valoarea/adresa)

Putem folosi **const** pentru a indica faptul ca metoda nu modifică obiectul (se verifică 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;
}
```

Supraîncărcarea operatorilor.

Definirea de semantică (ce face) pentru operatori uzuali când sunt folosiți pentru tipuri definite 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;
}
```

Operatori ce pot fi supraîncarcați:
+, -, *, /, +=, -=, *=, /=, %, %=, ++, –, =, ==, < >, <=, >=, !, !=, &&, ||, <<, >>, <<=, >>=, &, ^, |, &=, ^=, |=, ~, [] ,, () , ->*, →, new , new[] , delete, delete[],