PROGRAMARE ORIENTATĂ PE OBIECTE

LABORATOR 7

Anul universitar 2023 – 2024

Semestrul II

1 Noțiuni teoretice

1.1 Supraîncărcarea operatorilor unari

Operatorii unari au un singur operand. Iată câteva exemple:

- operatorul aritmetic -, de schimbare a semnului
- operatorul de negare logică !
- operatorii de incrementare (++) și decrementare (--). Aceștia sunt tratați în detaliu în acest articol.

De regulă, operatorii unari sunt prefixați (apar înainte de operand): !n, - 5, ++ x, dar există și situații în care sunt postfixați: i ++.

Următorul exemplu definește operatorul - pentru schimbarea semnului unei fracții.

```
#include <iostream>

using namespace std;

class Fractie{
    private:
        int numarator, numitor;
    public:
        void afiseaza()
        {
            cout << numarator << "/" << numitor << endl;
        }

        // constructor cu parametri default, care combina
        // constructorul implicit cu cel cu parametri
        Fractie(int a = 0, int b = 1)
        {
            // muta semnul numitorului (-) la numarator
            // in cazul in care numitorul este negativ
            if(b < 0)
            {
                a = -a;
                b = -b;
            }
            numarator = a;
            numitor = b;
        }
```

```
        Fractie operator- ()
        {
            Fractie F(- numarator , numitor);
            return F;
        }
};

int main(){
    Fractie X(1 , - 4);
    (-X).afiseaza();
    return 0;
}
```

La rulare, programul de mai sus afișează:

```
1/4
```

1.2 Supraîncărcarea operatorilor binari

Majoritatea operatorilor C++ sunt binari, adică au doi operanzi. Mulți dintre ei ne sunt binecunoscuți: operatorii aritmetici, relaționale, logici, dar sunt și operatori care cer, poate, mai multă atenție și pentru care mecanismul de definire va fi tratat în alte articole: operatorul de atribuire (=), operatorul de indexare ([]), etc.

Pentru a supraîncărca operatori binari, folosim metode sau funcții prietene, astfel:

- dacă primul operand este obiect al clasei, putem folosi fie metode, fie funcții prietene;
- dacă primul operand nu este obiect al clasei pentru care supraîncărcăm operatorul, vom folosi funcții prietene.

Observație: La supraîncărcarea unei operații folosind o metodă a clasei, primul operand este considerat obiectul curent. Al doilea operand este transmis ca parametru și poate fi de același tip cu primul sau de alt tip.

Exemplul următor implementează operațiile de adunare pentru fracții:

- între două fracții – printr-o metodă;
- între o fracție și un întreg – printr-o metodă;
- între un întreg și fracție – printr-o funcție prietenă.

```
#include <iostream>

using namespace std;

class Fractie{
    private:
        int numarator, numitor;
```

```
    public:
        void afiseaza() /// metodă pentru afișarea fractiei
        {
            cout << numarator << "/" << numitor << endl;
        }

        Fractie(int a = 0, int b = 1)
        {
            // muta semnul numitorului (-) la numarator
            // in cazul in care numitorul este negativ
            if(b < 0)
            {
                a = -a;
                b = -b;
            }
            numarator = a;
            numitor = b;
        }

        Fractie operator+ (Fractie F)
        {   /// metodă, operația F + G
            Fractie R;
            R.numarator = numarator * F.numitor + numitor * F.numarator;
            R.numitor = numitor * F.numitor;
            return R;
        }
        Fractie operator+ (int n)
        {   /// metodă, operația F + n
            Fractie R;
            R.numarator = numarator + n * numitor;
            R.numitor = numitor;
            return R;
        }
        friend Fractie operator + (int n, Fractie F)
        {   /// funcție prietenă, operația n + F
            Fractie R;
            R.numarator = F.numarator + n * F.numitor;
            R.numitor = F.numitor;
            return R;
        }
};

int main(){
    Fractie X(1 , 4), Y(2, 3);
    Fractie R = X + Y;
    R.afiseaza();
    R = X + 2;
    R.afiseaza();
    R = 2 + X;
    R.afiseaza();
    return 0;
}
```

Programul de mai sus va afișa:

```
11/12
9/4
9/4
```

1.3 Supraîncărcarea operatorilor relaționali

În C++ există operatorii relaționali (<, >, <=, >=, ==, !=) folosiți pentru a compara datele de tipuri predefinite (int, double, etc.). Ei pot fi supraîncărcați în contextul claselor, pentru a compara obiecte.

Fiind operatori binari, se aplică regulile prezentate în acest articol, iar rezultatul lor este de tip bool sau întreg: true (1), dacă relatia are loc și false (0) dacă relația nu are loc.

Este interesant de constatat că pentru a compara două date, trebuie precizată explicit numai relația de îndeplinit pentru operația <. Ceilalți operatori relationali pot fi implementați pe baza acesteia.

- `A < B /// se implementează explicit`
- `A == B` ⇔ `!(A < B) && !(B < A)`
- `A != B` ⇔ `(A < B) || (B < A)`
- `A <= B` ⇔ `A < B || A == B`
- `A > B` ⇔ `!(A <= B)`
- `A >= B` ⇔ `!(A < B)`

În exemplul de mai jos sunt implementate operatorul < pentru fracții:

- F < G, unde F și G sunt fracții;
- F < n, unde F este fracție, iar n este întreg;
- n < F, unde F este fracție, iar n este întreg.

```
#include <iostream>

using namespace std;

class Fractie{
    private:
        int numarator, numitor;
    public:
        void afiseaza() /// metoda pentru afisarea fractiei
        {
            cout << numarator << "/" << numitor << endl;
        }

        Fractie(int a = 0, int b = 1)
        {
            // muta semnul numitorului (-) la numarator
            // in cazul in care numitorul este negativ
            if(b < 0)
```

```
            {
                a = -a;
                b = -b;
            }
            numarator = a;
            numitor = b;
        }

        friend bool operator < (const Fractie& F , const Fractie& G)
        {   /// metoda, operatia F < G
            return F.numarator * G.numitor < F.numitor * G.numarator;
        }

        /* echivalent cu
        bool operator<(const Fractie& other) const {
            return numarator * other.numitor < numitor * other.numarator;
        }
        */

        friend bool operator < (Fractie& F , int n)
        {   /// F < n
            return F.numarator < F.numitor * n;
        }

        /* echivalent cu
        bool operator<(int n) const {
            return numarator < numitor * n;
        }
        */

        friend bool operator < (int n, Fractie& F)
        {   /// n < F
            return n * F.numitor < F.numarator * 1;
        }
};

int main(){
    Fractie X(1 , 4), Y(2, 3);
    cout << (X < Y) <<endl;
    cout << (Y < X) <<endl;
    cout << (X < 2) <<endl;
    cout << (2 < X) <<endl;
    return 0;
}
```

Programul de mai sus va afișa:

```
1
0
1
0
```

1.4 Supraîncărcarea operatorilor de atribuire

Operatorul de atribuire apare în C++ în forma V = E, unde V este o variabilă (dată de tip left value) iar E este o expresie, sau în forma V op= E, ca prescurtare pentru V = V op E, unde op este un operator binar (+, -, *, etc.).

Am văzut deja că atribuirea (=) este posibilă implicit pentru obiecte de același tip. Ea funcționează corect doar dacă datele membre ale clasei sunt alocate static, deoarece se face bit cu bit.

Este un operator binar, care se implementează ca metodă. Obiectul curent reprezintă primul operand, în timp ce al doilea operand va fi transmis ca parametru.

Rezultatul său este o referință la obiectul curent, cel care se modifică.

În continuare avem un exemplu care tratează atribuirea unei valori pentru un obiect de tip fracție; sunt implementați următorii operatori:

- F = G, unde F și G sunt de tip fracție;
- F = n, unde F este fracție și n este întreg;
- F += G, unde F și G sunt de tip fracție;
- F += n, unde F este fracție și n este întreg;
- similar se pot implementa operatorii -=, *=, /=, etc.

```
#include <iostream>

using namespace std;

class Fractie{
    private:
        int numarator, numitor;
    public:
        void afiseaza()
        {
            cout << numarator << "/" << numitor << endl;
        }

        Fractie(int a = 0, int b = 1)
        {
            // muta semnul numitorului (-) la numarator
            // in cazul in care numitorul este negativ
            if(b < 0)
            {
                a = -a;
                b = -b;
            }
            numarator = a;
            numitor = b;
        }
```

```
        Fractie& operator=(const Fractie& F)
        {
            if (this != &F) {
                this->numarator = F.numarator;
                this->numitor = F.numitor;
            }
            return *this;
        }

        Fractie & operator = (const int & n)
        {
            this->numarator = n;
            this->numitor = 1;
            return * this;
        }

        Fractie & operator += (const Fractie & F)
        {
            int a = numarator * F.numitor + numitor * F.numarator;
            int b = numitor * F.numitor;
            this->numarator = a;
            this->numitor = b;
            return * this;
        }

        Fractie & operator += (const int & n)
        {
            int a = numarator * 1 + numitor * n;
            int b = numitor * 1;
            this->numarator = a;
            this->numitor = b;
            return * this;
        }
};

int main(){
    Fractie X(1 , 4), Y(2, 3);
    X += Y;
    X.afiseaza(); // 11/12

    X = Fractie(1 , 4);
    X.afiseaza(); // 1/4

    X += 2;
    X.afiseaza(); // 9/4

    X = Fractie(1 , 4) , Y = Fractie(2 , 3);
    swap(X , Y);
    X.afiseaza(); // 2/3
    Y.afiseaza(); // 1/4

    X = Fractie(1 , 4) , Y = Fractie(2 , 3);
```

```
    X = (Y += 1);
    X.afiseaza(); // 5/3
    Y.afiseaza(); // 5/3
    return 0;
}
```

1.5 Supraîncărcarea operatorilor de citire și afișare

C++ permite citirea și afișarea datelor de tipuri simple (întreg, real, etc.) cu ajutorul operatorilor >> (de extracție din flux) și << (inserție în flux) aplicați obiectelor cin și cout, sau altor obiecte care permit accesarea unor fluxuri de caractere (din/în fișiere text, din/în stringuri).

Practic, citirea se implementează astfel: in >> n; ea este o operație binară, cu operanzii:

- in – un obiect, instanță a clasei istream – poate fi cin, un obiect de tip ifstream pentru citirea din fișier, etc;
- X – o dată de un tip de bază (de exemplu int) sau un obiect al clasei pentru catre supraîncărcăm operatorul.

Afișarea se implementează similar. Mecanismul de supraîncărcare a operatorilor constă în funcții prietene, deoarece primul operand nu este de obiect al clasei pentru care facem supraîncărcarea, ci de tip istream (pentru citire), respectiv ostream (pentru scriere).

Programul de mai jos implementează operațiile de citire și afișare pentru clasa Fractie.

```
#include <iostream>

using namespace std;

class Fractie{
    private:
        int numarator, numitor;
    public:
        Fractie(int a = 0, int b = 1)
        {
            // muta semnul numitorului (-) la numarator
            // in cazul in care numitorul este negativ
            if(b < 0)
            {
                a = -a;
                b = -b;
            }
            numarator = a;
            numitor = b;
        }

        friend istream & operator >> (istream & in , Fractie & F)
        {
            in >> F.numarator >> F.numitor;
```

```
            return in;
        }

        friend ostream & operator << (ostream & out , const Fractie & F)
        {
            out << F.numarator << "/" << F.numitor;
            return out;
        }
};

int main(){
    Fractie X;
    cin >> X; // 3 5
    cout << X << endl; // 3/5
    return 0;
}
```

În cazul citirii, operatorul >> returnează o referință la parametrul istream & in pentru a putea folosi citiri "înlănțuite", de forma cin >> X >> Y, unde X și Y sunt obiecte ale clasei definite sau variabile de alt tip.

Dintr-un motiv similar, operatorul << returnează o referință la ostream & out.

1.6 Supraîncărcarea operatorilor de incrementare/decrementare

Operatorii de incrementare (++) și decrementare (--) sunt unari, deoarece se aplică la un singur operand. Ei sunt totuși speciali, deoarece:

- pot fi prefixați (++ X) sau postfixați (X ++);
- efectul este modificare obiectului curent (de exemplul mărirea cu 1 sau altă modificare, în funcție de specificul clasei pentru care îi definim);
- rezultatul diferă:
    - pentru operatorul prefixat, rezultatul este obiectul curent, după modificare;
    - pentru operatorul postfixat, rezultatul este o copie a obiectului curent, înainte de modificare.

Implementarea se realizează astfel, ca metodă:

- pentru operatorul prefixat se utilizează o metodă fără parametri, care modifică obiectul curent și returnează o referință la acesta;
- pentru operatorul postfixat se utilizează, prin convenție, o metodă cu un parametru, de tip int; metoda modifică obiectul curent și returnează o copie a acestuia, realizată înainte de modificare.

Programul de mai jos supraîncarcă operatorii de preincrementare și postincrementare pentru clasa Fracție. Operațiile de decrementare se pot realiza similar.

```
#include <iostream>
```

```
using namespace std;

class Fractie{
    private:
        int numarator, numitor;
    public:
        Fractie(int a = 0, int b = 1)
        {
            // muta semnul numitorului (-) la numarator
            // in cazul in care numitorul este negativ
            if(b < 0)
            {
                a = -a;
                b = -b;
            }
            numarator = a;
            numitor = b;
        }

        friend ostream & operator << (ostream & out , const Fractie & F)
        {
            out << F.numarator << "/" << F.numitor;
            return out;
        }
        // incrementare prefixata
        Fractie & operator ++()
        {
            numarator += numitor;
            return * this;
        }

        // incrementare postfixata
        Fractie operator ++(int)
        {
            Fractie tmp = Fractie(numarator , numitor);
            numarator += numitor;
            return tmp;
        }
};

int main(){
    Fractie X , Y;
    Y = Fractie(1 , 2);
    X = Y ++;
    cout << X << " " << Y << endl; // 1/2 3/2

    Y = Fractie(1 , 2);
    X = ++ Y;
    cout << X << " " << Y << endl; // 3/2 3/2

    return 0;
}
```

Observație: Operatorii ++ și -- pot fi implemntați și ca funcții prietene. Exemplu:

```
#include <iostream>

using namespace std;

class Fractie{
    private:
        int numarator, numitor;
    public:
        Fractie(int a = 0, int b = 1)
        {
            // muta semnul numitorului (-) la numarator
            // in cazul in care numitorul este negativ
            if(b < 0)
            {
                a = -a;
                b = -b;
            }
            numarator = a;
            numitor = b;
        }

        friend ostream & operator << (ostream & out , const Fractie & F)
        {
            out << F.numarator << "/" << F.numitor;
            return out;
        }
        // incrementare prefixata
        friend Fractie & operator ++(Fractie & F)
        {
            F.numarator += F.numitor;
            return F;
        }

        // incrementare postfixata
        friend Fractie operator ++(Fractie & F, int)
        {
            Fractie tmp = Fractie(F.numarator , F.numitor);
            F.numarator += F.numitor;
            return tmp;
        }
};

int main(){
    Fractie X , Y;
    Y = Fractie(1 , 2);
    X = Y ++;
    cout << X << " " << Y << endl; // 1/2 3/2

    Y = Fractie(1 , 2);
    X = ++ Y;
    cout << X << " " << Y << endl; // 3/2 3/2
```

```
    return 0;
}
```

In cazul implementării prin funcții prietene, parametrul obiect trebuie tranmis prin referință, pentru a se întoarce din funcție modificat.

2 Aplicații

2.1 Complex

Se dă următoare definire a clasei Complex:

```
class Complex
{

private:
    double r, i;

public:

    // constructor implicit
    Complex();
    // constructor cu parametri
    Complex(double, double);
    // constructor de copiere
    Complex(const Complex&);

    // supraincarcare operator <<
    friend ostream& operator<<(ostream&, const Complex&);
    // supraincarcare operator >>
    friend istream& operator>>(istream&, Complex&);

    // supraincarcare operator +, cazul z1 + z2
    Complex operator+(const Complex&) const;
    // supraincarcare operator +, cazul z + 1
    Complex operator+(double) const;
    // supraincarcare operator +, cazul 1 + z
    friend Complex operator+(double, const Complex&);

    // supraincarcare operator =
    Complex& operator=(const Complex&);

    // supraincarcare operator ++, prefixat
    Complex& operator++();
    // supraincarcare operator ++, postfixat
    Complex operator++(double);

    // supraincarcare operator ==
    bool operator==(const Complex&);

};
```

Implementați clasa declarată anterior.

3 Rezolvări

3.1 Complex

```
#include <iostream>

using namespace std;

class Complex
{

private:
    double r, i;

public:

    // constructor implicit
    Complex();
    // constructor cu parametri
    Complex(double, double);
    // constructor de copiere
    Complex(const Complex&);

    // supraincarcare operator <<
    friend ostream& operator<<(ostream&, const Complex&);
    // supraincarcare operator >>
    friend istream& operator>>(istream&, Complex&);

    // supraincarcare operator +, cazul z1 + z2
    Complex operator+(const Complex&) const;
    // supraincarcare operator +, cazul z + 1
    Complex operator+(double) const;
    // supraincarcare operator +, cazul 1 + z
    friend Complex operator+(double, const Complex&);

    // supraincarcare operator =
    Complex& operator=(const Complex&);

    // supraincarcare operator ++, prefixat
    Complex& operator++();
    // supraincarcare operator ++, postfixat
    Complex operator++(int);

    // supraincarcare operator ==
    bool operator==(const Complex&);

};

// constructor implicit
Complex::Complex()
{
    r = 0;
    i = 0;
}
```

```
// constructor cu parametri
Complex::Complex(double re, double im)
{
    r = re;
    i = im;
}

// constructor de copiere
Complex::Complex(const Complex& z)
{
    r = z.r;
    i = z.i;
}

// supraincarcare operator <<
ostream& operator<<(ostream& out, const Complex& z)
{
    out << z.r << " + " << z.i << "i" << "\n";
    return out;
}

// supraincarcare operator >>
istream& operator>>(istream& in, Complex& z)
{
    cout << "Partea reala este: ";
    in >> z.r;
    cout << "Partea imaginara este: ";
    in >> z.i;

    return in;
}

// supraincarcare operator +, cazul z1 + z2
Complex Complex::operator+(const Complex& z) const
{
    double re = r + z.r;
    double im = i + z.i;

    Complex rezultat(re, im);

    return rezultat;
}

// supraincarcare operator +, cazul z + 1
Complex Complex::operator+(double a) const
{
    Complex rezultat(r + a, i);
    return rezultat;
}

// supraincarcare operator +, cazul 1 + z
Complex operator+(double a, const Complex& z)
```

```
{
    return z + a;
}

Complex& Complex::operator=(const Complex& z) {
    if (this != &z) {
        r = z.r;
        i = z.i;
    }
    return *this;
}

Complex& Complex::operator++() {
    r++;
    return *this;
}

Complex Complex::operator++(int) {
    Complex temp(*this);
    r++;
    return temp;
}

bool Complex::operator==(const Complex& z) {
    return (r == z.r && i == z.i);
}

int main()
{
    Complex z1, z2;

    cin >> z1 >> z2;
    cout << z1 << z2;

    Complex z3 = z1 + z2;
    cout << z3;

    cout << z1 + 1.0;
    cout << 2.0 + z1;

    ++z1;
    cout << z1;

    z2++;
    cout << z2;

    if (z1 == z2)
    {
        cout << "Egale!\n";
    }

    return 0;
}
```

4 Teme

4.1 Mulțime

Se dă clasa *Multime*, care simulează o mulțime de numere întregi. Clasa va simula operația de reuniune a două mulțimi, prin supraîncărcarea operatorilor. Se dă următoarea definire a clasei:

```
class Multime
{
private:
    int m;
    int* data;
    void init(int, int*);

public:

    // constructori
    Multime();
    Multime(int, int*);
    Multime(const Multime&);

    // afisare
    friend ostream& operator<<(ostream&, const Multime&);

    // citire
    friend istream& operator>>(istream&, Multime&);

    // atribuire
    Multime& operator=(const Multime&);

    // reuniune
    Multime operator+(const Multime&);

    // intersectie
    Multime operator*(const Multime&);

    // diferenta
    Multime operator-(const Multime&);

    // destructor
    ~Multime();

};
```

Se cere să se implementeze această clasă.

Următoarele probleme sunt opționale.

4.2 Funcție

Se cere implementarea unei clase Functie care să reprezinte o funcție liniară cu coeficienți reali și grad întreg pozitiv, folosind alocare dinamică pentru stocarea coeficienților. Definiția clasei este:

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

class Functie
{
private:
    int g;
    double* coef;

public:
    // constructor implicit
    Functie(int g = 0);

    // constructor cu parametri
    Functie(int g, const double* coef);

    // constructor de copiere
    Functie(const Functie&);

    // destructor
    ~Functie();

    // supraincarcare operator =
    Functie& operator=(const Functie&);

    // supraincarcare operator >>
    friend istream& operator>>(istream&, Functie&);

    // supraincarcare operator <<
    friend ostream& operator<<(ostream&, const Functie&);

    // supraincarcare operator + pentru suma a doua polinoame (P1 + P2)
    Functie operator+(const Functie&) const;

    // supraincarcare operator + pentru P1 + double
    Functie operator+(double) const;

    // supraincarcare operator + pentru double + P1
    friend Functie operator+(double, const Functie&);

    // evaluare functie in punctul x, P(x)
    double operator[](double x) const;

    // derivata de ordin 1 a functiei
    Functie operator!() const;
};


int main()
{
    Functie f1, f2;
    cin >> f1 >> f2;
```

```
    Functie s1 = f1 + f2;
    Functie s2 = f1 + 2.5;
    Functie s3 = 3.14 + f2;

    cout << s1 << p1;

    // F1(1.5)
    double val = f1[1.5];
    cout << "Valoarea functiei f1 in 1.5 este: " << val;

    Functie d = !f1;
    cout << "Derivata lui f1 este: " << d;

}
```

4.3 Timp

Se cere implementarea unei clase *Timp* care modelează un moment al zilei sub forma ore, minute și secunde. Clasa trebuie să gestioneze valori corecte (adică $0 \leq$ secunde < 60, $0 \leq$ minute < 60, etc.). Definiția clasei este:

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

class Timp {
private:
    int h, m, s;

       // pentru transformarea in formatul hh:mm:ss
    void normalizare();

public:
    // constructori
    Timp();
    Timp(int, int, int);

    // supraincarcare operatori >> si  <<
    friend istream& operator>>(istream&, Timp&);
    friend ostream& operator<<(ostream&, const Timp&);

    // supraincarcare operatori + si - pentru adunare si scadere doi timpi
    Timp operator+(const Timp&) const;
    Timp operator-(const Timp&) const;

    // supraincarcare operatori + si -
    // pentru adunarea si scaderea unui numar de secunde
    Timp operator+(int) const;
    friend Timp operator+(int, const Timp&);
    Timp operator-(int) const;

    // supraincarcare operatori ++ si -- (prefixat si postfixat)
    Timp& operator++();     // prefixat
```

```
    Timp operator++(int);   // postfixat
    Timp& operator--();     // prefixat
    Timp operator--(int);   // postfixat

    // supraincarcare operatori de comparare
    bool operator==(const Timp&) const;
    bool operator!=(const Timp&) const;
    bool operator<(const Timp&) const;
    bool operator<=(const Timp&) const;
    bool operator>(const Timp&) const;
    bool operator>=(const Timp&) const;

    // transformare in total secunde
    int operator()() const;

    // acces la componente prin index:
    // [0] = ore, [1] = minute, [2] = secunde
    int operator[](int index) const;
};


int main() {
    Timp t1(1, 59, 30), t2(0, 1, 40);

    cout << "Timp 1: " << t1 << '\n';
    cout << "Timp 2: " << t2 << '\n';

    Timp suma = t1 + t2;
    cout << "Suma: " << suma << '\n';

    Timp t3 = t1 - t2;
    cout << "Diferenta: " << t3 << '\n';

    cout << "Timp 1 + 75 secunde: " << t1 + 75 << '\n';
    cout << "75 + Timp 2: " << 75 + t2 << '\n';

    cout << "Timp 1 total secunde: " << t1() << '\n';
    cout << "Minute din Timp 1: " << t1[1] << '\n';

    ++t1;
    cout << "Dupa ++t1: " << t1 << '\n';

    t2++;
    cout << "Dupa t2++: " << t2 << '\n';

    cout << (t1 > t2 ? "t1 > t2" : "t1 <= t2") << '\n';

    return 0;
}
```

4.4 Adunare numere mari

Se cere implementarea unei clase Numar care să poată stoca și manipula numere întregi foarte mari, ce nu pot fi reprezentate folosind tipurile de date standard din C++ (int, long long, etc.). Numărul va fi stocat intern ca șir de caractere alocat dinamic, iar clasa va supraîncărca operatorul de adunare a două numere (algoritmul clasic de adunare a două numere cifră cu cifră, începând cu ultima cifră).

```
#include <iostream>

using namespace std;

class Numar {
private:
    // sir alocat dinamic pentru cifre (ex: "123456789")
    char* cifre;
    // numar de cifre
    int lungime;

    // pentru inversarea numerelor pt adunare
    void reverse(char* s, int len) const;

public:
    // constructori
    Numar();
    Numar(const char*);
    Numar(const Numar&);

    // destructor
    ~Numar();

    // operator de atribuire
    Numar& operator=(const Numar&);

    // operator de adunare
    Numar operator+(const Numar&) const;

    // operator << pentru afisare
    friend ostream& operator<<(ostream&, const Numar&);

    // operator >> pentru citire
    friend istream& operator>>(istream&, Numar&);
};

int main() {
    Numar a, b;
    cout << "Introdu primul numar: ";
    cin >> a;
    cout << "Introdu al doilea numar: ";
    cin >> b;

    Numar suma = a + b;
    cout << "Suma: " << suma << endl;
```

```
    return 0;
}
```