

Programare II

Limbajul C/C++

CURS 9



Curs anterior

- ❑ Clase. Obiecte

 - ❑ Constructori de copiere / mutare

 - ❑ Constructori și excepții

- ❑ Membri statici ai claselor

- ❑ Membri prieteni ai claselor

- ❑ Modificatori de acces

- ❑ Relații între clase

Cuprins

❑ Supraîncărcarea operatorilor

- ❑ Funcții membre

- ❑ Funcții prietene

❑ Operatori

- ❑ Asignare

- ❑ Binari

- ❑ Prescurtați

- ❑ Unari

- ❑ Conversii de tip

Ce este supraîncărcarea operatorilor?

☐ Supraîncărcarea operatorilor

- ☐ Permite definirea comportamentului operatorilor când sunt **aplicați** obiectelor unui **nou tip de date** (clase)

☐ Ce operatori trebui supraîncărcați?

- ☐ Cei care **au sens** pentru noul tip de date definit

☐ Restricții?

- ☐ Nu se poate **schimba „înțelesul”** operatorilor când sunt aplicați asupra **tipurilor de bază**
- ☐ Nu se pot defini **noi simboluri** pentru operatori
- ☐ Unul dintre **operandi** trebuie să fie **de tipul clasei** pentru care s-a supraîncărcat operatorul

Supraîncărcarea operatorilor

- ❑ Similară cu supraîncărcarea funcțiilor

- ❑ **Numele funcției** este înlocuit de cuvântul cheie `operator` urmat de simbolul operatorului

- ❑ **Tipul de return** reprezintă tipul valori care va fi **rezultatul operației**

- ❑ **Argumentele** sunt 1 sau 2 operanzii în funcție de **n-aritatea operatorului**

Supraîncărcarea operatorilor

❑ Exemplu

❑ Folosirea operatorului + pentru clasa numere complexe

```
class NrComplex{
public:
    friend NrComplex & operator+ (const NrComplex &, const NrComplex
&);
    ...
private:
    double _real; double _imag;
};

int main(){
    NrComplex c1, c2(4,5);
    NrComplex c3 = c1+c2;
    cout << c3;
}
```

Prototip funcție de supraîncărcare operator +

Folosirea operatorului + cu noul tip de date
NrComplex

Observații

- ❑ Nu se poate modifica

 - ❑ n-aritatea

 - ❑ Asociativitatea

 - ❑ Prioritatea

- ❑ Nu se pot utiliza **valori implicite** (default)

- ❑ Recomandări

 - ❑ Nu este bine să se **modifice sensul** operatorilor

 - ❑ Operatorul + sa nu însemne scădere

 - ❑ **Definiți consistente**

 - ❑ Dacă operatorul + este definit, atunci și operatorul += ar trebui definit

Sintaxă

❑ Definire funcție cu numele `operator` urmat de simbolul operatorului

❑ `tip_returnat operator simbol (lista_de_parametri);`

❑ simbol orice operator C++ cu excepția

❑ `.`

❑ `.*`

❑ `::`

❑ `?:`

❑ `sizeof`

❑ Apel operator: `NrComplex a(3, 4), b(5, 6), c;`

❑ Implicit

`c = a + b`

❑ Explicit

`c = operator+(a, b);`

Care operatori pot fi supraîncărcați?

❑ Aproape toți operatori

❑ Aritmetici

❑ `+, -, *, /, %, ++, --`

❑ Operatori pe biți

❑ `^, &, >>, <<`

❑ Operatori logici

❑ `&&, ||, !`

❑ Operatori relaționali

❑ `>, <, <=, >=, ==, !=`

❑ Operatorul de indexare, funcție, virgulă

❑ `[] (), ,`

❑ Operatorul de atribuire și cei compuși

❑ `=, ^=, |=, &=, +=, -=, *=, /=, %=, <<=, >>=`

❑ Supraîncărcați global de către compilator

❑ `new, delete, =, &, ->*, ->`

Moduri de supraîncărcare

❑ Cum se face corespondența dintre operatorii și metodele clasei?

❑ Ex. utilizare `a+b`; `a+=b`;

❑ Soluții

❑ Ca **funcții prietene** ale clasei

❑ `friend NrComplex& operator+= (NrComplex &a, const NrComplex &b);`

❑ `friend NrComplex operator+ (const NrComplex &a, const NrComplex &b);`

Pentru operatorul `+=` nu este preferată această formă de supraîncărcare deoarece operatorul modifică valoarea primului operand

❑ Ca **funcții membre** ale clasei

❑ `NrComplex& operator+= (const NrComplex &b);`

❑ `NrComplex operator+ (const NrComplex &b) const;`

Pentru operatorul `+` nu este preferată această formă de supraîncărcare deoarece întotdeauna primul operand trebuie să fie de tip `NrComplex` și funcția trebuie să fie constantă

Moduri de supraîncărcare

Ca funcții prietene ale clasei

❑ Prototip

```
friend NrComplex operator + (  
    const NrComplex& a, const NrComplex& b);
```

❑ Definire

```
NrComplex operator + (const NrComplex& a,  
const NrComplex& b)  
{  
    return NrComplex(a._real  
        + b._real,  
        a._imag+b._imag);  
}
```

❑ Apel

```
NrComplex x,y;  
NrComplex z = x+y;
```

Ca funcții membre ale clasei

❑ Prototip

```
NrComplex operator + (const NrComplex& b)  
const;
```

❑ Definire

```
complex NrComplex::operator + (  
    const NrComplex& altObj) const  
{  
    return NrComplex(_real+  
        altObj._real,  
        _imag+ altObj._imag);  
}
```

❑ Apel

```
❑ NrComplex x,y;  
complex z = x+y;
```

Moduri de supraîncărcare

❑ Operatori care trebuie supraîncărcați ca funcții membre sunt

❑ = operatorul de atribuire

❑ [] operatorul de indexare

❑ () operatorul de apel al unei funcții

❑ -> operatorul de accesare indirectă a unui membru

❑ ->* operatorul de accesare indirectă a unui membru pointer

Operatori supraîncărcați în mod uzual

□ Uni dintre operatori supraîncărcați în mod uzual sunt

□ \ll, \gg

□ $=$

□ $<$

Supraîncărcarea operatorului =

- ❑ În lipsa supraîncărcării **compilatorul generează** o copie membru cu membru a datelor clasei
- ❑ Comportament similar cu constructorul de copie
- ❑ Operatorul de atribuire trebuie să fie **supraîncărcat ca funcție membră**, nu modifică al **doilea operand** (trebuie să fie o **referință constantă**)
- ❑ Operatorul de asignare poate fi **înlănțuit**, deci trebuie să **întoarcă o referință**

Supraîncărcarea operatorului =

```
class Student{
    char * nume;

public:
    Student & operator = (const Student &);

};

Student & Student::operator = (const Student & s2) {
    if (this == &s2)
        return *this; //verificare autoreferință
    if (nume){
        delete []nume; //ștergere valoare existentă
        nume = new char[strlen(s2.nume)+1];
        strcpy(nume, s2.nume);
    }
    return *this;
}
```

Supraîncărcarea operatorului =

❑ Observație

❑ Dacă o clasă conține variabile membru de tip pointer următoarele funcții trebuie implementate

❑ Constructor

❑ Constructor de copiere

❑ Destructor

❑ Supraîncărcarea operatorului =

Supraîncărcarea operatorilor << și >>

- ❑ Operatorii << și >> se supraîncarcă pentru a putea insera în fluxurile de ieșire și extrage din fluxurile de intrare
- ❑ Biblioteca *iostream* suprascrie operațiile de scriere/citire pentru tipurile implicite
- ❑ Pentru noile tipuri de date definite de utilizatori aceștia sunt responsabili cu suprascrierea lor
- ❑ Nu pot fi supraîncărcați ca funcții membre, trebuie utilizate funcții non-membre, deoarece primul operand este un obiect de tipul ostream/istream și nu o referință la clasă

Supraîncărcarea operatorilor << și >>

❑ Prototip

```
friend ostream &operator<<(ostream &, const NrComplex &);  
friend istream &operator>>(istream &, NrComplex &);
```

Valoare de return are același tip cu primul operand

Primul operand poate fi streamul cin sau cout sau fișier

Citirea modifică al doilea operand

❑ Utilizare

```
NrComplex c1(7,9);  
cin >> c1;  
cout << c1 << endl;
```

Supraîncărcarea operatorilor << și >>

❑ Recomandare

❑ Nu introduceți mesaje și formatare când supraîncărcați operatorii << și >>

```
ostream &operator<<(ostream &out, const complex &c)
{
    out << " Complex[" << c._real << "+" << c._imag << "i]";
    return out;
}
```

```
istream &operator<<(istream &in, complex &c)
{
    cout << "Introduceti partea reala"; in >> c._real;
    cout << "Introduceti partea imaginara"; in >> c._imag;
    return in;
}
```

Supraîncărcarea operatorilor aritmetici

❑ Operatori aritmetici

- ❑ $+$, $-$, $*$, $/$, $\%$

- ❑ Supraîncărcare folosind funcții friend

❑ Operatori prescurtați

- ❑ $+=$, $-=$, $*=$, $/=$, $\%=$

- ❑ Supraîncărcare folosind funcții membre

❑ Pentru consistență dacă supraîncărcăm operatorul binar (de exemplu $+$) este bine să supraîncărcăm și operatorul prescurtat ($+=$)

Supraîncărcarea operatorilor aritmetici

❑ Exemplu

```
class NrComplex{
public:
    friend NrComplex operator+ (double &, const NrComplex &);
    friend NrComplex operator+ (const NrComplex &, double &);
    friend NrComplex operator+ (const NrComplex &, const NrComplex &);
    NrComplex & operator += (double &);
    NrComplex & operator += (const NrComplex &);
    ...
};

NrComplex operator+ (const NrComplex &c1, const NrComplex &c2) {
    return NrComplex(c1._real+c2._real, c1._imag+c2._imag);
}

NrComplex& NrComplex ::operator+= (const NrComplex &c) {
    _real += c._real;
    _imag += c._imag;
    return *this;
}
```

NU folosiți implementarea operatorului + pentru implementarea operatorului +=
NrComplex & operator+= (const NrComplex &c) {
 return *this = *this + c;
}

Supraîncărcarea operatorilor relaționali

❑ Recomandare

- ❑ Supraîncărcați ca **funcții non-membre**, deoarece primul operand ar putea fi o valoare de tip diferit față de clasă
- ❑ Valorile **operanzilor** nu se modifică, deci ar trebui să fie **referințe constante**
- ❑ Valoarea de **return** ar trebui să fie de tip **bool**
- ❑ Dacă sunt specificații ca funcții membre, ele trebuie să fie constante

Supraîncărcarea operatorilor relaționali

```
class NrComplex{
public:
    friend bool operator< (double &, const NrComplex &);
    friend bool operator< (const NrComplex &, double &);
    friend bool operator< (const NrComplex &, const NrComplex &);
    friend bool operator== (double &, const NrComplex &);
    friend bool operator== (const NrComplex &, double &);
    friend bool operator== (const NrComplex &, const NrComplex &);
    ...
};

bool operator< (const NrComplex &c1, const NrComplex &c2) {
    return c1.modul()<c2.modul();
}
```

Supraîncărcarea operatorilor unari

- ❑ Operatorii unari

- ❑ ++, --, !, -, +

- ❑ Funcții membre non-stactice, fără argumente

- ❑ Funcții non-membre, cu un argument

- ❑ Argumentul trebuie să fie o referință la obiectul clasei

Supraîncărcarea operatorilor unari

```
class Punct {
private:
    double m_x, m_y;

public:
    Punct (double x=0.0, double y=0.0):
        m_x(x), m_y(y) {
    }

    // Convertește un punct în echivalen tul
    // lui avânt coordonatele negativ
    Punct operator- () const;

    // Întoarce adevărat dacă punctul est e
    // setat în origine
    bool operator! () const;
};
```

```
Punct Punct::operator- () const {
    return Punct(-m_x, -m_y);
}

bool Punct::operator! () const {
    return (m_x == 0.0 && m_y == 0.0);
}

int main(){
    Punct p;
    if (!p)
        cout << "Punctul nu este in origine";
    else
        cout << "Punctul este in origine";
    cout << -p;
}
```

Supraîncărcarea operatorilor ++ și --

- ❑ Operatorii ++ și -- pot fi utilizați în două moduri prefixați și postfixați

- ❑ Recomandare

 - ❑ Trebuie definiți ca și funcții membre

- ❑ Prefixat

```
Contor& Contor ::operator ++ () { .... //corp}  
Contor& Contor ::operator -- () { .... //corp}
```

- ❑ Postfixat

```
Contor Contor ::operator ++ (int a) { .... //corp}  
Contor Contor ::operator -- (int a) { .... //corp}
```

Întoarce o referință la obiectul curent
deoarece la evaluarea se apelează înaintea
evaluării

Parametrul a este fictiv nu are nici un rol. Operatorul întoarce
o copie a obiectului care intră în evaluarea expresiei

Supraîncărcarea operatorilor ++ și --

❑ Exemplu

```
class Contor {  
public:  
    Contor & operator++() {  
        // codul de incrementare  
        return *this;  
    }  
  
    Contor operator++(int) {  
        Contor tmp(*this); // copie  
        operator++(); // pre-increment  
        return tmp; // returnarea vechi valori  
    }  
};
```

Supraîncărcarea operatorului de indexare []

❑ Recomandări

- ❑ Ar trebui supraîncărcată ca **funcție membră**, primul operand ar trebui să fi un membru al clasei
- ❑ Pentru consistență **al doilea operator** ar trebui să fie de **tip întreg** (transmis prin valoare)
- ❑ Deoarece starea obiectului nu se modifică **funcția** ar trebui să fie **constantă**
- ❑ **Valoarea de return** ar trebui să fie de **tipul tabloului** de elemente conținut de clasă

Supraîncărcarea operatorului de indexare []

❑ Exemplu

```
class string{
    char*str;
    int len;
public:
    char & operator [] (int) const;
    ...
};

char & string::operator [] (int index) const {
    return str[index];
}
```

Conversii de tip

❑ Reguli

- ❑ La **evaluarea unei expresii** se efectuează conversii sistematice de la tipurile întregi scurte la tipul `int` sau `unsigned int`. Apoi, pentru fiecare operand, dacă operanzii au tipuri diferite, se efectuează conversia necesară pentru a obține ambii operanzi de același tip.
- ❑ La **atribuire** se efectuează conversia valorii atribuite la tipul operandului care primește valoarea
- ❑ La **transferul parametrilor** se efectuează conversia valorii fiecărui parametru efectiv la tipul parametrului formal

Conversii de tip

☐ Conversii posibile

- ☐ De la tip de bază la tip clasă

- ☐ De la tip clasă la tip de bază

- ☐ De la tip clasă la tip clasă

☐ Cum se pot realiza?

- ☐ Supraîncărcarea **operatorului de cast**

- ☐ Prin intermediul **constructorilor**

Prin intermediul constructorului

```
class NrComplex {  
    int re, im;  
  
public:  
    NrComplex(float f) {  
        re = (int) f;  
    }  
  
    NrComplex(const char *) ;  
};
```

```
void f() {  
    Complex c = 4.8f;  
    Complex c1 = "2+6i";  
}
```


```
void fct(Complex c) {  
    ...  
}
```

```
int main() {  
    f();  
    fct("2+6i");  
}
```

Apel constructor NrComplex(float)



Apel constructor NrComplex(const char)*



Operatorul de cast

- ❑ Conversie de la tip de clasă la tip de bază/altă clasă

- ❑ Sintaxă

```
operator tipDeData ();
```

- ❑ Observații

- ❑ Operatorul este **întotdeauna funcție membră** a clasei

- ❑ **Tipul de return** este **implicit** tipul operatorului

- ❑ **Nu are parametrii** deoarece primește implicit adresa obiectului curent

Operatorul de cast

```
class NrComplex {  
    int re, im;  
  
public:  
    operator double() {  
        return re;  
    }  
};
```

```
void f() {  
    NrComplex c(5, 6);  
    NrComplex e(9, 3);  
    double d = c;  
    double dd = c + e;  
}
```

Apel operator operator double()



Operatorul de cast

- ❑ Supraîncărcarea operatorului de cast **nu** poate realiza **conversii dintr-un tip fundamental în tip clasă**
- ❑ Operatorul de cast este unar
- ❑ Definirea constructorilor **nu** permite **conversia unui tip clasă la un tip de bază**

Polimorfism. Supraîncărcarea operatorilor

- ❑ Polimorfismul este abilitatea de a folosi o metodă în moduri diferite
 - ❑ Ad-hoc polimorfism
 - ❑ Operatorii prin supraîncărcare au comportament diferit în funcție de tipul argumentelor
- ❑ Supraîncărcarea operatorilor face o expresie mai ușor de citit (exprimat) – **syntactic sugar**

ÎNTREBĂRI

