C4: FRIEND si supradefinirea operatorilor

Cuprins:

- Functii si clase “friend” (prietene)
- Supradefinirea operatorilor

C4: FRIEND si supradefinirea operatorilor

```
class complex {

private:
  double re, im;

public:
    complex(double x = 0, double y = 0) {
                set(x, y);
    }
    void set(double x, double y){
                    re=x;
                    im=y;
    }
    double getRe() const{
                return re;
    }
     void afisare() const{
              cout << re << " "<<im<<endl;
     }
 friend void sort_dupa_re(int , complex *);
//Ar avea sens sortarea sa fie functie membra? Cum s-ar apela?
};
```

```
void sort_dupa_re(int n, complex *vec){
for (int i=0;i<n-1;i++)
    for (int j=i+1;j<n;j++)
        if (vec[i].re>vec[j].re){
          complex aux(vec[i]);
          vec[i] = vec[j] ;
          vec[j] = aux;
          }
       }
//nu e functie membra a clasei, dar e declarata
//functie prietena =>  am acces la toate atributele
//si metodele clasei

int main() {
  complex *vec=new complex[3];
  for (int i=0;i<3;i++) vec[i].set(3-i,3-i);

  for (int i=0;i<3;i++) vec[i].afisare();
  cout<<endl;

  sort_dupa_re(3,vec);

  for (int i=0;i<3;i++) vec[i].afisare();

  delete [] vec;
  return 0;
}
```

```
void sort_dupa_re(int n, complex *vec){
for (int i=0;i<n-1;i++)
  for (int j=i+1;j<n;j++)
    if (vec[i].getRe()>vec[j].getRe()){
          complex aux(vec[i]);
          vec[i] = vec[j]  ;
          vec[j] = aux ;
     }
 }
//functie ne-membra a clasei complex dar
//care manipuleaza obiecte de acest tip
```

Pentru sortare am o functie, care nu e membra a clasei, ceea ce inseamna ca nu am acces direct la atributele private si ,deci, trebuie sa implementez in clasa respectiva metodele necesare (ex: getRe()) si sa le folosesc daca vreau sa am acces la atribute.
Exista o varianta mai simpla/rapida?

DA

Functii prietene (friend) ale clasei – au acces la toate datele si metodele clasei, dar nu sunt membre ale clasei.

C4: FRIEND si supradefinirea operatorilor

In ce alte cazuri mai am nevoie de asa ceva?

Ex: Compar modulele a doua numere complexe :

```
//#include <math.h>
 friend   bool compar(const complex   &x, const complex   &y)
{
    return ( sqrt(pow(x.re,2)+pow(x.im,2)) > sqrt(pow(y.re,2)+pow(y.im,2)));
 }
//Cum se apeleaza aceasta functie?
complex c1(1,2), c2(5,4);
compar(c1,c2);
```

Cum fac implementarea ca functie membra? Are sens?

```
bool compar( const complex   &y) const
{
    return ( sqrt(pow(re,2)+pow(im,2)) > sqrt(pow(y.re,2)+pow(y.im,2)));
 }
//Cum se apeleaza aceasta functie?
complex c1(1,2), c2(5,4);
c1. compar(c2);
```

```
//As putea sa implementez functia ca externa si independenta de clasa?
//Da, dar folosesc metode getRe si getIm ca sa am acces la atributele re si im
```

C4: FRIEND si supradefinirea operatorilor

Functii "friend"

- trebuie declarate explicit in interiorul clasei folosind cuvantul friend

- nu sunt membre ale clasei, dar au acces la membrii acesteia: atribute si metode (inclusiv private)

- nu sunt apelatate folosind un obiect de tipul clasei cu care sunt prietene

- permit abateri de la mecanismul de protectie a datelor prin incapsulare => => **C++ nu este un limbaj pur orientat pe obiecte** (mai degraba pragmatic – in caz ca e nevoie de un astfel de instrument – se poate folosi)

Exista functii (ex. compar) ce pot fi implementate ca :

- *functie friend*
- *functie membra*
- *functie exterioara clasei*

In general este preferata implementarea ca functie membra pentru ca nu strica incapsularea datelor (atat timp cat apelul e unul firesc).

Q: Functia pentru sortare se putea implementa ca functie membra? Cum s-ar fi apelat?

Situatii posibile:

(a) Daca o **functie independenta** e prietena a uneia sau mai multor clase ;

$\Rightarrow$ trebuie declarata friend in toate clasele cu care e prietena

(b) Daca o **functie membra a unei clase** M e prietena a unei clase B ;

$\Rightarrow$ trebuie declarata friend in clasa B

(c) Daca **toate functiile membre ale unei clase** M sunt prietene ale unei clase B;

in acest caz se spune ca M (clasa) este prietena a clasei B.

$\Rightarrow$ clasa M trebuie declarata friend in clasa B

Observatii:

- functiile membre au ca argument implicit this, functiile prietene **– *NU***

- prin urmare apelul in cazul functiilor friend nu se face cu numeObj(.) sau (->) si numele functiei, ci doar cu numele functiei.

- ***NU*** *este un mecanism care se mosteneste*

- ***NU*** *este un mecanism tranzitiv:*

Daca:

clasa B1 e prietena clasei B2

clasa B2 e prietena clasei B3

Nu rezulta *ca B1 e prietena lui B3*

- *NU* *e reciproc:*

Daca:

clasa B1 e prietena clasei B2

Nu rezulta *ca B2 este prietena a lui B1*

Cand le folosim ?

– Doar cand este nevoie, protejand cat mai mult incapsularea datelor.

(a) O functie independenta prietena a unei clase

```
class point
{
        int x, y;                                                   // coordonate
        public:
                point (int, int);                                   // constructor
                friend bool coincid(const point &, const point &);  // functie friend
};

bool coincid (const point &p1, const point &p2)
//nu mai precizez inca o data ca e prietena cu clasa
//nu e membra a clasei, deci nu scriu:  bool point::coincid (const point &p1, const point &p2)
{
        return (p1.x == p2.x && p1.y == p2.y);
} //functie independenta, prietena cu clasa
```

Apelare:

```
point poz1(2,1),  poz2(2,2),   obiect(1,1);

cout<<coincid (poz1, poz2);                                 //apelare corecta

cout<< obiect.coincid (poz1, poz2);                         //apelare incorecta
```

(b) O functie membra a unei clase este functie friend pentru alta clasa

```
//Avem doua clase: student si profesor; profesorul da nota unui student  si are acces la nota acestuia.
// studentul are o nota, poate sa isi afle nota, dar nu poate sa o modifice
#include <cstdlib>
#include <iostream>
using namespace std;

class student;          // Inainte de a folosi tipul de date student in clasa profesor, trebuie sa
                        // spunem ca exista implementat mai jos.
        // Am implementat ambele clase in acelasi fisier; cel mai bine implementam in headere separate
class prof {

    //atribute:  nume_prof,  id_prof...

public:

  int getNota(const student & ); //afla nota unui student

  void setNota(student & , int );  //da o nota unui student; trebuie sa aiba acces la atributele clasei
                                    //student; altfel - in student trebuie sa am o metoda setNota – exclus-
                                   //exact asta vrem sa nu se intample
private:

  int getNota2(const student &  );   //poate fi apelata doar intr-o metoda din clasa prof

};
```

```
class student {
private:
  //nume, id, ...
  int nota_ex_X; //nota e a studentului -> nota e atribut al tipului student

public:
        student() {
                nota_ex_X=0;
        }

         int getNota() {       //afla nota la examenul X
                return nota_ex_X;
          }

      //dar nu isi poate da nota singur

      friend void prof::setNota( student & ,int);

      friend int prof::getNota( const student & );

// prof::setNota, prof::getNota sunt functii prietene ale clasei student
// deci prof::setNota si prof::getNota au acces la toti membrii din student(inclusiv cei  privati)

// friend void prof:: getNota2(const student &); //functia era private – nu pot sa o declar friend cu student
};
  //  prof::getNota trebuie sa fie prietena cu clasa student?    NU!
```

```
int prof::getNota( const student& b )
{
                //"nota studentului b - ceruta de profesor";
                return b.nota_ex_X; //b.getNota();
 }

void prof::setNota( student& b, int x )
{
                 b.nota_ex_X=x;
}

/*int prof::getNota2( const student & b )
 {
 return b.nota_ex_X;
}*/
// ERROR – functia e declarata private in prof si nu e vizibila in clasa student unde a fost
//declarata friend => doar metodele publice pot sa fie prietene ale altei clase

int main(int argc, char *argv[])
{
   prof p; student s;
   p.setNota(s,10);
   cout<<p.getNota(s);
   cout<<s.getNota();

   return 0;
}
```

(c): Clase prietene

```
// declarare clasa prietena

class M
{
        // ..............
};

class B
{
        // ..............
        friend class M;
// vreau ca metodele publice din M sa aiba acces la toate
//atributele si  metodele din B
};
```

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

class OClasa {
friend class AltaClasa;  // declarata clasa prietena cu OClasa – seteaza si testeaza codul
//private:
  int cod;  //ex. un cod de acces – nu il seteaza singura
public:
   int getCod() { return cod; }
};

class AltaClasa {
//toate functiile membre din aceasta clasa au acces la atributele si metodele clasei OClasa
public:
  void modifica( OClasa& oc, int x ){oc.cod = x;}
  void testeaza( OClasa& oc  ){ if (oc.cod == 0) cout<<“cod invalid”;}
};

int main() {
  OClasa oc1;
  AltaClasa ac1;

  ac1.modifica( oc1, 5 );
  ac1.testeaza( oc1);
  cout<<oc1.getCod(); //5

  return 0;
}
```

Functii si clase friend

Dezavantaje:

- **Se strica incapsularea datelor.**
- **Se creaza dependente periculoase.**

Avantaje:

- **Diminueaza timpul de executie (am acces direct la atribute).**
- **Permit supradefinirea operatorilor clasici (<<, >>, +, -)**
- **Permit interactiuni facile intre diferite entitati.**

E de preferat solutia fara folosirea functiilor/claselor friend - daca implementarea permite.

Dar exista si cazuri cand functiile friend trebuie neaparat folosite.

In Java nu exista functii friend – strica incapsularea datelor; Java –lb. perfect orientat pe obiect

C4: FRIEND si supradefinirea operatorilor

Supradefinirea operatorilor

Ganditi-va la "+" – face lucruri diferite in functie de tipurile de date pentru care e folosit.

Am putea sa aplicam "+" pentru obiecte de tipuri de date create de noi (class)?

Da – pentru complex chiar are sens.

Cum? 1.Am putea sa implementam niste functii care sa ii mimeze comportamentul

```
friend const complex  aduna(const complex &x, const complex& y)
{return complex(x.re+y.re,x.im+y.im);}
```

Cu apel:
```
complex a, b(1,1), c(2,2);
a=aduna(b,c);
```

DAR

Ar fi mai clar si mai simplu daca am putea sa folosim chiar notatia de operator +
As vrea sa pot scrie **a=b+c**;

C++ permite supradefinirea operatorilor (sunt niste functii)

Nu putem crea operatori noi, dar putem sa ii supradefinim pe cei existenti pentru tipurile de date de baza in tipurile definite de noi (in clase). (Ce operatori cunoasteti?)

Avantaje

– permite claselor sa puna la dispozitie o semantica naturala pentru operatii intr-o maniera similara cu cea oferita pentru tipurile de baza (ex: pentru clasa complex)

Dezavantaje

– implementarea neadecvata si folosirea gresita conduce la un cod confuz

Ex: de implementare gresita pentru + : a+b – modifica valoarea lui a; utilizatorul nu se astepta la asta

– multe clase nu pot sa supradefineasca semantica naturala ceea ce conduce la supradefinirea de operatori in mod confuz = > **ii supradefinim doar daca are sens**

Student a, b,c; a=b+c; //Are sens???????

Si din cauza acestor confuzii si a potentialelor implementari nedorite - Java omite aceasta facilitate.

Operatorii sunt tratati ca niste functii:

- au nume: **operator**+, **operator**-, etc
- primesc argumente/parametri (numite operanzi)
- returneaza valori
- permit celui care face implementarea sa defineasca semantica (ce fac ei exact)

DAR

Operatorii pot sa fie invocati ca o functie: operator+(a,b) SAU scriind direct a+b.

Tipuri de operatori

Operatori unari (cu un argument/operand):

- utilizare: -a, ++b, b++

Operatori binari (cu doua argumente/operanzi)

- utilizare: a + b, c <= d, a= b

Notiuni de interes in contextul utilizarii si supradefinirii operatorilor:

-asociativitatea (de la dreapta la stanga sau stanga la dreapta)

-precendenta (ordinea in care se executa)

-numarul de operanzi

a * b + c => ordinea este

1. p=a * b
2. p + c

```
int x=2,y=2;
x+=y+2; //?
```

Cum functioneaza operatorul +?

a+b+c+d

- Pentru **adunare** asociativitatea e implicit **de la stanga la dreapta (LR);**

  a+b+c+d⇔(((a+b)+c)+d)

OBS! + nu modifica operanzii

Prioritate mare

Categoria	Operatorii	Semnificație	Mod de asociere
Primari	( ) [ ] . ->	Apel funcție, selecție	stânga - dreapta
Unari	++ -- + - ! ~ (tip) & * sizeof	Incrementare/decrementare Stabilire semn Negare logică / bit cu bit Conversie explicită de tip Adresa Conținutul adresei Dimensiune în octeți	dreapta - stânga
Multiplicativi	* / %		stânga - dreapta
Adunare, scădere	+ -		stânga - dreapta
Deplasare (nivel bit)	<< >>		stânga - dreapta
Relaționali	< > <= >=		stânga - dreapta
Testare egalitate	== !=		stânga - dreapta
ȘI (nivel bit)	&		stânga - dreapta
SAU exclusiv (nivel bit)	^		stânga - dreapta
SAU inclusiv (nivel bit)	|		stânga - dreapta
ȘI logic	&&		stânga - dreapta
SAU logic	||		stânga - dreapta
Condițional (operator ternar)	? :		stânga - dreapta
Atribuire	= += -= *= /= %= <<= >>= &= ^= |=		dreapta - stânga
Virgula	,		stânga - dreapta

Prioritate mica

Cum supradefinim operator+ pentru tipul complex?

//ca functie membra

```
const complex complex::operator+ ( const complex & a ) const
{
return complex ( re + a. re, im + a.im );
}
```

// sau cu functie friend – solutia preferata pentru ca mimeaza cel mai bine +

```
friend const complex  operator+ ( const complex & a , const complex & b)
{
return complex ( a.re + b. re, a. im + b.im );
}
```

Observatie: **Pentru a face supradefinirea unui operator pentru un tip de date, cel putin un operand trebuie sa fie membru al clasei sau functia operator trebuie sa fie membra a clasei.**

Ce operatori se pot supradefini?

Aproape toti:

- aritmetici: +, -, *, /, ++, --
- pentru operatii pe biti: &,|, >>, <<
- pentru operatii logice: &&, ||, !
- pentru comparare: >, <, <=, >=, ==, !=
- indexare, functie, enumerare: [], (), ,
- atribuire si operatii compuse cu atribuire: =, +=, -=, *=, /=, <<=, >>=

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

Ce nu putem supradefini si restrictii:

- acces la membrii clasei : .
- operatorul de rezolutie: ::
- operatorul conditional (3 operanzi): ?:
- operatorul sizeof

Nu pot sa fie creati operatori noi: |x|, y := x, y = x ** 2

Cel putin un operand trebuie sa fie membru al clasei sau functia operator sa fie membra a clasei!

Supradefinirea trebuie sa aiba sens.

Pentru supradefinirea operatorilor nu avem voie sa schimbam:

- **Precedenta**
- **Asociativitatea (de la dreapta la stanga sau de la stanga la dreapta)**
- **Numarul de operanzi**

In ce maniera se pot supradefini operatorii?

1. ca functii membre ale clasei
2. ca functii friend ale clasei
3. ca functii externe clasei (rar si nerecomandat)

1. - o functie membra a clasei primeste in mod implicit adresa obiectului pt. care este apelata, de aceea prototipul este:

TR operator ☺ (T);
binari: a☺b

TR operator ☺ ();
unari : ☺a, sau a☺

- operandul din stanga e de tipul clasei din care face parte operatorul
- functia primeste ca argument un parametru de tipul clasei - in cazul operatorilor binari, si niciun alt argument - pentru operatorii unari.

2. - au nevoie de parametri pentru amandoi operanzii (daca operatorii sunt binari)
- pot primi operanzi de alt tip decat clase cu care sunt prietene
- trebuie sa aiba cel putin un operand de tipul clasei cu care sunt prietene

friend TR operator ☺ (T sau X, T sau X);
//dar nu simultan de tip X
binari: a☺b

friend TR operator ☺ (T);
unari : ☺a, sau a☺

*TR – tip de date returnat; T – tipul de date pt care se face supradefinirea operatorului

Operator+ pentru tipul complex - cu al doilea operand de tip double

```
//functie membra
const complex  complex::operator+(const double & a ) const
{
return complex (re+a, im);
}

// sau ca functie friend
friend const complex operator+( const complex & c1 , const double & c2)
{
return complex (c1.re+c2, c1.im);
}

//Apel:
//complex a(3,4),r; double x=4.11;
//r=a+x;//ce se intampla aici?
//pot sa apelez si asa: c=a.operator+(x); si respectiv: c=operator+(a,x);
```

Ce faceam daca doream ca primul operand sa fie double?

```
friend const complex  operator+( const double & c1 , const complex & c2)
{
return complex (c1+c2.re, c2.im);
}
// Putea sa fie implementat ca functie membra?
```

Situatii intalnite si moduri de apelare:

a) Operator binar – functie membra

a☺b <=> a.operator ☺(b)

b) Operator binar – functie friend

a☺b <=> operator ☺(a, b)

c) Operator unar – functie membra

a☺ <=> a.operator ☺()

☺a

d) Operator unar – functie friend

a☺ <=> operator ☺(a)

☺a

Supradefinirea operatorilor unari

– ca functii **nestatice membre** fara argumente
– ca functii friend cu un argument

•Argumentul trebuie sa fie obiect de tipul clasei sau referinta catre un obiect de tipul clasei

!Functiile statice acceseaza doar date statice

Supradefinirea operatorilor binari

– ca functii **nestatice membre** cu un argument
– ca functii ne membre cu doua argumente

•Cel putin un argument trebuie sa fie de tipul clasei sau referinta catre un obiect de tipul clasei

Observatie

Aproape toti operatorii supradefiniti - ca functii membre trebuie sa fie const (mai putin ce care implica atribuiri) deoarece nu trebuie sa modifice argumentul implicit

- trebuie sa aiba argumente constante

```
const complex complex::operator+ ( const complex & a ) const
{
return complex ( re + a. re, im + a.im );
}

complex a ( 1, 2);
complex b ( 2, 3);
complex c ( 2, 3);
complex d = a + b + c;
 // a, b si c nu trebuie sa se modifice;
// rezultatul a+b e stocat intr-un obiect temporar care nu trebuie sa poata fi modificat: (a+b).set(2,2);
```

Forme unare si binare ale aceluiasi operator

Atata forma unara cat si cea binara a unui operator poate sa fie supradefinita
– se face diferenta prin numarul de argumente
– ex: - ; pot sa il folosesc ca -a sau a-b

```
const complex complex::operator- () const
{
return complex ( -1 * re, -1 * im);
}
```

Apel : complex x(2,3),y;
y=-x; // sau y=x.operator-();

```
const complex complex::operator- ( const complex & b ) const
{
return complex ( re– b.re, im- b.im );
}
```

Apel : complex x(2,3),y(4,3),z;
z=y-x; // sau z=y.operator-(x);

SAU:

```
friend const complex operator- (const complex & a, const complex & b)
{return complex ( a.re– b.re, a.im- b.im );
}
```

Apel : complex x(2,3),y(4,3),z; z=y-x; // sau z=operator-(y,x);

Atentie!!!

```
complex & complex::operator- ()
{
re *= -1;
im*= -1;
return *this;
}
// Gresit


const complex  complex::operator- () const
{
return complex ( -1 * re, -1 * im);
}
// Corect

complex a ( 1, 2);
complex b = -a; // operatorul- nu il modifica pe a
```

Atentie!!! operator<,>,<=,>=,==,!=

Nu toate entitatile admit relatii de ordine.

Ex: Nr. complexe nu se pot ordona.

Daca fac implementarea pentru oricare dintre operatorii de mai sus este indicat sa ii implementez si pe restul.

Utilizatorul, o data ce vede ca unul poate fi folosit, incearca sa ii foloseasca si pe ceilalti (in plus unele compilatoare ii genereaza automat si nu suntem siguri daca ii genereaza cu implementarea dorita de noi).

– operator<= (!(a > b) implica a <= b)

=>implementare rapida

– operator== (!(a != b) implica a == b)

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

class ora_ex
{
     int ora;
     int min;
     int sec;
public:
      ora_ex(int o=0, int m=0, int s=0)
      {
              ora=o;
              min=m;
              sec=s;
       }

       int inSec() const                  //vedeti ce se intampla daca functia
                                          //nu garanteaza ca datele nu vor fi modificate
       {
        return ora*3600+min*60+sec;
         }
```

```
bool operator<(const ora_ex &oe)const
{  return (inSec()<oe.inSec());
 }
```

```
 /*friend bool operator<(const ora_ex &oe1, const ora_ex &oe
 {  return (oe1.inSec()<oe2.inSec());
}
```

```
bool operator>=(const ora_ex &oe) const
{  return (!(*this<oe));
}
```

```
friend bool operator>=(const ora_ex &oe1, const ora_ex &oe2)
 {  return (!(oe1<oe2));
}
```

```
bool operator==(const ora_ex &oe) const
 { return (inSec()==oe.inSec());
}
```

```
friend bool operator==(const ora_ex &oe1, const ora_ex &oe2)
 {  return (oe1.inSec()==oe2.inSec());
  }
```

```
bool operator!=(const ora_ex &oe) const
{  return (!(*this==oe));
}//as putea sa le declar inline
```

```
friend bool operator!=(const ora_ex &oe1, const ora_ex &oe2)
 {  return (!(oe1==oe2));
}*/
```

```
};
int main(int argc, char *argv[])
{   ora_ex oe1(3,20,4), oe2(4,20,0);
    cout<<(oe1<oe2)<<endl;
    cout<<(oe1>=oe2)<<endl;
    cout<<(oe1==oe2)<<endl;
    cout<<(oe1!=oe2)<<endl;
    //se mai pot apela si asa (daca fct membre):
    cout<<(oe1.operator<(oe2))<<endl;
    cout<<(oe1.operator>=(oe2))<<endl;
    cout<<(oe1.operator==(oe2))<<endl;
    cout<<(oe1.operator!=(oe2))<<endl;
```

```
//sau asa daca sunt fctii friend
// cout<<(operator<(oe1,oe2))<<endl;
// cout<<(operator>=(oe1,oe2))<<endl;
// cout<<(operator==(oe1,oe2))<<endl;
// cout<<(operator!=(oe1,oe2))<<endl;

return 0;
  }
```

Operator ++

Prefix (++x)

Ex: int x=3;

```
cout<<++x;//4
cout<<x;//4
```

Supradefinire:

- ca functie membra : const T& T::operator++();
- ca functie friend: friend const T& operator++(T&);

In ambele cazuri returneaza referinta constanta la obiectul incrementat.

Postfix (x++)

Ex: int x=3;

```
cout<<x++;//3
cout<<x;//4
```

Supradefinire:

- ca functie membra : const T T::operator++(int);
- ca functie friend: friend const T operator++(T& , int);

Se foloseste parametrul int (0) pentru a distinge intre forma prefixata si cea postfixata

- obj++ se traduce in obj.operator++(0)

Returneaza un obiect care contine valoarea originala – inainte de incrementare; dupa care se face modificarea

```
const ora_ex& operator++()   //forma prefixata
{
 this->sec++;
 return *this;
}

const ora_ex operator++(int i)  //forma postfixata
{
   ora_ex aux(*this);   //fac o copie a obiectului nemodificat
   this->sec++;  //modific valoarea sa
   return aux;   //returnez copia cu valoarea originala
}

void afis() const                          //ce se intampla daca nu era declarata const?
 {cout<<ora<<":"<<min<<":"<<sec<<endl;}
```

Pot sa am vreo problema cu aceasta implementare?

Da, ar trebui sa testez daca sec>59, etc. Tema!

```
  Utilizare:
    ora_ex oe1(3,20,4);
     (++oe1).afis();

     (oe1++).afis();
     oe1.afis();
```

operator=

• Compilatorul C++ genereaza intotdeauna* metoda operator=

– chiar si daca nu ii este cerut

– face o copiere bit cu bit a obiectelor

– asigura copierea integrala in cazul atributelor de tip de baza (deep copy)

– face copii superficiale (shallow copy) in cazul atributelor de tip pointer, carora trebuie sa le alocam spatiu de memorie

* ***http://www.cplusplus.com/doc/tutorial/classes2/*** ***("Tema de studiu individual")***

• Toate clasele (ce contin atribute de tip pointer carora urmeaza sa li se aloce/elibereze spatiu de memorie) ar trebui sa puna la dispozitie implementarea specifica pentru operator= pentru a evita confuziile si comportamentele nedorite/default

• shallow vs. deep copy (ca si in cazul constructorilor de copiere generati automat)

Cum functioneaza?

a=b=c=d
De fapt secventa este asta:
c=d si se returneaza valoarea lui c (intr-un obiect temporar; cine face copia?)
b=c si se returneaza valoarea lui b
a=b si se returneaza valoarea lui a
<=>atribuire de la dreapta la stanga (RL) ; se returneaza mereu variabila modificata

```
complex & operator= ( const complex & a )
{
re = a.re;
im = a.im;
return *this;
}

//Utilizare:
complex a(1,3),b(4,5);
a=b;
a.afisare();
b.afisare();
```

De ce nu e recomandata transmiterea parametrului ca referinta ne-constanta?

```
complex & operator= (complex & a )//Nu asa
{
re = a.re;
im = a.im;
//this->set(a.re, a.im);
return *this;
}
```

Problema este urmatoarea. Daca incerc o atribuire de tipul:

```
complex c;
c=complex(1,3);
```

codul nu va functiona. De ce? Care e tipul pentru operandul din dreapta?

In partea dreapta a expresiei de atribuire avem un obiect temporar (fara nume).

C++ interzice compilatorului sa transmita un obiect temporar printr-un parametru de tip referinta ne-constanta (pentru ca sa protejeze acel obiect temporar de modificari care oricum s-ar pierde odata cu disparitia lui; de fapt e protejat programatorul)

Cand se apeleaza operator= si cand constructorii?

Atentie!!!

```
complex a(1,2);
complex v=a;
```

!!!De fapt, in acest caz, nu se apeleaza operator= ci in functie de standard un constructor sau alt tip de operator=.

O sa revenim la aceasta discutie.

*** http://www.cplusplus.com/doc/tutorial/classes2/**

In cazul:

```
complex v;
v=a;
```

!!! In prima faza se apeleaza constuctorul pentru v si apoi operatorul=

Test de autoatribuire

Considerati clasa vector:

```
class vector
{   int n;
    int *buf;
public:
    vector(int n)
     { dim=n;
       if (n<=0) buf=NULL;
       else{
          buf=new int[n];
          for (int i=0; i<n;i++) buf[i]=0;
      }
     }

     vector& operator=(const vector& v)
    {   n=v.n;
        if (buf!=NULL) delete [] buf;
        if (n<=0) buf=NULL;
         else{     buf=new int[v.n];
                   for (int i=0; i<n;i++) buf[i]=v.buf[i]; }
      return *this;
    }
};//ce mai trebuie neaparat implementat pentru clasa vector?
```

```
//Utilizare
vector c(3);
c=c; //ce se intampla?
```

Implementare precauta

```
vector& operator=(const vector& v)
 {
      if(this!=&v)       //verific sa nu fie acelasi obiect – testez adresele
     {      n=v.n;
            if (buf!=NULL) delete [] buf;
            if (n<0) buf=NULL;
            else{
                buf=new int[n];
                for (int i=0; i<n;i++)
                        buf[i]=v.buf[i];
             }
      }
  return *this;
}
```

Problema 1

Supradefiniti operatorii: *, +=, +, ==, != pentru clasa complex.
Supradefiniti operatorii: *, +=, + cu al doilea/primul operand de tip double/int pentru clasa complex.

1. operator*

```
//ca functie membra
const complex  complex::operator*( const complex & a ) const
{
return complex (re*a.re-im*a.im, re*a.im+im*a.re);
}

// sau ca functie friend
friend const complex   operator*( const complex & c1 , const complex & c2)
{
return complex (c1.re*c2.re-c1.im*c2.im, c1.re*c2.im+c1.im*c2.re);
}

//Apel:
//complex a(3,4), b(1,2);
//complex c=a*b;//ce se intampla aici?
//pot sa apelez si asa: c=a.operator*(b); si respectiv: c=operator*(a,b);
//de ce returnez o const?
//(a+b).f(); //f face niste modificari pe obiectul care o apeleaza; dar obiectul dispare
//ma asigur ca decat functiile const pot sa fie apelate de catre obiectele temporare
```

2. operator+=

```
//cum functioneaza?   a+=b+=c;

complex & complex::operator+= ( const complex & a )
{
re=re+a.re;
im=im+a.im;
return *this;
}

// Apel:
//complex a(1,1),b,c;
b+=a;
//sau
//b.operator+=(a);
```

De ce transmit operanzii prin referinta const?
Vreau sa pot sa scriu si ceva de genul: **b+=(complex(1,1)); sau b+=(c-a);**
Daca transmiteam prin referinta neconstanta=>**ERROR** pt ca complex(1,1) e obiect temporar – deci constant.

De asemenea vreau sa ma asigur ca, in general, nu modific valoarea operandului din dreapta.

Observatie: Am supradefinit operatorii ca sa ii folosesc cu semantica lor naturala: +,-…

3. operator== si !=

```
bool operator==(const complex& a) const
 { return ((re==a.re)&&(im==a.im));
}
```

sau:

```
friend bool operator==(const complex& a, const complex& b)
 { return ((a.re==b.re)&&(a.im==b.im));
}
```

```
bool operator!=(const complex& a) const
{  return (!(*this==a));
}
```

sau:

```
friend bool operator!=(const complex& a, const complex& b)
{  return (!(a==b));
}
```

//De ce transmit referinte constante? Stergeti cuvintele const si incercati sa rulati:
complex a(1,1); cout<<(a==(complex(3,3)));

Probleme propuse

1. Pentru clasa Angajat/Student implementati operatorul =.
   Implementati == si !=, eventual <,>,<=,>= (sau macar operatorul folosit la sortare).

2. Supradefiniti operatorii care au sens sa fie supradefiniti pentru clasele Vector, Matrice, Complex, Ora_ex, Fractie, Multime.