Capitolul 4. Supraîncărcarea operatorilor

Motivație

```
#include <iostream>

int main(){
    int a;
    int b;

    std::cin >> a >> b;

    std::cout << (a + b) / 2.0 << std::endl;

    return 0;
}
```

Scrieți un program care calculează media aritmetică a două numere **întregi**, `a` și `b`, citite de la tastatură.

Motivație

```
#include <iostream>

class Complex{
private:
    float re;
    float im;
public:
    Complex(float re=0, float im=0):re(re), im(im){}
};

int main(){
    float re;
    float im;

    std::cin >> re >> im;
    Complex c1(re, im);
    std::cin >> re >> im;
    Complex c2(re, im);

    std::cout << (c1 + c2) / 2;

    return 0;
}
```

```
error: no match for 'operator+' (operand types are 'Complex' and 'Complex')
=== Build failed: 1 error(s), 0 warning(s) (0 minute(s), 0 second(s)) ===
```

Scrieți un program care calculează media aritmetică a două numere **complexe**, `a` și `b`, citite de la tastatură.

$$a = re_1 + i * im_1$$
$$b = re_2 + i * im_2$$

$$a + b = (re_1 + re_2) + i * (im_1 + im_2)$$
$$a - b = (re_1 - re_2) + i * (im_1 - im_2)$$
$$a * \beta = re_1 * \beta + i * im_1 * \beta$$
$$a \,/\, \beta = re_1 \,/\, \beta + i * im_1 \,/\, \beta$$
$$a * b = [re_1 * re_2 - im_1 * im_2] + i * (re_1 * im_2 + re_2 * im_1)$$
$$a \,/\, b = \frac{re_1 + i * im_1}{re_2 + i * im_2} = \frac{(re_1 + i * im_1) * (re_2 - i * im_2)}{(re_2 + i * im_2) * (re_2 - i * im_2)} = \frac{(re_1 * re_2 + im_1 * im_2) + i * (re_2 * im_1 - re_1 * im_2)}{re_2^2 + im_2^2} = \frac{re_1 * re_2 + im_1 * im_2}{re_2^2 + im_2^2} + i * \frac{re_2 * im_1 - re_1 * im_2}{re_2^2 + im_2^2}$$

Motivație

```
#include <iostream>
class Complex{
private:
    float re;
    float im;
public:
    Complex(float re=0, float im=0):re(re), im(im){}

    Complex adunare(const Complex &c){
        return Complex(this->re+c.re, this->im+c.im);
    }

    Complex impartire_la_scalar(float scalar){
        if (scalar != 0){
            return Complex(this->re/scalar, this->im/scalar);
        }
        else {
            std::cout << "Impartire la 0";
            exit(1);
        }
    }

    void afisare(){
        std::cout << this->re << " + i*"<< this->im;
    }
};
```

```
int main(){
    float re;
    float im;

    std::cin >> re >> im;
    Complex c1(re, im);

    std::cin >> re >> im;
    Complex c2(re, im);

    c1.adunare(c2).impartire_la_scalar(2).afisare();

    return 0;
}
```

Motivație

```
#include <iostream>

class Complex{...};

int main(){
    float re;
    float im;

    std::cin >> re >> im;
    Complex c1(re, im);

    std::cin >> re >> im;
    Complex c2(re, im);

    // (c1 * c2) / (c1 - c2 + 14)
    c1.inmultire(c2).impartire(c1.scadere(c2).adunare_cu_scalar(14)).afisare();

    return 0;
}
```

Scrierea devine complicată și neintuitivă

Supraîncărcarea operatorilor

Supraîncărcarea operatorilor = mecanism pentru personalizarea funcționalității operatorilor pentru tipurile de date definite de utilizator (clase/structuri/templates).

Operatorii supraîncărcați sunt, în esență, funcții cu nume speciale, supraîncărcate. Ei pot aparține clasei pentru care sunt definiți sau pot fi non-membri.

Supraîncărcarea operatorilor

```
#include <iostream>
#include <string>

int main(){
    std::string s1="Acesta este un";
    std::string s2;
    std::string s3;
    int nr_crt;

    std::cin >> nr_crt; // operatorul >> pentru intregi
    std::cin >> s2; // operatorul >> pentru std::string
    s3 = s1 + s2; // operatorii = si + pentru std:: string

    std::cout << "Testul #" << nr_crt << " pentru operatori:"; // operatorul << pentru intregi
    std::cout << std::endl;
    std::cout << s3; // operatorul << pentru std::string

    return 0;
}
```

Supraîncărcarea operatorilor

Sintaxă:

```
operator op // operatori unari si binari
operator type // operatori de conversie de tip
operator new // operator de alocare (dinamica)
operator new [] // operator de alocare (dinamica)
operator delete // operator de dealocare (dinamica)
operator delete [] // operator de dealocare (dinamica)
```

Supraîncărcarea operatorilor

```
#include <iostream>
class Complex{
private:
    float re;
    float im;
public:
    Complex(float re=0, float im=0):re(re), im(im){}

    Complex operator+(const Complex &c){
        return Complex(this->re+c.re, this->im+c.im);
    }

    Complex operator/(float scalar){
        if (scalar != 0){
            return Complex(this->re/scalar, this->im/scalar);
        }
        else {
            std::cout << "Impartire la 0";
            exit(1);
        }
    }

    void afisare(){
        std::cout << this->re << " + i*"<< this->im;
    }
};
```

```
int main(){
    float re;
    float im;

    std::cin >> re >> im;
    Complex c1(re, im);

    std::cin >> re >> im;
    Complex c2(re, im);

    ((c1 + c2) / 2).afisare();

    return 0;
}
```

Supraîncărcarea operatorilor

```
class Complex{
...
    Complex operator+(const Complex &c){
        return Complex(this->re+c.re, this->im+c.im);
    }
...
};


c1 + c2 <=> c1.operator+(c2);
```

<u>Funcție membră a clasei Complex</u>
Obiectul din care se face apelul funcției este considerat a fi unul dintre operanzi și este transmis implicit prin pointerul `this`.

Majoritatea operatorilor nu modifică operanzii

```
Complex operator+(const Complex &lhs, const Complex &rhs){
    return Complex(lhs.re+rhs.re, lhs.im+rhs.im);
}


c1 + c2 <=> operator+(c1, c2);
```

<u>Funcție non-membră a clasei Complex</u>
Ambii operanzi trebuie să fie transmiși funcției.

Accesul la membrii `private` se poate face în 2 moduri:
1. prin funcții publice (getters)
2. prin utilizarea mecanismului `friend`

Supraîncărcarea operatorilor

```
class Complex{
private:
    float re;
    float im;
public:
    ...
    float get_re() const {return this->re;}
    float get_im() const {return this->im;}
    ...
};


Complex operator+(const Complex &lhs, const Complex &rhs){
    return Complex(lhs.get_re() + rhs.get_re(),
                   lhs.get_im() + rhs.get_im());
}
```

```
class Complex{
private:
    float re;
    float im;
public:
    ...
    friend Complex operator+(const Complex &lhs,
                             const Complex &rhs);
    ...
};


Complex operator+(const Complex &lhs,
                  const Complex &rhs){
    return Complex(lhs.re + rhs.re,
                   lhs.im + rhs.im);
}
```

Supraîncărcarea operatorilor

Implementarea ca membru al clasei vs non-membru (reguli orientative):

1. Operatorii unari se implementează ca funcții membre;
2. Operatorii binari care nu modifică operanzii se implementează ca funcții ne-membre;
3. Operatorii binari care modifică unul dintre operanzi (de obicei pe cel stâng) se implementează ca funcție membră.

Supraîncărcarea operatorilor

La întâlnirea unui operator pentru care cel puțin unul dintre operanzi este un tip de date definit de utilizator, apelul funcției operator (supraîncărcate) se face căutându-se în următoarea ordine:

1. candidați membri (interiorul clasei/structurii/template-ului);
2. candidați non-membri
3. candidați predefiniți de limbajul C++

Supraîncărcarea operatorilor

Ce operatori pot fi supraîncărcați?

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

Ce operatori nu pot fi supraîncărcați?

.	.*	::	?:

Supraîncărcarea operatorilor

Implementări canonice: operatorii supraîncărcați ar trebui să aibă un comportament cât mai asemănător posibil cu implementarea implicită, dată de limbaj. Operatori supraîncărcați în mod uzual:

1. operatorul de asignare (cu cazurile particulare: copiere, mutare): =;
2. operatorii de extragere/inserție din/în fluxuri: <<, >>
3. operatorii de incrementare/decrementare: ++, --
4. operatorii aritmetici binari: +, -, *, /
5. operatorii relaționali: <, >, <=, >=, ==, !=

Supraîncărcarea operatorilor

Indicații relativ la aplicarea supraîncărcării operatorilor:

1. dacă există dubii asupra funcționalității unui operator atunci nu se va supraîncărca;
2. se folosesc doar implementări intuitive;
3. dacă se supraîncarcă un operator, atunci se vor supraîncărca toți operatorii înrudiți.

Operatorul de copiere prin asignare

```
class A{
private:
    int *p_array;
    int dim;
public:
    ... // destructor, constructori default si de copiere
    A& operator=(const A& a){
        if (this == &a) {
            return *this;
        }

        if (dim != a.dim){
            delete [] this->p_array;
            this->p_array = nullptr;
            this->dim = 0;
            this->p_array = new int[a.dim];
            this->dim = a.dim;
        }

        for (int i=0; i<dim; i++){
            *(this->p_array+i) = *(a.p_array+i);
        }

        return *this;
    }
}
```

```
int main(){
        int *p = new int[6];
    for (int i=0; i<6; i++){
        *(p+i) = i;
    }
    A a(p, 6);
    A b;
    b = a;
        return 0;
}
```

Operatorul de copiere prin asignare

Regula celor 3: dacă se implementează în mod explicit oricare dintre:

1) destructor,
2) constructor de copiere,
3) operator de copiere,

atunci probabil trebuie implementate explicit toate 3.

Operatorul de mutare prin asignare

```
#include <iostream>
#include <utility>

class A{
private:
    int dim;
    int *p_array;
public:
    ... // destructor, constructori de copiere si de
mutare, operator de copiere
    A& operator=(A&& a) noexcept {
        if (this == &a) {
            return *this;
        }
        delete[] this->p_array;
        this->p_array = std::exchange(a.p_array, nullptr);
        this->dim = std::exchange(a.dim, 0);
        return *this;
    }
};

A f(A a){return a;}
```

```
int main(){
    int *p = new int[6];
    for (int i=0; i<6; i++){
        *(p+i) = i;
    }
    A a(p, 6);
    A b;
    b = f(a);
    b.display();
    return 0;
}
```

Operatorul de mutare prin asignare

Începând cu C++11, “Regula celor 3” devine:

Regula celor 5: dacă se implementează în mod explicit oricare dintre:

1) destructor,
2) constructor de copiere,
3) operator de copiere,
4) constructor de mutare,
5) operator de mutare,

atunci probabil trebuie implementate explicit toate 5.

Supraîncărcarea operatorilor << și >>

```
#include <iostream>

class Complex{
private:
    float re, im;
public:
    Complex(float re=0, float im=0):re(re), im(im){}
    friend std::ostream& operator<<(std::ostream& os, const Complex& c);
    friend std::istream& operator>>(std::istream& is, Complex& c);
};

std::ostream& operator<<(std::ostream& os, const Complex& c){
    os << c.re << " + i*" << c.im;
    return os;
}

std::istream& operator>>(std::istream& is, Complex& c){
    is >> c.re >> c.im;
    return is;
}
```

```
int main(){
    Complex c;

    std::cin >> c;
    std::cout << c;

    return 0;
}
```

Supraîncărcarea operatorilor ++

```
#include <iostream>

class Complex{
private:
    float re, im;
public:
    Complex(float re=0, float im=0):re(re), im(im){}
    Complex& operator++()
    {
        this->re++;
        this->im++;
        return *this; // intoarce valoarea noua prin referinta
    }

    Complex operator++(int)
    {
        Complex temp = *this;
        this->re++;
        this->im++;
        return temp; // intoarce valoarea veche
    }
    ... // supraincarcarea operatorilor << si >>
};
```

```
int main(){
    Complex c;

    std::cin >> c;
    std::cout << "c:\t" << c << std::endl;
    std::cout << "c++:\t" << c++ << std::endl;
    std::cout << "c:\t" << c << std::endl;
    std::cout << "++c:\t" << ++c << std::endl;

    return 0;
}
```

Supraîncărcarea operatorilor --

```
#include <iostream>

class Complex{
private:
    float re, im;
public:
    Complex(float re=0, float im=0):re(re), im(im){}
    Complex& operator--()
    {
        this->re--;
        this->im--;
        return *this; // intoarce valoarea noua prin referinta
    }

    Complex operator--(int)
    {
        Complex temp = *this;
        this->re--;
        this->im--;
        return temp; // intoarce valoarea veche
    }
    ... // supraincarcarea operatorilor << si >>
};
```

```
int main(){
    Complex c;

    std::cin >> c;
    std::cout << "c:\t" << c << std::endl;
    std::cout << "c--:\t" << c-- << std::endl;
    std::cout << "c:\t" << c << std::endl;
    std::cout << "--c:\t" << --c << std::endl;

    return 0;
}
```

Supraîncărcarea operatorilor aritmetici binari

Să se implementeze operatorii aritmetici binari pentru clasa Complex – formulele se găsesc pe slide-ul 3.

Supraîncărcarea operatorilor relaționali

```
#include <iostream>

class Complex{
private:
    float re, im;
public:
    Complex(float re=0, float im=0):re(re), im(im){}
    friend bool operator<(const Complex& lhs, const Complex& rhs);
    friend bool operator>(const Complex& lhs, const Complex& rhs);
    friend bool operator<=(const Complex& lhs, const Complex& rhs);
    friend bool operator>=(const Complex& lhs, const Complex& rhs);
    ... // supraincarcarea operatorilor >>, ++
};

bool operator<(const Complex& lhs, const Complex& rhs){
    return ((lhs.re<rhs.re) && (lhs.im<rhs.im));
}
bool operator>(const Complex& lhs, const Complex& rhs){
    return rhs<lhs;
}
bool operator<=(const Complex& lhs, const Complex& rhs){
    return !(rhs<lhs);
}
bool operator>=(const Complex& lhs, const Complex& rhs){
    return !(lhs<rhs);
}
```

```
int main(){
    Complex c;
    std::cin >> c;
    Complex a = c;
    c++;

    std::cout << (a>c) << std::endl;
    std::cout << (a<c) << std::endl;
    std::cout << (a<=c) << std::endl;
    std::cout << (a>=c) << std::endl;

    return 0;
}
```

Sfârșit capitol 4