Curs 3 – Clase și obiecte

- **Limbajul de programare C++**
- **Programare orientată obiect**

Limbajul de programare C++

Urmașul limbajului C apărut în anii 80 (C cu clase), dezvoltat inițial de Bjarne Stroustrup

Bibliografie: B. Stroustup, The C++ Programming Language, Addison Wesley

Limbajul C++

- multiparadigmă
- suportă paradidma 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)

Tipul de date mulțime - modular, implementat in C++

```
/*set.h*/
struct Mult;
typedef Mult* Set;
/**
 * Create an empty set. Need to be invoked before we can use the set
 * n - the maximum number of element in the set; m - the set
 */
void createSet(Set &m, int n);
/**
 * Add an element to the set
 * m - the set; e the element to be added
 * return 0 if we can not add the element
 */
int add(Set m, Element e);
```

```
/*set.cpp*/
#include "set.h"
struct Mult {
	Element* e;
	int c;
	int max;
};
/**
 * Release the memory allocated for this set
 */
void destroyM(Set& m) {
	for (int i = 0; i < m->c; i++)
		destroyE(m->e[i]);
	delete[] m->e;
	delete m;
}
/**
 * Add an element to the set
 * m - the set; e the element to be added
 * return 0 if we can not add the element
 */
int add(Set m, Element e) {
	if (m->c == m->max)
		return 0;
	if (!(contains(m, e))) {
		(m->c)++;
		atrib(e, m->e[(m->c) - 1]);
	}
	return 1;
}
```

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;
}
```

Tipul referință

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 pointere:

- 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;
}
```

Const Pointer

1 `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";
//change the memory address (valid)
p2 = &i;
cout << *p2 << "\n";
//change the value (compiler error)
*p2 = 7;
cout << *p2 << "\n";
```

2 type * `const`

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

Valoarea se poate schimba folosind acest pointer dar adressa 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";
```

3 `const` type* `const`

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

Atât adresa cât și valoarea sunt constante

Paradigma de programare orientată-object

Este metodă de proiectare şi 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 interacționeaza pentru a rezolva problema, există relații între clase
- Tipuri noi de date modeleaza 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 fisier separat (.cpp)
Se foloseste operatorul :: (scope operator) pentru a indica apartanența 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 initializare 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() << endl;
cout << r2.toFloat() << endl;
cout << r3.toFloat() << endl;
```

Acces la atribute (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 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 attributelor 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 cand se crează instanțe noi (se declara o variabilă locală, se crează un obiect folosind **new**)

Numele coincide cu numele clasei, nu are tip returnat
Constructorul alocă memorie pentru datele membre, inițializează attributele

```
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 crează – 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

Constructor de copiere

Constructor folosit când se face o copie a obiectului

- la atribuire
- la transmitere de parametrii (prin valoare)
- când se returnează o valoare dintr-o metodă

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

Există un constructor de copiere implicit (chiar dacă nu se declară în clasă) acesta copiează câmpurile obiectului, dar nu este potrivit mai ales în cazul în care avem attribute alocate dinamic

Alocare dinamică de obiecte

operatorul new se foloseste pentru alocarea de memorie pe heap pentru obiecte

```
	Rational *p1 = new Rational;
	Rational *p2 = new Rational(2, 5);
	cout << p1->toFloat() << endl;
	cout << (*p2).toFloat() << endl;
	delete p1;
	delete p2;
```

Destructor

Destructorul este apelat de fiecare data cand se dealocă un obiect

- dacă am alocat pe heap (new), se apeleaza la delete
- dacă e variabilă statică, se dealoca în momentul în care nu mai e vizibil (out of scope)

```
DynamicArray::DynamicArray() {
	capacity = 10;
	elems = new Rational[capacity];
	size = 0;
}
```

```
DynamicArray::~DynamicArray() {
	delete[] elems;
}
```

Obiecte ca parametrii de funcții

Se foloseste `const` pentru a indica tipul parametrului (in/out,return).
Dacă obiectul nu-și schimbă valoarea în interiorul funcţiei, 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/adressa)

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[],