SUPRAÎNCĂRCAREA OPERATORILOR

11

11.1. Moduri de supraîncărcare a operatorilor
11.1.1. Supraîncărcarea prin funcții membre
11.1.2. Supraîncărcarea prin funcții prietene
11.2. Restricții la supraîncărcarea operatorilor
11.3. Supraîncărcarea operatorilor unari
11.4. Membrii constanți ai unei clase
11.5. Supraîncărcarea operatorilor insertor şi extractor
11.6. Supraîncărcarea operatorului de atribuire =
11.7. Supraîncărcarea operatorului de indexare [ ]
11.8. Supraîncărcarea operatorilor *new* şi *delete*
11.9. Supraîncărcarea operatorului ( )
11.10. Supraîncărcarea operatorului ->
11.11. Conversii

11.1. MODURI DE SUPRAÎNCĂRCARE A OPERATORILOR

Supraîncărcarea (supradefinirea, termenul ***overloading***) operatorilor permite atribuirea de noi semnificații operatorilor uzuali (operatorilor intâlniți pentru tipurile de date predefinite). Aşa cum am subliniat în numeroase rânduri, clasa reprezintă un tip de date (o mulțime de valori pentru care s-a adoptat un anumit mod de reprezentare şi o mulțime de operații care pot fi aplicate acestora). Astfel, operatorul + foloseşte la adunarea a două date de tip int, float sau double, însă aceluiaşi operator i se poate atribui semnificația de "alipire" a două obiecte de tipul `şir`, sau de adunare a două obiecte de tipul `complex`, `vector` sau `matrice`.

Observație: Operatorii sunt deja supradefiniți pentru a putea opera asupra mai multor tipuri de bază (de exemplu, operatorul + admite operanzi de tip int, dar şi float sau double), sau pot avea seminificații diferite (de exemplu, operatorul * poate fi folosit pentru înmulțirea a doi operanzi numerici sau ca operator de deferențiere, operatorul >> poate avea semnificația de operator extractor sau operator de deplasare pe bit).

Prin supraîncărcarea operatorilor, operațiile care pot fi executate asupra instanțelor (obiectelor) unei clase pot fi folosite ca şi în cazul tipurilor de date predefinite.

Exemplu: Pentru clasa punct (vezi capitolul 10), putem atribui operatorului + semnificația: expresia `a+b` (a, b sunt obiecte din clasa punct) reprezintă "suma" a două puncte şi este un punct ale cărui coordonate sunt date de suma coordonatelor punctelor a şi b. Astfel, supradefinirea operatorului + constă în definrea unei ***funcții*** cu numele: `operator +`

```
tip_val_întoarsă operator op (lista_declar_parametri)
{
      // . . . .                corpul funcției
}
```

Deci, limbajul C++ permite supradefinirea operatorului **`op`** prin definirea unei funcții numite

`operator op`

Funcția trebuie să poată accesa datele membre private ale clasei, deci s***upradefinirea operatorilor*** se poate realiza în ***două moduri***:
- printr-o ***funcție membră*** a clasei;
- printr-o ***funcție prietenă*** a clasei.

11.1.1. SUPRAÎNCĂRCAREA OPERATORILOR PRIN FUNCȚII MEMBRE

În situația în care supraîncărcarea operatorului + se realizează printr-o funcție membră, aceasta primeşte ca parametru implicit adresa obiectului curent (pentru care este apelată). Deci primul operand al operatorului este transmis implicit.

Exemplu:
```
class punct{
      double x, y;
public:
      //......
      punct operator + (punct);
};
//Metodele clasei punct……………………

punct punct::operator + (punct a)
{     punct p;
      p.x=x + a.x;       //echivalent cu p.x=this->x+a.x;
      p.y=y + a.y;       //echivalent cu p.y=this->y+a.y;
      return p;
}
void main()
{     punct A(1.1, 2.2); A.afişare();
      punct B(-5.5, -6.6); B.afişare();
      punct C;
      C=A+B;  C.afişare();
      C=A+B+C; C.afişare();
}
```

Expresia `C=A+B` este interpretată ca `C = A.operator + (B)`.
Expresia `C=A+B+C` poate fi interpretată, în funcție de compilator, astfel:
Unele compilatoare crează ***un obiect temporar T***: `T = A.operator + (B)`
`C = T.operator + (C)`
Alte compilatoare interpretează expresia ca: `C=(A.operator + (B)).operator + (C)`.

11.1.2. SUPRAÎNCĂRCAREA OPERATORILOR PRIN FUNCȚII PRIETENE

Fie clasa punct definită anterior. Reamintind faptul că funcțiile prietene au acces la membrii privați ai unei clase, însă **nu** primesc ca argument implicit pointerul către obiectul curent (`this`), să supraîncărcăm operatorul + printr-o funcție prietenă a clasei punct:
```
class punct{
      double x, y;
public:
      //......
      friend punct operator + (punct, punct);
};
//Metodele clasei punct…………………….
punct operator + (punct a, punct b)
{     punct p;
      p.x=a.x + b.x;  p.y=a.y + b.y;
      return p;
}
void main()
{     punct A(1.1, 2.2); A.afişare();
      punct B(-5.5, -6.6); B.afişare();
      punct C;
      C=A+B;  C.afişare();
      C=A+B+C; C.afişare();
}
```

Expresia `C=A+B` este interpretată de compilator ca `C=operator + (A, B)`.
Expresia `C=A+B+C` este evaluată ținându-se cont de regulile de prioritate şi de asociativitate a operatorului: `(A+B)+C` , ceea ce conduce la un apel de forma: `operator + (operator + (A, B), C)`.

Observație: În exemplul anterior, transmiterea parametrilor către funcția prietenă de supraîncărcare a operatorului + se realizează prin *valoare*. Parametrii pot fi transmişi şi prin *referință*, pentru a evita crearea (în momentul apelului funcției) unor copii locale ale parametrilor efectivi în cei formali. La transmiterea parametrilor prin referință, funcția operator + are prototipul:

```
punct operator + (punct &, punct &);
```

Pentru a proteja argumentele transmise prin referință la eventualele modificări, se poate folosi *modificatorul de acces* ***const***: `punct operator + (const punct &, const punct &);`

11.2. RESTRICȚII LA SUPRAÎNCĂRCAREA OPERATORILOR

Supraîncărcarea operatorilor se poate realiza, deci, prin funcții membre sau funcții prietene. Dacă supraîncărcăm acelaşi operator printr-o metodă şi printr-o funcție prietenă, funcția prietenă va avea, întotdeauna, un parametru în plus față de metodă (deoarece funcției prietene nu i se transmite ca parametru implicit pointerul `this`).

Totuşi, supraîncărcarea operatorilor este supusă următoarelor restricții:

- Se pot supraîncărca doar operatorii existenți; nu se pot crea noi operatori.
- Nu se poate modifica aritatea (numărul de operanzi) operatorilor limbajului (operatorii unari nu pot fi supraincărcați ca operatori binari, şi invers).
- Nu se poate modifica precedența şi asociativitatea operatorilor.
- Deşi operatorii supraîncărcați păstrează aritatea şi precedența operatorilor predefiniți, ei ***nu*** moştenesc şi ***comutativitatea*** acestora.
- Nu pot fi supraîncărcați operatorii **. ::?** şi **:**

Observații:

- În tabelul 2.8. (capitolul 2) sunt prezentați operatorii existenți, precedența şi asociativitatea acestora.
- Dacă operatorul **=** nu este supraîncărcat, el are o semnificație implicită.
- Operatorii **, new delete [ ] ->** şi **cast** impun restricții suplimentare care vor fi discutate ulterior.
- Funcția operator trebuie să aibă cel puțin un argument (implicit sau explicit) de tipul clasei pentru care s-a supraîncărcat operatorul. Astfel:
  - ✓ La supraîncărcarea unui operator unar printr-o funcție membră a clasei, aceasta are un argument implicit de tipul clasei (obiectul care îl apelează) şi nici un argument explicit. La supraîncărcarea operatorului unar printr-o funcție prietenă, aceasta are un argument explicit de tipul clasei.
  - ✓ La supraîncărcarea unui operator binar printr-o funcție membră a clasei, aceasta are un argument implicit de tipul clasei (obiectul care îl apelează) şi un argument explicit. La supraîncărcarea operatorului binar printr-o funcție prietenă, aceasta are două argumente explicite de tipul clasei.
- Se poate atribui unui operator orice semnificație, însă este de dorit ca noua semnificație să fie cât mai apropiată de semnificația naturală. De exemplu, pentru adunarea a două obiecte se poate supraîncărca operatorul * , dar este mai naturală folosirea operatorului + cu semnificația de adunare.
- În cazul supradefinirii operatorilor, nu se poate conta pe comutativitatea acestora.
  De exemplu, dacă se supraîncarcă operatorul + pentru clasa complex printr-o funcție prietenă a clasei complex: `complex operator + (complex, double)`
  Operatorul poate fi folosit în expresii cum ar fi: `a+7.8` (a este obiect al clasei complex), dar nu în expresii ca: `7.8 + a`.
- Dacă un operator trebuie să primească ca prim parametru un tip predefinit, acesta nu poate fi supradefinit printr-o funcție membră.
- Operatorii care prezintă şi alte particularități, vor fi tratați separat (vezi 11.7.,11.8., 11.10., 11.11.).
- În principiu, metodele care supraîncarcă un operator nu sunt statice. Excepția o constituie operatorii **`new`** şi **`delete`** (vezi 11.8.).
- Diferența între forma prefixată şi postfixată, la supraîncărcarea operatorilor predefiniți ++ şi --, se poate face doar de anumite compilatoare (de exemplu, compilatorul de BorlandC, versiune>3.0, se poate face diferența).

11.3. SUPRAÎNCĂRCAREA OPERATORILOR UNARI

Operatorii unari pot fi supradefiniți printr-o funcție membră nestatică (fără parametri expliciți) sau printr-o funcție prietenă cu un parametru explicit de tipul clasă.
Ca exemplu, să supraîncarcăm operatorul unar ++ pentru clasa punct, pentru a putea fi folosit atât în formă prefixată, cât şi postfixată (doar pentru compilatoarele care permit acest lucru!!). Vom folosi clasa punct implementată anterior, cu modificarea ca datele membre sunt de tipul int.

```
class punct{
      int x, y;
public: //...
      punct & operator ++ (int );           //forma postfixată
      punct & punct::operator++();          //forma prefixată
};

punct punct::operator ++ (int)
{punct p=*this;x++; y++;return p;}

punct  & punct::operator++()
{x++;y++;  return *this;}

void main()
{ punct A(11, 10); punct C=A++;A.afişare( ); C.afişare( );
punct C=++A;A.afişare( ); C.afişare( );
}
```

11.4. MEMBRII CONSTANȚI AI UNEI CLASE

Aşa cum s-a subliniat în capitolul 10, o clasă poate avea *membrii statici*: *date membru statice* (figurează într-un singur exemplar pentru toate instanțele clasei) sau *metode statice* (nu li se transmite pointerul this şi pot modifica doar date membru statice). Deasemenea, o clasă poate avea **metode constante.** O metodă este declarată constantă prin utilizarea modificatorului **`const`** în antetul ei (vezi exemplul de la pagina 150), după lista parametrilor formali. Metodele constante nu modifică obiectul pentru care sunt apelate.

Ca oricăror variabile de tip predefinit, şi **obiectelor** de tip definit de utilizator li se poate aplica modificatorul const. Pentru **un obiect constant** este permis doar apelul metodelor constante, a constructorilor şi a destructorilor.

11.5. SUPRAÎNCĂRCAREA OPERATORILOR INSERTOR ŞI EXTRACTOR

Operatorul **<<** se numeşte *operator insertor*, deoarece inserează date în stream-ul (fluxul) de ieşire. Operatorul **>>** se numeşte *operator extractor*, deoarece extrage date din stream-ul (fluxul) de intrare.
În exemplul următor, aceşti operatori sunt supraîncărcați pentru clasa complex, astfel încât să poată fi folosiți ca pentru obiectele de tip predefinit.

Exemplu:

```
complex z1, z2;
cin>>z1>>z2;                 //extrage valorile lui z1 şi z2
cout<<"z1="<<z1<<'\n';    //inserează şir constant, apoi valoarea lui z1
cout<<"z2="<<z2<<'\n';    //inserează şir constant, apoi valoarea lui z2
```

Deoarece întotdeauna operandul stâng este de tip **`istream`** (cin este obiect predefinit, de tip istream) sau **`ostream`** (cout este obiect predefinit, de tip ostream), şi nu de tipul introdus prin clasă, operatorii << şi >> pot fi supraîncărcați **numai** prin **funcții prietene**. Prototipurile operatorilor sunt:

```
friend ostream &operator << (ostream &,const complex&);//operator afişare complex
friend istream & operator >> (istream &,complex&);      //operator citire complex
```

Definițiile funcțiilor operator:
```
ostream &operator<<(ostream &ecran, const complex &z)
      {ecran<<"("<<z.re;
      if (z.im>=0)    ecran<<'+';ecran<<z.im<<"*i)"; return ecran;}
istream &operator>>(istream &tastatura, complex &z)
      {tastatura>>z.re>>z.im;return tastatura;}
```

Prototipurile funcțiilor operator `<< și >>` pentru un tip abstract **`tip`**, sunt:
```
friend ostream &operator<<(ostream &,const tip&);
friend istream &operator >> (istream &,tip&);
```

11.6. SUPRAÎNCĂRCAREA OPERATORULUI DE ATRIBUIRE =

În cazul în care operatorul de atribuire nu este supraîncărcat explicit, compilatorul generează unul implicit (ca în exemplul clasei punct sau segment). În absența unei supraîncărcări explicite, operatorul copie valorile datelor membre ale operandului drept în datele membre ale operandului stâng.

Exemplu:

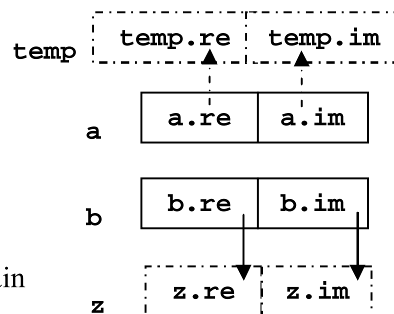
`punct a (8,9), b;`

`b=a;` /* operator atribuire implicit: zona de memorie ocupat de a se copie, ***bit cu bit***, în zona de memorie ocupată de b: `b.x=a.x` si `b.y=a.y` */

Operatorul de atribuire implicit este nesatisfăcător în situațiile în care obiectele clasei au ca date membre pointeri, sau în situațiile în care memoria este alocată în mod dinamic.

O supraîncărcare explicită a operatorului pentru clasa complex (ambii operanți de tip complex) poate fi făcută fie prin metodă, fie prin funcție prietenă.

```
class complex
{     double re,im;
public:
      complex operator = (complex );
};
complex complex::operator = (complex z)
      { re=z.re; im=z.im; return *this;
      //this este pointer către obiectul curent, a în main
      }
void main()
      {complex a, b;
      a = b;   //a.operator=(b); (figura 11.1.)
      }
```



(Obiectul z este local funcției operator=)

Figura 11.1. Supraîncărcarea operatorului = prin metodă a clasei complex

Deoarece funcția `operator=` returnează valoare de tip complex, se construiește un obiect temporar `temp`, a cărui valoare se atribuie lui `a`.

O altă modalitate, mai eficientă, de a supraîncărca operatorul de atribuire prin metodă a clasei complex, este aceea prin care funcția primește ca parametru referință către operandul drept (se lucrează, astfel, chiar cu obiectul `b`, deoarece `z` și `b` sunt variabile referință; în plus, modificatorul `const` interzice modificarea operandului transmis ca parametru referință; în plus, nu se mai crează obiectul local z, se ia ca referință obiectul existent) și returnează o referință (adresa obiectului `a`), așa cum prezintă figura 11.2.

```
complex &complex::operator = (const complex &z)
      { re=z.re; im=z.im; return *this;}
void main()
      {complex a, b;
      a = b;   //a.operator=(b); (figura 11.2.)
      }
```

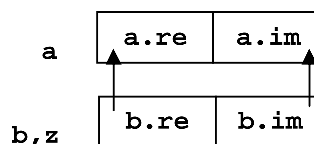


Figura 11.2. Supraîncărcarea operatorului = prin metodă a clasei complex

Deasemenea, operatorul binar de atribuire poate fi supraîncărcat prin funcție prietenă (în acest caz, nu primeşte parametrul implicit `this`, deci are doi operanzi).
Paramertrii `z1`, `z2` sunt transmişi prin referință, deci se lucrează chiar cu obiectele `a`, `b`. Funcția returnează adresa obiectului a. Modificatorul const interzice modificarea operandului drept.

```
class complex
      { double re,im;
public:
      friend complex&operator=(complex&,const complex&);       //funcție prietenă constantă
};
complex & operator = (complex &z1, complex &z2)
      { z1.re=z2.re; z1.im=z2.im; return z1;}
void main()
      {complex a, b;
      a = b;   //a.operator=(b);  (figura 11.3.)
      }
```

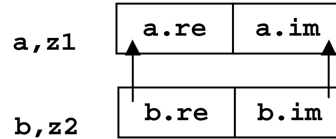


Figura 11.3. Supraîncărcarea operatorului = prin funcție prietenă a clasei complex

Deoarece întotdeauna operandul stâng al operatorului de atribuire este de tipul clasei pentru care se supraîncarcă, este preferabil ca supraîncărcarea să se realizeze prin metodă a clasei. Reamintim că asociativitatea operatorului este de la dreapta la stânga. Operatorul poate apare în expresii de forma: `a=b=c=d;`

```
//FISIERUL complex.h
#define PI      3.14159265358979
#include <iostream.h>
class complex
      { double re,im;
public:
      complex (double r=0,double i=0);     //constructor
      complex (const complex&);            //constructor copiere
      ~complex(){cout<<"Destructor complex("<<re<<","<<im<<")\n";}    //destructor
      double modul();              //metoda care returneaza modulul unui complex
      double arg();                //metoda care returneaza argumentul unui complex
      void ipi();                  // metoda care incrementeaza partea imag.
      void dpi();                  //met. de decrem. a partii imag
      //operator + binar
      friend complex operator+(const complex&,const complex&);//complex+complex
      friend complex operator+(const double,const complex &);      //real+complex
      friend complex operator +(const int,const complex &);        //int+complex
      friend complex operator +(const complex&,const double);      // complex+double
      complex operator - (const complex &) const; //operator - binar: complex-complex
      //operator inmultire binar: complex*complex
      friend complex operator * (const complex &,const complex &);
      complex operator *(const complex ) const;
       /*TEMA:
            friend complex operator / (const complex &,const complex &);
            complex operator / (const complex &);          */
      complex  & operator + () const;              //operator + unar; metodă constantă
      complex  operator - () const;                //operator - unar
      complex &operator=(const complex &);
      complex & operator += (const complex &z);
      complex operator += (const double);
      complex operator -= (const complex&);
      complex & operator /= (const complex &z);
       /* TEMA
            complex operator *= (const complex&);
            complex operator /= (const complex&);*/
      complex & operator ++ ();
```

```
        complex operator ++ (int);      //forma postfixată
        complex operator--();           //decrementarea părţii reale a obiectului complex curent
        complex operator ! ();          //calcul. rădăcinii pătrate a obiectului complex curent
        int operator == (complex &z); //compară doi complecşi şi returnează 1 în caz de egalit.
        friend int operator == (complex &, complex &);   //return. 1 daca 2 compl egali
        int operator != (complex &);
        friend int operator != (complex &, complex &);
        friend ostream &operator<<(ostream &,const complex&); //operator afisare complex
        friend istream & operator >> (istream &,complex&);     //operator citire complex
};
```

```
// FISIERUL complex.cpp
#include "complex.h"
#include <stdlib.h>
#include <math.h>
inline complex::complex(double r,double i)
      {re=r;im=i;cout<<"Constructor implicit ptr complex("<<re<<","<<im<<")\n";}
complex::complex(const complex & z)
      {re=z.re;im=z.im;cout<<"Constructor copiere ptr complex(";
      cout<<re<<","<<im<<")\n";}
inline double complex::modul()
      { return sqrt(re*re+im*im); }
double complex::arg()
      { double a;
      if (re==0 && im==0) return (double)0;
      if (im==0)
            if (re>0) return 0.0;
            else return PI;
      if (re==0)
            if (im==0) return PI/2;
             else return (3*PI)/2;
      a=atan(im/re);
      if (re<0) return PI+a;
      if (im<0) return 2*PI+a;
      return a;
      }
inline void complex::ipi()
      { im++; }
inline void complex::dpi()
      { im--; }
complex operator +(const complex &a, const complex &b)
      {complex z; z.re=a.re+b.re;   z.im=a.im+b.im;   return z; }
complex operator +(const double d, const complex &a)
      {complex z;z.re=d+a.re;z.im=a.im;return z;}
complex operator +(const int d, const complex &a)
      {complex z;z.re=d+a.re;z.im=a.im;return z;}
complex operator +(const complex &a, const double d)
      {complex z;z.re=d+a.re;z.im=a.im;return z;}
complex complex::operator-(const complex &a) const
      {complex z;z.re=re+a.re;z.im=im+a.im;return z;}
complex operator *(const complex &x,const complex &y)
      {complex z;z.re=x.re*y.re-x.im*y.im;z.im=x.re*y.im+x.im*y.re;return z;}
complex complex::operator *(const complex x) const
      {complex z;z.re=re*x.re-im*x.im;z.im=re*x.im+im*x.re;return z;}
complex & complex::operator +() const
      {return *this;}
complex  complex::operator -() const
      {complex z;z.re=-re;z.im=-im;return z;}
complex & complex::operator=(const complex &a)
      {re=a.re;im=a.im;return *this;}                    // returnează obiectul curent
complex & complex::operator+=(const complex &x)
```

```
      {double re1=re*x.re-im*x.im;double im1=re*x.im+im*x.re;
      re=re1; im=im1;return *this;}
complex complex::operator+=(const double d)
      {re+=d; return *this;}
complex complex::operator-=(const complex &x)
      {re-=x.re;im-=x.im; return *this;}
complex &complex::operator /= (const complex &z)
      {double numitor=z.re*z.re+z.im*z.im;
      double re1=(double)(re*z.re+im*z.im)/numitor;
      double im1=(double)(im*z.re-re*z.im)/numitor;
      re=re1; im=im1;return *this;}
complex & complex::operator++()                         //forma prefixata
      {cout<<"F. prefixata!\n";re++; return *this;}
complex complex::operator ++ (int)                      //forma postfixata
      { cout<<"F. postfixata!\n";complex z=*this; re++; return z;}
complex complex::operator--()
      {re--; return *this;}
complex complex::operator ! ()
      { complex w; double d,e;
      if ((d=modul())==0)       return w;
      e=arg();d=sqrt(d); e=e/2;w.re=d*cos(e);w.im=d*sin(e);return w;}
int complex::operator==(complex &x)
      {return re==x.re && im==x.im;}
int operator==(complex &x, complex &y)
      {return (x.re==y.re && x.im==y.im);}
int complex::operator!=(complex &x)
      {return !(re==x.re && im==x.im);}
int operator!=(complex &x, complex &y)
      {return !(x.re==y.re && x.im==y.im);}
ostream &operator<<(ostream &ecran, const complex &z)
      {ecran<<"("<<z.re;
      if (z.im>=0)    ecran<<'+';ecran<<z.im<<"*i)";
      return ecran;}
istream &operator>>(istream &tastatura, complex &z)
      {tastatura>>z.re>>z.im;return tastatura;}

//FISIERUL de test
#include "complex.cpp"
#include <conio.h>
#include <stdio.h>

complex a(2, -6), b;
void main()
{ cout<<"Intrare in main\n";
complex x(3,7), y(-1, 28), z; cout<<"b="<<b<<'\n';cout<<"x="<<x<<'\n';
cout<<"y="<<y<<'\n';cout<<"z="<<z<<'\n'; cout<<"a="<<a<<'\n';
cout<<"a++="<<a++<<'\n';cout<<"++a="<<++a<<'\n';
printf("Pentru continuare introdu un car!\n");  getch();
      { complex w; cout<<"Introduceti w:\n";
      cin>>w; cout<<"w citit este: "<<w<<'\n';
      w.ipi(); cout<<"Dupa increm. p. imag:"<<w<<'\n';
      w.dpi(); cout<<"Dupa decrem. p. imag:"<<w<<'\n';
      cout<<"Modulul lui w este:"<<w.arg()<<'\n';
      cout<<"Argumentul lui w este:"<<w.arg()<<'\n';
      printf("Pentru continuare introdu un car!\n");  char rasp;
      cout<<"Se iese din blocul interior!\n";
      }
printf("Pentru continuare introdu un car!\n");getch();
cout<<"a="<<a<<'\n';
++a; cout<<"Dupa increm. p. reale: "<<a<<'\n';
--a; cout<<"Dupa decrem. p. reale: "<<a<<'\n';
a=x;cout<<"x="<<x<<'\n';cout<<"Dupa atribuire: a="<<a<<'\n';getch();
```

```
a=x++;cout <<"a = "<<a<<'\n';complex k1=a;cout<<"k="<<k1<<'\n';
a=++x;cout <<"a = "<<a<<'\n';complex k=a;cout<<"k="<<k<<'\n';getch();
k=-a;cout<<"- unar aplicat lui k:"<<k<<'\n';cout<<"-k="<<-k<<'\n';
k=x+y;cout<<x<<" + "<<y<<" = "<<k<<'\n';getch();
k=4+x;cout<<" 4 + "<< x <<" ="<<4+x<<'\n';k=a*x;
cout<<a<<" * "<<x<<" = "<<k<<'\n';
}
```

```
//FISIERUL tst_compl1.cpp
#include <iostream.h>
#include "complex.cpp"
complex a1(2,-6),b1;
void main()
{
cout<<"Intrare in main!!\n";
complex a(1,3), b(-1.3, 2.4),c;cout<<"a="<<a<<'\n';cout<<"b="<<b<<'\n';
c=(a+=b);cout<<"c="<<c<<'\n';complex x(3,7),y(-1,28),z;
cout<<"x="<<x<<" y="<<y<<" z="<<z<<'\n';c=a*b;
cout<<"a*b="<<c<<'\n';complex d;cout<<"Astept d ";cin>>d;
cout<<"d="<<d<<'\n';
      cout<<"\nIntrare in blocul interior!!\n";
      {complex w; cout<<"w="<<w<<'\n';    w=a;
      cout<<"Dupa atribuirea w=a:  w="<<w<<'\n';
      w=-a; cout<<"Dupa atribuirea w=-a: w="<<w<<'\n';w=+a;
      cout<<"Dupa atribuirea w=-a: w="<<w<<'\n';      w+=a;
      cout<<"Dupa atribuirea w+=a: w="<<w<<'\n';      b=x+2;
      cout<<"Dupa atribuirea b=x+2: w="<<w<<'\n';     z=2+x;
      cout<<"Dupa atribuirea b=2+x: w="<<w<<'\n';
      cout<<"Iesire din blocul interior\n";
}
cout<<"a+4="<<(a+4)<<'\n';cout<<"4+a="<<(4+a)<<'\n';cout<<"Iesire din main!!\n";
}
```

Exerciţiu: În exerciţiul următor se implementează **clasa fracţie**. Ea are ca *date membre private* `nrt` şi `nmt` (numărătorul şi numitorul). Tot ca metodă privată este definită metoda `simplifică()`, folosită pentru evitarea unor calcule cu numere mari.

Ca metode publice sunt definite: un *constructor*, un *destructor*, funcţia `numărător` care returnează valoarea datei membre *nrt*, funcţia `numitor` care returnează valoarea datei membre *nmt*, funcţia `valoare` care returnează valoarea reală obţinută prin împărţirea numărătorului la numitor şi funcţia `afişare`.

Se supraîncarcă operatorii +, -, *, / prin *funcţii prietene* ale clasei fracţie. Semnificaţia dată este cea de a realiza operaţii de adunare, scădere, înmulţire şi împărţire a obiectelor din clasa fracţie.

Se supraîncarcă operatorii +=, -=, *=, /= prin *funcţii membre* ale clasei fracţie.

Funcţia `cmmdc` (care implementează algoritmul lui Euclid pentru aflarea celui mai mare divizor comun a două numere) nu este nici funcţie membră a clasei fracţie, nici funcţie prietenă. Ea este apelată de funcţia simplifică. Funcţia `simplifică` este utilizată pentru a evita obţinerea unor valori mari pentru datele membre *nrt* şi *nmt*.

```
#include <iostream.h>
class fracţie
{     int nrt,nmt;                              // numărător,numitor
      void simplifică();                        //metodă de simplificare a fracţiei
public:
      fracţie(int nrti=0, int nmti=1);          // constructor iniţializare
      ~fracţie()  {cout<<"DESTRUCTOR!!!\n";};   //destructor
      int  numărător()  {return nrt;}
      int numitor()     {return nmt;}
      double valoare()  {return (double)nrt/(double)nmt;}
      void afişare();
      friend fracţie operator+(const fracţie&, const fracţie&);
      friend fracţie operator-(const fracţie&, const fracţie&);
```

```
      friend fracţie operator*(fracţie&, fracţie&);
      friend fracţie operator/(fracţie&, fracţie&);
      fracţie& operator =(const fracţie&);
      fracţie& operator +=(const fracţie&);
      fracţie& operator -=(const fracţie&);
      fracţie& operator *=(const fracţie&);
      fracţie& operator /=(const fracţie&);
};
int cmmdc(int x,int y)              //calculează şi returnează cmmdc pentru x, y
      {int z; if (x==0 || y==1) return 1;
      if (x<0)    x=-x;
      if (y<0)    y=-y;
      while (x!=0){
            if (y>x)    {z=x;x=y;y=z;}
            x%=y;
      }return y;}

void fracţie::simplifică()
      {int cd;
      if (nmt<0)  {nrt=-nrt;nmt=-nmt;}
      if (nmt>1){ cd=cmmdc(nrt,nmt);if (cd>1)   {nrt/=cd; nmt/=cd;}     }
      }
fracţie::fracţie(int nri, int nmi)
      {nrt=nri; nmt=nmi; simplifică(); cout<<"Constructor!\n";}
fracţie operator +(const fracţie &f1, const fracţie &f2)
      {int dc; fracţie f;
      dc=cmmdc(f1.nmt,f2.nmt);f.nmt=(f1.nmt/dc)*f2.nmt;
      f.nrt=f1.nrt*(f2.nmt/dc)+f2.nrt*(f1.nmt/dc);f.simplifică();return f;}
fracţie operator -(const fracţie &f1, const fracţie &f2)
      {int dc; fracţie f;dc=cmmdc(f1.nmt,f2.nmt);f.nmt=(f1.nmt/dc)*f2.nmt;
      f.nrt=f1.nrt*(f2.nmt/dc) - f2.nrt*(f1.nmt/dc);f.simplifica();return f;}
fractie operator * (fractie &f1, fractie &f2)
      { int dc;fractie f;dc=cmmdc(f1.nrt,f2.nmt);
      if (dc>1)   {f1.nrt/=dc;f2.nmt/=dc;}
      dc=cmmdc(f2.nrt,f1.nmt);
      if (dc>1)   {f2.nrt/=dc;f1.nmt/=dc;}
      f.nrt=f1.nrt*f2.nrt; f.nmt=f1.nmt*f2.nmt;return f; }
fractie operator / (fractie & f1, fractie & f2)
      { int dc;fractie f;dc=cmmdc(f1.nrt,f2.nrt);
      if (dc>1)   {f1.nrt/=dc;f2.nrt/=dc;}
      dc=cmmdc(f2.nmt,f1.nmt);if (dc>1)   {f2.nmt/=dc;f1.nmt/=dc;}
      f.nrt=f1.nrt*f2.nmt; f.nmt=f1.nmt*f2.nrt;return f;}
void fractie::afisare()
      {cout<<"f="<<nrt<<'/'<<nmt<<'\n';}
fractie& fractie::operator=(const fractie &f1)
      { nmt=f1.nmt;nrt=f1.nrt; return *this;}
fractie& fractie::operator+=(const fractie &f1)
      { int dc=cmmdc(nmt,f1.nmt);
      nmt=(nmt/dc)*f1.nmt;nrt=nrt*(f1.nmt/dc)+f1.nrt*(nmt/dc);
      simplifica();return *this;}
fractie& fractie::operator-=(const fractie &f1)
      { int dc=cmmdc(nmt,f1.nmt);
      nmt=(nmt/dc)*f1.nmt;nrt=nrt*(f1.nmt/dc)-f1.nrt*(nmt/dc);simplifica();
      return *this;}
fractie& fractie::operator *=(const fractie &f1)
      { int dc;dc=cmmdc(nrt,f1.nmt);if (dc>1)   {nrt/=dc;f1.nmt/=dc;}
      dc=cmmdc(f1.nrt,nmt);if (dc>1)      {f1.nrt/=dc;nmt/=dc;}
      nrt=nrt*f1.nrt;nmt=nmt*f1.nmt;simplifica();return *this;}
fractie& operator /=(const fractie &f1)
      { int dc;dc=cmmdc(nrt,f1.nrt);
      if (dc>1)   {nrt/=dc;f1.nrt/=dc;}
      dc=cmmdc(f1.nmt,nmt);   if (dc>1)   {f1.nmt/=dc;nmt/=dc;}
```

```
	nrt=nrt*f1.nmt; nmt=nmt*f1.nrt;return *this;}

void main()
{	double n1, n2;fractie f(4,5);f.afisare();
	fractie f1(5,4);fractie sum=f+f1;sum.afisare();
	cout<<"NOUA AFISARE:\n"<<sum;cout<<"Numarator:"; cin>>n1;
	cout<<"Numitor:"; cin>>n2;fractie f4(n1, n2); f4.afisare();
	f4+=f1;f4.afisare();
	f4=f=f2; f4.afisare();
	fractie f2; f2.afisare();
}
```

Observaţii:

- Nu a fost necesară definirea unui constructor de copiere şi nici supraîncărcarea operatorului de atribuire, deoarece clasa fracţie nu contine pointeri către date alocate dinamic. În ambele situaţii se face o ***copiere bit cu bit***, conform procedurii standard de copiere a structurilor din limbajul C. Într-o atribuire cum ar fi: `f4=f;` (unde `f4`, `f` de tip fractie), se realizează copierea bit cu bit a fracţiei `f` în fracţia `f4`.
- Operatorii binari simpli `+`, `-`, `*`, `/` au fost supraîncărcaţi prin funcţii prietene, pentru a putea fi folosiţi în expresii de tipul `n+f`, în care `n` este operand de *tip int* şi `f` este operand de *tip fracţie*. Dacă aceşti operatori ar fi fost supraîncărcaţi prin metode ale clasei fracţie, ar fi putut fi utilizaţi doar în expresiile în care *operandul stâng* era de *tip fracţie*.
- Operatorii binari compuşi `+=`, `-=`, `*=`, `/=` au fost supraîncărcaţi prin funcţii membre, deoarece *operandul stâng* este întotdeauna de *tip fracţie.*
- În programul de test fracţiile `f` şi `f1` au fost iniţializate cu valorile 4 si 5, respectiv 5 si 4. Fracţia `f2` a fost iniţializată cu 0 şi 1, datorită parametrilor impliciţi ai constructorului clasei. Acest lucru se observă în urma apelului funcţiei `afisare` pentru obiectele f, f1, f2.
- În cazul unei atribuiri de forma `f=n;` , unde f este fracţie si n este de tip int, înainte de atribuirea propriu-zisă are loc o conversie a lui `n` în fracţie. Întregul n este convertit automat în fracţie `(n,1)`. Acelaşi efect s-ar fi obţinut în urma unei conversii explicite: `(fractie) n`.
- Pentru un obiect din clasa fracţie, constructorul poate fi apelat explicit: `f=fractie(4,4);` (în loc de `fractie f(4,5);`).

Pentru a evita lucrul cu fişiere care au sute de linii de cod, se folosesc două abordări:

a) Se crează fişierul sursă numit **`fractie.h`** (header al utilizatorului) care conţine declararea clasei fracţie.
Se crează fişierul **`fractie.cpp`** în care se implementează metodele clasei fracţie. În acest fişier se include header-ul `"fractie.h"`.
Se crează un al treilea fişier care testează tipul de date fracţie, în care se include fişierului `"fractie.cpp"`. Se compilează, se linkeditează şi se lansează în execuţie fişierul executabil obţinut.

b) Se construieşte un ***proiect***. De exemplu, dacă se lucrează sub un mediu integrat, cum ar fi BorlandC, se crează cele trei fişiere (`fractie.h` care conţine declararea clasei, `fractie.cpp` care implementează metodele clasei şi fişierul de test (`test_fractie.cpp`)). Fişierul header `fractie.h` va fi inclus atât în `fractie.cpp`, cât şi în `test_fractie.cpp`. Din meniul **`"Project"`** se selectează **`"Open Project"`**, apoi comanda **`"Add item…"`**, acre permite adăugarea fişierelor **`fractie.cpp`** si **`test_fractie.cpp`**.

Pentru a evita includerea aceluiaşi fişier header de mai multe ori, se folosesc ***directivele de compilare condiţionată***.

Exemplu:

```
	#ifndef  _fractie_h
	#include "fractie.h"
	#define _fractie_h
	#endif
```

Exerciţiu: Se defineşte tipul `şir`, cu ***date membre (private):***

✓ `int lung`
Lungimea propriu-zisă (nr. de caractere din şir), fără terminator

✓ `char *sirul`
    Adresa început şir (şirul-pointer către început şir)

Metode:

✓ `sir();`
    Constructor vid

✓ `sir (char *);`
    Constructor de inițializare care primeşte ca parametru un pointer către un şir de caractere (alocare dinamică).

✓ `sir(const sir&);`
    Constructor de copiere: primeşte ca argument o referință către un obiect din clasa şir şi realizează copierea.

✓ `~sir();`
    Destructor care eliberează memoria alocată dinamic.

✓ `int lungime();`
    Returnează valoarea datei membre lung (nr. de carctere din şir).

✓ `const char *continut();`
    Returnează conținutul unui obiect de tip şir.

✓ `sir &operator=(const sir&);`
    Supraîncărcarea operatorului de atribuire printr-o funcție membră. A fost necesară supraîncarcarea operatorului de atribuire datorită faptului că această clasă conține ca date membre, pointeri.

✓ `sir &operator+=(const sir&);`
    Operator supraîncarcat prin funcție membră care realizează concatenarea obiectului curent (operandul implicit, de tip şir) cu obiectul de tip şir primit ca parametru.

✓ `friend sir operator+(const sir& s1, const sir& s2);`
    Supraîncarcă operatorul de adunare printr-o funcție prietenă. Acesta concatenează obiectele de tip şir primite ca parametri. Returnează şirul obținut în urma concatenării.

✓ `friend ostream &operator<<(ostream &, const sir&);`
    Supraîncarcă operatorul insertor printr-o funcție prietenă a clasei şir.

✓ `friend istream &operator>>(istream &, sir&);`
    Supraîncarcă operatorul extractor printr-o funcție prietenă a clasei şir.

```
// FISIERUL sir.h
#include <iostream.h>
class sir
{       int lung;             //lungimea propriu-zisa, fara terminator
        char *sirul;          //adresa inceput sir (sirul-pointer catre inceput sir)
public:
        sir();                //constructor vid: construieste un sir vid
        sir (char *);         //constructor initializare primeste ca arg. un sir standard
        sir(const sir&);
// constructor copiere: primeste ca arg. o referinta catre un obiect din cls. sir si trebuie sa faca o copiere
        ~sir();                     //destructor
        int lungime()               //metoda care return. nr de car din componenta sirului
        const char *continut()  //metoda inline-returneaza continutul sirului curent
        sir &operator=(const sir&);        /*supraincarcare operator atribuire(necesar dat. faptului
ca, in cls sir, exista un pointer catre data membra "sirul" supraincarcat prin f-ctie membra, ptr ca intotdeauna
membrul stang este un obiect din clasa sir */
        sir &operator+=(const sir&);   //concateneaza argumentul primit la sirul curent
        friend sir operator+(const sir&, const sir&);       //concateneaza argumentele
        friend ostream &operator<<(ostream &,const sir&);//supraincarcare operator insertor
        friend istream &operator>>(istream &,sir&);//supraincarcare operator extractor
};
// FISIERUL sir.cpp
//conține definițiile funcțiilor din clasa şir.
#ifndef _sir_h
#include "sir.h"
```

```
#define _sir_h
#endif

#ifndef _stdio_h
#include "stdio.h"
#define _stdio_h
#endif

#ifndef _string_h
#include "string.h"
#define _string_h
#endif

#ifndef _iostream_h
#include "iostream.h"
#define _iostream_h
#endif

sir::sir()
      {sirul=0;lung=0; cout<<"Constructor vid\n";}
sir::sir(char *s)
      {cout<<"Apel constructor\n";
       lung=strlen(s);
        if (lung>0){sirul=new char[lung+1];
              if (sirul!=0) strcpy(sirul,s);
              else    lung=0;
        }
        else sirul=0;
      }
sir::sir(const sir &s)
      {cout<<"Constructor copiere\n";
      if (s.lung>0){
              sirul=new char[s.lung+1];
              if (sirul!=0){
                   strcpy(sirul,s.sirul);
                   lung=s.lung;
              }
              else lung=0;
      }
      else {lung=0;sirul=0;}
      }
sir::~sir()
      {cout<<"Destructor\n";if (sirul!=0) delete sirul;}
int sir::lungime()
      {return lung;}
const char *sir::continut()
      {return sirul;}
sir &sir::operator=(const sir&s)
      {cout<<"Operator de atribuire\n";
      if (sirul!=0)  delete sirul;
        if (s.lung>0){
             sirul=new char[s.lung+1];
             if (sirul!=0){
                  strcpy(sirul,s.sirul);
                  lung=s.lung;
              }
              else lung=0;
      }
      else {sirul=0; lung=0;}
      return *this;
      }
sir & sir::operator+=(const sir &s)
      { if (s.lung>0){
```

```
              char *ps; int lung1; lung1=lung+s.lung;  ps=new char[lung1+1];
              if (ps!=0){
                   strcpy(ps,sirul); strcat(ps,s.sirul);
                   delete sirul; sirul=ps;lung=lung1;
                  }
       }
      return *this;
      }
sir operator+(const sir &s1, const sir &s2)
      { sir s;
       s.lung=s1.lung+s2.lung; s.sirul=new char[s.lung+1];
       if (s.sirul!=0){
            if (s1.lung>0)
                   strcpy(s.sirul,s1.sirul);
            else strcpy(s.sirul,"");
       if (s2.lung>0)
             strcat(s.sirul,s2.sirul);
       }
       else {s.lung=0; s.sirul=0;}
       return s;
      }
ostream &operator<<(ostream &ies, const sir &s)
      { if (s.lung>0) ies<<s.sirul;  return ies; }
istream &operator>>(istream &intr, sir &s)
      { char s1[100];printf("Astept sir:"); scanf("%s", s1);s=s1;return intr;}
```

```
// FISIERUL test_sir.cpp
// Program de test pentru clasa şir
#include "sir.cpp"
#include <conio.h>
void main( )
      { const char *p;clrscr();cout<<"Declaratii obiecte din cls sir\n";
      sir s1("SIR INITIALIZAT!"), s2(s1), s3, s4;getch();cout<<"\nAfisari\n";
      cout<<"s1="<<s1<<'\n';    cout<<"s2="<<s2<<'\n';   cout<<"s3="<<s3<<'\n';
      cout<<"s4="<<s4<<'\n';s3=s2;cout<<"\nAtribuire: s3=s2 :s3="<<s3<<'\n';
      getch();s4="Proba de atribuire";cout<<"s4="<<s4<<'\n';
      cout<<"\nConcatenare s1 (ob. curent) cu s4.\n";
      s1+=s4;cout<<"s1="<<s1<<'\n';
      cout<<"\nConcatenare s1 cu s4, rezultat in s3\n";s3=s1+s4;
      cout<<"s3="<<s3<<'\n';getch();sir q;cout<<"Astept car. din sirul q:";
      cin>>q;cout<<"Nr car. introduse in sir:"<<q.lungime()<<'\n';
      cout<<"Continutul sirului:"<<q.continut()<<'\n';getch(); }
```

Aşa cum se observă din exerciţiul prezentat, în corpul constructorului se alocă memorie dinamic (cu operatorul `new`). Destructorul eliberează memoria alocată dinamic (cu operatorul `delete`).

11.7. SUPRAÎNCĂRCAREA OPERATORULUI DE INDEXARE [ ]

Operatorul de indexare este un operator binar şi are forma generală: `nume[expresie]`. Să considerăm clasa vector, definită astfel:

```
      class vector{
      private:
            int nrcomp;                //nr. componente
            double *tabcomp;           //tabloul componentelor; adresa de început
      public:
            double &operator[](int);
      }
```

Pentru tipul abstract `vector`, operatorul de indexare poate fi supradefinit, astfel încât să permită accesarea elementului de indice n. În acest caz, operatorul de indexare se poate supradefini printr-o funcţie membră a

clasei (deoarece operandul stâng este de tipul clasei), şi poate fi folosit sub forma: **`v[n]`** (unde v este obiect al clasei vector; n-expresie întreagă). Expresia **`v[n]`** este echivalentă cu **`v.operator[](n)`** (apelul explicit al funcţiei **`operator []`**). Transferul parametrului către funcţia care supraîncarcă operatorul se poate face prin *valoare* sau prin *referinţă*. În mod obligatoriu, funcţia trebuie să returneze ***referinţa către elementul*** aflat pe poziţia n (pentru a permite eventualele modificări ale elementului, deoarece `vector[n]` este **`lvalue`**).

Pentru un tip abstract, prototipul funcţiei care supradefineşte operatorul de indexare este (`const` protejează argumentul la modificările accidentale):

```
tip_element  &  operator [ ] (const int);
```

În cazul în care operatorul se supraîncarcă printr-o funcţie prietenă, prototipul funcţiei este:

```
tip_elem & operator (tip, int);
```

Exerciţiu: Să definim clasa `vector` are ca ***date membre (private)*** (figura 11.4.)***:***

- ✓ `int nrcomp;` - numărul elementelor vectorului
- ✓ `int err;` - indice de eroare
- ✓ `double *tabcomp;` - adresa de început a tabloului componentelor

Metode:

- ✓ `vector(int nrc=0);`
  Constructor iniţializare cu parametru implicit: număr elemente 0. Crează dinamic un vector de elemente reale (double) şi iniţializează elementele cu valoarea 0.
- ✓ `vector(const vector&);`
  Constructor de copiere: Pe baza vectorului primit ca argument, crează un nou vector (de aceeaşi dimensiune, cu aceleaşi valori ale componentelor).
- ✓ `virtual ~vector();`
  Destructor: eliberează memoria alocată dinamic la crearea unui obiect din clasa vector.
- ✓ `int dimens() const;`
  Returnează dimensiunea (numărul elementelor) pentru obiectul curent.
- ✓ `double &operator[](int);`
  Supraîncarca **operatorul de indexare `[]`** prin funcţie membră (metodă). Returnează referinţa către elementul cu numărul de ordine indicat ca argument.
- ✓ `vector &operator=(const vector&);`
  Supraîncarcarea operatorului de atribuire printr-o funcţie membră. A fost necesară supraîncarcarea operatorului de atribuire datorită faptului că această clasă conţine pointeri către datele membre.
- ✓ `int nrerori() const;`
  Metodă constantă (nu poate modifica obiectul curent) care returnează valoarea datei membre err;
- ✓ `void anulari();`
  Metoda care anulează indicele de eroare.
- ✓ `int comparare(const vector&) const;`
  Metoda constantă care compară obiectul curent (operand stâng, argument implicit) cu obiectul primit ca parametru (tot vector). Returnează o valoare întreagă, care este: 2 dacă vectorii au număr de componente diferit; 0 dacă obiectul curent are 0 elemente sau dacă vectorii comparaţi au aceleaşi elemente; 1 dacă vectorii au cel puţin un element diferit. Metoda nu modifică operandul stâng. Mod de apelare: `a.comparare(b);` (unde a, b vectori).
- ✓ `friend int prodscal(const vector& v1, const vector& v2, double& p);`
  Funcţie prietenă a clasei vector care calculează şi returnează valoarea produsului scalar pentru vectorii v1 şi v2, transmişi ca argumente:
  $$p= \sum_{k=0}^{nrcomp-1} v1[k]*v2[k]$$
- ✓ `friend int suma(const vector& v1, const vector& v2, vector& v);`
  Funcţie prietenă care calculează vectorul sumă v:
  $$v[k] = v1[k] + v2[k]$$
  Returnează o valoare întreagă: 1 dacă numărul de elemente din v1 este diferit de numărul elementelor din v2; 0 dacă v2 are 0 elemente, sau dacă s-a calculat suma; 3 dacă v are 0 elemente

✓ `friend int diferenta(const vector& v1, const vector& v2, vector& v);`

Funcție prietenă care calculează vectorul diferență v:

$$v[k] = v1[k] - v2[k]$$

Returnează o valoare întreagă: 1 dacă numărul de elemente din v1 este diferit de de numărul elementelor din v2; 0 dacă v2 are 0 elemente, sau dacă s-a calculat diferența; 3 dacă v are 0 elemente

✓ `friend ostream &operator<<(ostream &ies, const vector&);`

Operator de afişare supraîncarcat prin funcție prietenă. Apelează metoda privată constantă `afişare`.

✓ `virtual void afisare(ostream &)const;`

Cuvântul ***virtual*** care apare în antetul funcției indică faptul că metoda este o funcție virtuală. Ea poate fi eventual redefinită (cu acelaşi prototip) şi în clasele derivate din clasa vector. În cazul unei redefiniri, funcția ar fi supusă *"legării dinamice"*, ceea ce înseamnă că selecția ei se va face abia în momentul execuției.

✓ `double operator *(const vector& v1) const;`

Operator de înmulțire supraîncărcat prin metodă constantă care returnează o valoare reală reprezentând produsul scalar dintre obiectul curent şi cel primit ca argument. În funcție nu se crează o copie a lui v1, se lucrează cu v1 din programul apelant (parametru transmis prin referință).

✓ `vector operator+(const vector&) const;`

Operator de adunare supraîncărcat prin metodă constantă care returnează un vector (întoarce o copie care poate fi utilizată în programul apelant) reprezentând suma dintre obiectul curent şi cel primit ca argument.

✓ `vector operator-(const vector&) const;`

Operator de scădere supraîncărcat prin metoda constantă care returnează un vector (întoarce o copie care poate fi utilizată în programul apelant) reprezentând diferența dintre obiectul curent şi cel primit ca argument.

✓ `vector &operator+=(const vector& b);`

Operator supraîncărcat prin metodă, deoarece întotdeauna operandul stâng este de tipul vector. Este folosit în expresii cum ar fi: `a+=b` (a şi b de tipul vector).

✓ `vector &operator-=(const vector&);`

Operatorul `-=` supraîncărcat prin metodă, deoarece întotdeauna operandul stâng este de tipul vector.

✓ `int sort(char='A');`

Metodă care testează argumentul primit. Dacă acesta este valid ('A' sau 'D') apelează **metoda quicksort**, pentru ordonarea crescătoare sau descrescătoare a alementelor unui vector.

✓ `void quicksort(int, int, char);`

Metoda este protejată şi realizează ordonarea crescătoare (argumentul 'A') sau descrescătoare (argumentul 'D').

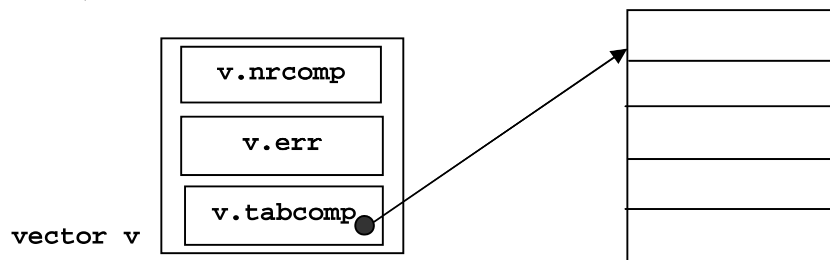


Figura 11.4. Obiectul v, de tip vector

Se prezintă varianta de lucru în care se crează un ***proiect***.

```
// FISIERUL vector.h
#ifndef _iostream_h
#include <iostream.h>
#define _iostream_h
#endif
class vector{
private:
        int nrcomp;                    //nr. componente
        int err;                       //indice eroare
```

```
        double *tabcomp;              //tabloul componentelor; adresa de început
public:
      vector(int nrc=0);             //constructor initializare pentru un vector vid
      vector(const vector&);         //constr. copiere
      ~vector();                     //destructor
      int dimens() const             //metoda constanta
                {return nrcomp;}
      double &operator[](int);       //supraincarc operator indexare
      vector &operator=(const vector&);     //supraincarcare operator de atribuire
      int nrerori() const
                  {return err;}
      void anulari()
                {err=0;}
      int comparare(const vector&) const;   //compara 2 vectori
      friend int prodscal(const vector&, const vector&, double&);
      friend int suma(const vector&, const vector&, vector&);
      friend int diferenta(const vector&, const vector&, vector&);
      friend ostream &operator<<(ostream &ies, const vector&);
      double operator *(const vector&) const;
      vector operator+(const vector&) const;
                  //intoarce copie care poate fi utilizata in progr. apelant
      vector operator-(const vector&) const;
      vector &operator+=(const vector&);  //a+=b
      vector &operator-=(const vector&);
      int sort(char='A');
private:
      void afisare(ostream &)const;
      void quicksort(int,int,char);
};


// FISIERUL vector.cpp
#ifndef _vector_h
#include "vector.h"
#define _vector_h
#endif

#ifndef _string_h
#include <string.h>
#define _string_h
#endif

#ifndef _ctype_h
#include <ctype.h>
#define _ctype_h
#endif

vector::vector(int nrc)
      {err=0;cout<<"Constructor vector\n";
      if (nrc>0){ nrcomp=nrc; tabcomp=new double[nrcomp+1];
            if (tabcomp==0)   nrcomp=0;
            else
            for (int i=0;i<=nrcomp;i++) tabcomp[i]=0;
            //initializare elemente vector cu 0
            }
       else{      nrcomp=0;   tabcomp=0; }
      }
vector::vector(const vector &v)
      { err=v.err;  cout<<"Constructor copiere!\n";
        if (v.nrcomp>0){nrcomp=v.nrcomp; tabcomp=new double[nrcomp+1];
              if (tabcomp==0){ nrcomp=0;  tabcomp=0;   err=1; }
```

```
              else{
                for(int k=0;k<=nrcomp;k++)   tabcomp[k]=v.tabcomp[k];}
         }
         else{  nrcomp=0;   tabcomp=0; }   }
vector::~vector()
      {cout<<"Destructor!\n"; if (tabcomp!=0)   delete tabcomp; }
double &vector::operator[](int i)
      { if (i<0 || i>=nrcomp){err++; return tabcomp[nrcomp];   }
         else        return tabcomp[i];  }

int vector::comparare(const vector&v) const
//w.comparare(v)
      { int k;
       if (nrcomp!=v.nrcomp) return 2;
       if (nrcomp==0)        return 0;
       for (k=0;k<nrcomp && tabcomp[k]==v.tabcomp[k];k++)
        if (k<nrcomp)      return 1;     //vectorii au cel putin un elem. diferit
      return 0;
      }
//v.afisare(cout)
void vector::afisare(ostream &ies) const
      { ies<<'[';
      for (int i=0;i<(nrcomp-1);i++)   ies<<tabcomp[i]<<", ";
      if (nrcomp>0)      ies<<tabcomp[nrcomp-1];
       ies<<']';
}
ostream &operator<<(ostream &ies, const vector &v)
      { v.afisare(ies);   return ies; }
vector &vector::operator=(const vector &v)
      { cout<<"Operator atribuire!\n";
        if (tabcomp!=0)        delete tabcomp;
        nrcomp=0;  err=0;  tabcomp=0;
        if (v.nrcomp>0){
              tabcomp=new double[v.nrcomp+1];
              if (tabcomp!=0){
                nrcomp=v.nrcomp;  err=v.err;
                for (int k=0;k<=nrcomp;k++)
                        tabcomp[k]=v.tabcomp[k];
              }
        }
        return *this;
      }
int prodscal(const vector &v1, const vector &v2, double &p)
//p=SUMA(v1[k]v2[k])
      { p=0;
        if (v1.nrcomp!=v2.nrcomp)     return 1;
        if (v1.nrcomp==0)             return 2;
        for (int k=0;k<v1.nrcomp;k++)        p+=v1.tabcomp[k]*v2.tabcomp[k];
          // sau: p+=v1[k]*v2[k] pentru ca s-a supraincarcat operatorul de  indexare
          //mod apel j=prodscal(w,v,p);
          //mod apel prodscal(w,v,p);
              return 0;
      }
int suma(const vector &v1, const vector &v2, vector &v)
      { v.nrcomp=v.err=0;
        if (v.tabcomp!=0)        delete v.tabcomp;
        v.tabcomp=0;
        if (v1.nrcomp!=v2.nrcomp)     return 1;
        if (v2.nrcomp==0)             return 0;
        v.tabcomp=new double[v1.nrcomp+1];
        if (v.tabcomp==0)        return 3;
        v.nrcomp=v1.nrcomp;
```

```
          for (int k=0;k<=v1.nrcomp;k++)v.tabcomp[k]=v1.tabcomp[k]+v2.tabcomp[k];
        return 0;
        }

int diferenta(const vector &v1, const vector &v2, vector &v)
        { v.nrcomp=v.err=0;
          if (v.tabcomp!=0)        delete v.tabcomp;
          v.tabcomp=0;
          if (v1.nrcomp!=v2.nrcomp)        return 1;
          if (v2.nrcomp==0)     return 0;
          v.tabcomp=new double[v1.nrcomp+1];
          if (v.tabcomp==0)     return 3;
          v.nrcomp=v1.nrcomp;
          for (int k=0;k<=v1.nrcomp;k++) v.tabcomp[k]=v1.tabcomp[k]-v2.tabcomp[k];
        return 0;
        }
double vector::operator*(const vector &b) const
//z=a*b; a-operandul din stânga
        { double z=0.0;
        if (nrcomp!=b.nrcomp){
              // err++;
           cout<<"Nr. componente diferit! Nu se poate face produs scalar!\n";
          return 4;}
        else if (nrcomp>0)
          for (int k=0;k<nrcomp;k++)  z+=tabcomp[k]*b.tabcomp[k];
        return z;
        }
vector vector::operator+(const vector &b) const
//c=a+b
        {if (nrcomp!=b.nrcomp)  {vector c;c.err=1;return c;}
         if (nrcomp==0)         {vector c;return c;}
         vector c(nrcomp);
         for (int k=0;k<nrcomp;k++) c.tabcomp[k]=tabcomp[k]+b.tabcomp[k];
         return c;
        }
vector vector::operator-(const vector &b) const
//c=a-b
        {if (nrcomp!=b.nrcomp){vector c;c.err=1;return c;}
         if (nrcomp==0)       {vector c;return c;}
         vector c(nrcomp);
         for (int k=0;k<nrcomp;k++) c.tabcomp[k]=tabcomp[k]-b.tabcomp[k];
         return c;
        }
vector &vector::operator+=(const vector &b)
        { if (nrcomp!=b.nrcomp) err++;
         else
                if (nrcomp>0)
                        for (int k=0;k<nrcomp;k++) tabcomp[k]+=b.tabcomp[k];
        return *this;
        }
vector &vector::operator-=(const vector &b)
        { if (nrcomp!=b.nrcomp) err++;
               else
                if (nrcomp>0)
                        for (int k=0;k<nrcomp;k++) tabcomp[k]-=b.tabcomp[k];
         return *this;
        }
void vector::quicksort(int i1,int i2,char modsort)
        {int i,j; double a,y;i=i1;j=i2;a=tabcomp[(i1+i2)/2];
        do{
                switch (modsort)
                { case 'A': while (tabcomp[i]<a) i++;
```

```
                    while (tabcomp[j]>a) j--;
                    break;
                case 'D': while (tabcomp[i]>a) i++;
                    while (tabcomp[j]<a) j--;
               }
          if (i<=j)
             {y=tabcomp[i];tabcomp[i]=tabcomp[j];tabcomp[j]=y;i++;j--;}
      } while (i<=j);
      if (i1<j) quicksort(i1,j,modsort);
      if (i<i2) quicksort(i,i2,modsort);
      }
int vector::sort(char modsort)
      { modsort=toupper(modsort);
      if (modsort!='A' && modsort!='D')   return 1;
      if (nrcomp==0)                      return 2;
      if (nrcomp==1) return 0;
      quicksort(0,nrcomp-1,modsort);
      return 0;
      }

//FISIERUL test_vec.cpp
#ifndef _vector_h
#include "vector.h"
#define _vector_h
#endif

void main()
{int ier;
      vector v(10),w(10),z;   double t[10]={4,2,6,1,4,9,-3,2,5,2};
      int k,i2,i3; double p,p1;cout<<"v="<<v<<'\n';
      cout<<"w="<<w<<'\n';cout<<"z="<<z<<'\n';cout<<"v[3]="<<v[3]<<'\n';
      for (k=0;k<10;k++)
            { v[k]=t[k];cout<<"intr. w["<<k<<"]=";cin>>w[k];}
      cout<<"Vect. v neordonat:\n"<<v<<'\n';
      cout<<"Nr. erori:"<<v.nrerori()<<'\n';
      ier=v.sort('A');cout<<"Vect. v ordonat crescator:"<<v<<'\n';
      cout<<"ier="<<ier<<'\n';
      cout<<"Vect. w neordonat:\n"<<w<<'\n';ier=w.sort('D');
      cout<<"Vect. w ordonat descrescator:"<<w<<'\n';cout<<"ier="<<ier<<'\n';
      vector cc(v);cout<<"cc="<<cc<<'\n';
      int i1=prodscal(v,w,p); cout<<"Produsul scalar="<<p<<'\n';
      cout<<"Produs scalar="<<(v*w)<<'\n';
      i2=suma(v,w,z);cout<<"Vector suma:"<<z<<'\n';
      cout<<"Vector suma:"<<(v+w)<<'\n';
      i3=diferenta(v,w,z);cout<<"Vector diferenta:"<<z<<'\n';
      cout<<"Vector diferenta:"<<(v-w)<<'\n';
      cout<<"Inainte de atribuire:\n";cout<<"z="<<z<<'\n';cout<<"v="<<v<<'\n';
      z=v;cout<<"Dupa atribuire:\n";cout<<"z="<<z<<'\n';cout<<"v="<<v<<'\n';
      z+=v;cout<<"z="<<z<<'\n';cout<<"v="<<v<<'\n';
      int test=z.comparare(z);
      if (test==1)            cout<<"Siruri egale!\n";
      else                    cout<<"Siruri diferite!\n";
      test=z.comparare(v);
      if (test==1)            cout<<"Siruri egale!\n";
      else                    cout<<"Siruri diferite!\n";
}
```

11.7. SUPRAÎNCĂRCAREA OPERATORILOR *NEW* ŞI *DELETE*

Avantajul alocării dinamice a memoriei şi a eliberării acesteia cu ajutorul operatorilor **`new`** şi **`delete`**, faţă de utilizarea funcţiilor `malloc`, `calloc` sau `free` (vezi capitolul 6.9.), constă în faptul că operatorii alocă

(eliberează) memorie pentru obiecte, date de tip abstract. Acest lucru este posibil deoarece aceşti operatori au o ***supraîncărcare globală, standard***.

În cazul în care supraîncărcarea standard este insuficientă, utilizatorul poate supraîncărca operatorii prin **metode (implicit!) statice**.
Pentru operatorul new, funcția care supraîncarcă ooperatorul `new` are prototipul:

```
void  * nume_clasa::operator new (size_t lungime);
```

Funcția returnează un pointer generic a cărui valoare este adresa de început a zonei de memorie alocate dinamic. Tipul `size_t` este definit în `stdlib.h` (vezi capitolul 6.9.). La aplicarea operatorului, nu se indică nici o valoare pentru parametrul `lungime` (mărimea zonei de memorie necesare obiectului pentru care se alocă dinamic memorie), deoarece compilatorul o determină, automat.
Modul de utilizare pentru operatorul **`new`**:

```
nume_clasa *p = new nume_clasa;
```

Sau:

```
nume_clasa *p = new nume_clasa(p1, p2, p3);
```

Aplicarea operatorului `new` supradefinit de utilizator determină, automat, apelul constructorului corespunzător clasei, sau al constructorului implicit. În a doua formă, la alocarea dinamică a memoriei apar şi parametrii constructorului (`p1`, `p2`, `p3`).

Operatorul **`delete`** se supradefineşte printr-o funcție cu prototipul:

```
void nume_clasa::operator delete (void *);
```

La aplicarea operatorului `delete` se apelează, automat, destructorul clasei.
<u>Exemple:</u>

```
class c1{
      double n1, n2;
public:
      c1(){n1=0; n2=0;}
};
//. . . .
void main( )
{ c1 *pc1=new c1;          //se alocă memorie pentru păstrarea unui obiect de tip c1
c1 *pc2=new c1(7, 5);      //odată cu alocarea dinamică, se realizează şi inițializarea
}
```

Operatorii new, delete permit alocarea dinamică şi pentru tablouri. În aceste situații, se utilizează întotdeauna operatorii supraîncărcați global predefiniți şi se apelează constructorul fără parametri (dacă acesta există).
<u>Exemplu:</u>

```
class c1{
      double n1, n2;
public:
      c1(){n1=0; n2=0;}
      c1(double x, double y)
            {n1=x; n2=y;}
};
c1 *pct1;
pct1=new c1[100]; /*Se rezervă memorie ptr. 100 obiecte de tip c1. Se apelează constructorul
implicit de 100 de ori */
}
```

<u>Exemplu:</u>

```
#include <iostream.h>
class numar
{     double *n;
public:
      număr (double nr1);
      ~număr();
       double val(){return *n;}
};
număr::număr(double nr1)
```

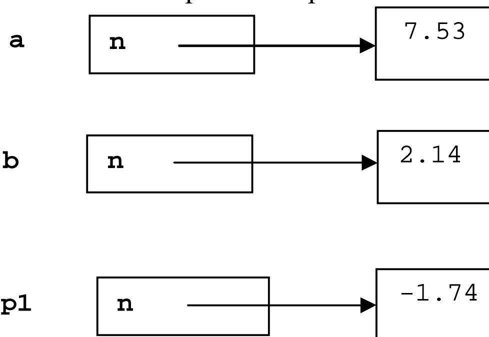


Figura 11.5. Obiectele `a`, `b`, `p1` de tip număr şi pointer spre număr

```
        {n=new double(nr1);}
număr::~număr()   { delete n;}
void main()
      {număr a(7.53),b(2.14);
      număr *p1,*p2;    p1=new număr(-1.74);
      cout<<p1<<'\n';   cout<<&p1<<'\n';
      *p1=a;      cout<<"p1="<<p1<<'\n';  delete p1;  }
```

11.9. SUPRAÎNCĂRCAREAOPERATORULUI ( )

Supraîncărcarea operatorului ***"apel de funcție"*** permite crearea unui operator binar, nestatic, de forma:

`expresie (lista_param_efectivi);`

Avantajele unui astfel de operator sunt:
- Evaluarea şi verificarea listei de argumente în mod similar unei funcții obişnuite;
- Modul de funcționare a mecanismului de apel. Deşi operatorul este binar, cel de-al doilea operand fiind o listă de argumente (inclusiv listă vidă), funcția operator poate avea oricâți parametri.

În cazul în care numele funcției este un pointer către o anumită funcție (vezi pointeri către funcții), apelul funcției se realizează prin:

`(*point_f) (lista_param_efectivi);`

Funcția care supraîncarcă operatorul trebuie să fie ***metodă nestatică***. Supraîncărcarea operatorului **`( )`** se utilizează în mod frecvent la definirea aşa-numitului ***iterator***. Iteratorii se utilizează în legătură cu tipuri abstracte de date, care conțin colecții de elermente (liste, arbori, tabele de dispersie, etc.). Pe lângă protecția datelor, iteratorii oferă un mijloc simplu de acces la elementele unei colecții, fără a intra în detaliile legate de implementarea colecției (independență a utilizării unei colecții şi implementării unei colecții). În principiu, un iterator se implementează printr-o clasă ataşată unui tip abstract care conține o colecție de elemente. Fie, de exemplu, clasa `container` care este o colecție de obiecte de tip oarecare, care poate fi implementată ca un tablou de obiecte, ca o listă de noduri, ca un arbore binar, etc. În funcție de această organizare, clasa `container` conține o dată membră de tip `obiect` sau un ***pointer către*** `obiect` şi este capabil să livreze, pe rând, obiectele elemente ale colecției.

11.10. SUPRAÎNCĂRCAREAOPERATORULUI ->

Supraîncărcarea operatorului unar **`->`** se realizează printr-o ***metodă nestatică***. Expresia
`obiect -> expresie` va fi interpretată ca **`(obiect.operator->())->expresie`**
De aceea, funcția-operator trebuie să returneze fie un pointer la un obiect al clasei, fie un obiect de un tip pentru care este supradefinit operatorul **`->.`**

<u>Exemplu:</u>

```
#include <iostream.h>
typedef struct ex{      int membru; };

class ex1{
      ex *pointer;
public:
      void set(ex &p)         {pointer=&p;}
      ex * operator -> (void)       {return pointer;}
};

class ex2{
      ex1 *pointer;
public:
      void set(ex1 &p) {pointer=&p;}
      ex1 * operator -> (void)      {return pointer;}
};
```

```
void main()
{ex A; ex1 B; ex2 C; B.set(A);
B->membru=10;              //apel al funcţiei ex1::operator->()
cout<<B->membru<<'\n';  }
```

Exerciţiu: Se implementeaz clasa matrice. Matricea este privită ca un vector de linii.

Date membre (protected):

- ✓ `int Dim1,Dim2;` // nr. linii, nr. coloane
- ✓ `double *tab;` // pointer către tabloul componentelor
- ✓ `int err;` // indice de eroare

Metode:

- ✓ `matrice (int dim1=0, int dim2=0);` Constructor matrice, cu alocare dinamică.
- ✓ `matrice(const matrice&);` Constructor de copiere
- ✓ `~matrice();` Destructor, cu rolul de a elibera memoria alocată dinamic.
- ✓ `int pune(int ind1, int ind2, double elem);`
  Iniţializează elementul de indici (ind1, ind2) cu valoarea transmisă ca argument (al treilea parametru). Întoarce valoarea întreagă 1 dacă indicii sunt incorecţi.
- ✓ `int dim1()const;` Metodă constantă care returnează numărul de linii.
- ✓ `int dim2() const;` Metodă constantă care returnează numărul de coloane.
- ✓ `int nrerori() const;`

Metodă constantă (nu poate modifica obiectul curent) care returnează valoarea datei membre err;

- ✓ `void anulari();` Metodă care anulează indicele de eroare.
- ✓ `double elem(int ind1, int ind2) const;`
  Metodă constantă care returnează valoarea elementuluilui de indici (ind1,ind2).
- ✓ `friend ostream &operator<<(ostream &, const matrice&);`
  Supraîncărcarea operatorului de inserţie printr-o funcţie membră. Apelează metoda *afişare*.
- ✓ `void afişare(ostream &)const;`
- ✓ `matrice &operator=(const matrice&);`
  Supraîncărcarea operatorului de atribuire printr-o funcţie membră. A fost necesară supraîncărcarea operatorului de atribuire datorită faptului că această clasă conţine pointeri către datele membre.
- ✓ `int dimtab()const;`
  Metodă constantă care returnează numărul de elemente din matrice (Dim1*Dim2).
- ✓ `virtual int comparare(const matrice&) const;`
  Metodă constantă care compară obiectul curent cu obiectul primit ca argument.
- ✓ `matrice operator+(const matrice&) const;`
  Operator de adunare supraîncărcat prin metodă constantă care returnează o matrice (întoarce o copie care poate fi utilizată în programul apelant) reprezentând suma dintre obiectul curent şi cel primit ca argument.
- ✓ `matrice operator-(const matrice&) const;`
  Operator de scădere supraîncărcat prin metodă constantă care returnează o matrice (întoarce o copie care poate fi utilizată în programul apelant) reprezentând suma dintre obiectul curent şi cel primit ca argument.
- ✓ `matrice &operator+=(const matrice&);`
  Operator supraîncărcat prin metodă, deoarece întotdeauna operandul stâng este de tipul matrice. *Este folosit în expresii cum ar fi: a+=b* (a şi b de tipul matrice).
- ✓ `matrice &operator-=(const matrice&);`
  Operatorul -= supraîncărcat prin metodă, deoarece întotdeauna operandul stâng este de tipul matrice. *Este folosit în expresii cum ar fi: a-=b* (a şi b de tipul matrice).
- ✓ `friend matrice operator*(double, const matrice&);`
  Operator supraîncărcat prin funcţie prietenă, pentru a putea fi utilizat în expresii n*M, unde n este de tip real sau întreg şi M de tipul matrice.
- ✓ `matrice operator*(const matrice&) const;`
  Operator supraîncărcat prin funcţie membră, care înmulţeşte obiectul curent (tip matrice) cu obiectul primit ca argument (tot tip matrice).
- ✓ `int indtab(int i,int j) const;`

Metodă care returnează indicele din tabloul unidimensional (matricea este privită ca un tablou unidimensional, cu elementele memorate într-un tablou unidimensional, întâi elementele primei linii, în continuare elementele celei de-a doua linii, etc.) pentru elementul[i][j].

- ✓ `int erind(int, int) const;` Metodă care testează eventualele erori de indexare.
- ✓ `matrice transp() const;` Metoda calculează şi returnează matricea transpusă pentru obiectul curent. Nu modifică obiectul curent.
- ✓ **`double operator ( ) (int i, int j);`**

  Supraîncărcarea operatorului ( ) prin metodă a clasei care returnează valoarea elementului de indici i şi j pentru obiectul curent, sau 0 în cazul în care apare o eroare de indexare (vezi şi metoda `elem`). Apelul metodei elem: `A.elem(1, 3)` este echivalent cu `apelul A(1, 3)`.

```
//FISIERUL  matrice.h
#ifndef _iostream_h
#include <iostream.h>
#define _iostream_h
#endif

class matrice{
      int Dim1,Dim2;
      double *tab;
      int err;
public:
      matrice (int dim1=0,int dim2=0);
      matrice(const matrice&);
      ~matrice();
      int pune(int ind1, int ind2, double elem);
            //pune elem. elem pe poz. de indici (ind1, ind2); întoarce 1 dacă indicii sunt incorecți
      friend ostream &operator<<(ostream &, const matrice&);
      matrice transp() const;
      matrice &operator=(const matrice&);
      int dim1()const              {return Dim1;}
      int dim2() const             {return Dim2;}
      int dimtab() const;
      int nrerori() const          {return err;}
      void anulerori()             {err=0;}
      double elem(int ind1, int ind2) const; //întoarce val. elem-lui de indici (ind1,ind2)
      int comparare(const matrice&) const;
      matrice operator+(const matrice&) const;
      matrice operator-(const matrice&) const;
      matrice &operator+=(const matrice&);
      matrice &operator-=(const matrice&);
      friend matrice operator*(double, const matrice&);
      matrice operator*(const matrice&) const;
      // PTR MATRICI SIMETRICE:
      double operator()(int i, int j);

   private:
      int indtab(int i,int j) const
            {return i*Dim2+j;}
                  //indicele din tab al unui elem.
      int erind(int, int) const;
                  //test er. de indexare
      void afisare(ostream &)const;
      matrice inv() const;
};

// FISIERUL matrice.cpp
#ifndef _matrice_h
#include "matrice.h"
#define _matrice_h
```

```
#endif

matrice::matrice (int d1,int d2)
// constructor matrice
      {int k,dtab;      err=0;
      if (d1<=0 || d2<=0)     {Dim1=0;Dim2=0;tab=0;}
      else{ dtab=d1*d2; tab=new double[dtab];
            if (tab!=0)  { Dim1=d1;Dim2=d2;
             for (k=0;k<dtab;k++)         tab[k]=0;
             cout<<"Construit matr. de dim. "<<d1*d2<<'\n';
             }
            else {Dim1=0;Dim2=0;err=1;cout<<"Construit matr. de dim. 0"<<'\n';}
        }
      }
matrice::matrice(const matrice &M)
//constructor copiere
      {cout<<"Constructor copiere!\n";err=0;int k,dtab;
      if (M.Dim1<=0 || M.Dim2<=0)   {Dim1=0;Dim2=0;tab=0;}
      else{ dtab=M.Dim1*M.Dim2;tab=new double[dtab];
            if (tab!=0)
            { Dim1=M.Dim1; Dim2=M.Dim2;
            for (k=0;k<dtab;k++)
                  tab[k]=M.tab[k];
            }
      else {Dim1=0;Dim2=0;}
      }
      }
matrice::~matrice()
      { if (tab!=0) delete [] tab;}
int matrice::pune(int i,int j, double val)
      { int iret; iret=erind(i,j); if (iret!=0) return iret;
       tab[indtab(i,j)]=val; return 0;
      }
int matrice::erind(int i,int j) const
      { if (Dim1==0 ||Dim2==0)      return 2;
      if (i<0 || i>=Dim1 || j<0 || j>=Dim2)     return 1;
      return 0;
      }
void matrice::afisare(ostream & ies) const
      { int i,j;
      if (tab!=0){
        ies<<'\n';
        for (i=0;i<Dim1;i++){
            for (j=0;j<Dim2;j++)
                  ies<<tab[indtab(i,j)]<<' ';
            cout<<'\n';}
       }
      }
ostream &operator<<(ostream &ies, const matrice &M)
      { M.afisare(ies);return ies;}
matrice &matrice::operator=(const matrice &M)
      { int k,dtab,vdtab;cout<<"Operator atribuire!\n";
      err=M.err;dtab=M.Dim1*M.Dim2;
      //dimens. M
      if (dtab==0){ Dim1=0;Dim2=0;
              if (tab==0){delete [] tab;tab=0;}
      }
      else{ vdtab=Dim1*Dim2;
            if (vdtab!=dtab){
                delete [] tab;
                tab=new double [dtab];
            if (tab!=0){Dim1=0;Dim2=0;err=1;}
```

```
      }
      if (tab!=0){
            Dim1=M.Dim1;Dim2=M.Dim2;
            for (k=0;k<dtab;k++)
                  tab[k]=M.tab[k];
       }
      }
      return *this;     }
int matrice::comparare(const matrice &M) const
      { int k,dtab;if (M.Dim1!=Dim1 || M.Dim2!=Dim2) return 2;
      dtab=Dim1*Dim2;
      for (k=0;k<dtab;k++)
        if (M.tab[k]!=tab[k])   return 1;
      return 0;   }
matrice matrice::operator+(const matrice &B) const
      { matrice C;int k,dtab;
      if (Dim1!=B.Dim1 || Dim2!=B.Dim2)C.err=1;
      else{
        dtab=Dim1*Dim2;  C.tab=new double [dtab];
        if (C.tab==0)   {C.Dim1=0; C.Dim2=0; C.err=2; }
        else     {C.Dim1=Dim1;C.Dim2=Dim2;
                  if (dtab!=0)
                         for (k=0;k<dtab;k++)    C.tab[k]=tab[k]+B.tab[k];
                 }
       }
      return C;   }
matrice matrice::operator-(const matrice &B) const
      { matrice C;int k,dtab;
      if (Dim1!=B.Dim1 || Dim2!=B.Dim2)        C.err=1;
      else{
        dtab=Dim1*Dim2;C.tab=new double [dtab];
        if (C.tab==0)     {C.Dim1=0; C.Dim2=0; C.err=2; }
        else    {C.Dim1=Dim1;C.Dim2=Dim2;
                  if (dtab!=0)
                         for (k=0;k<dtab;k++) C.tab[k]=tab[k]-B.tab[k];
                 }
        }
      return C;   }
matrice &matrice::operator+=(const matrice &B)
      { int dtab;
      if (Dim1!=B.Dim1 || Dim2!=B.Dim2)       err++;
      else { dtab=Dim1*Dim2;
            if (dtab!=0)
                for (int k=0;k<dtab;k++)   tab[k]+=B.tab[k];
       }
      return *this;     }
matrice &matrice::operator-=(const matrice &B)
      { int dtab;
      if (Dim1!=B.Dim1 || Dim2!=B.Dim2)
              err++;
      else {  dtab=Dim1*Dim2;
        if (dtab!=0)
                for (int k=0;k<dtab;k++)   tab[k]-=B.tab[k];
       }
      return *this;     }
matrice operator*(double a, const matrice &B)
      { if (B.tab==0)  { matrice C; C.err=B.err; return C;}
       { int k,dtab;    matrice C(B.Dim1, B.Dim2);
       if (B.tab==0)    {C.err=3;return C;}
      dtab=C.Dim1*C.Dim2; for (k=0;k<dtab;k++)  C.tab[k]=a*B.tab[k];
       return C;
       }
      }
```

```
matrice matrice::operator*(const matrice &B) const
      { if (Dim2!=B.Dim2)       { matrice C; C.err=1; return C;}
       if (tab==0 && B.tab==0)  {matrice C; return C;}
       if (tab==0 || B.tab==0)  {matrice C; C.err=2; return C;}
       { int i,j,k; double S; matrice C(Dim1, B.Dim2);
       if (C.tab==0)   { C.err=3; return C;}
       for (i=0;i<Dim1;i++)
              for (j=0;j<B.Dim2;j++)
                { S=0;
                  for (k=0;k<Dim2;k++)
                        S+=tab[indtab(i,k)]*B.tab[B.indtab(k,j)];
                  C.tab[C.indtab(i,j)]=S;
                 }
        return C;
        }
      }
matrice matrice::transp() const
      { int i,j,dtab; dtab=Dim1*Dim2;
      if (dtab==0)        {matrice C; C.err=err; return C; }
      { matrice C(Dim2,Dim1);
        if (C.tab==0)        {C.err=1; return C; }
        for (i=0;i<C.Dim1;i++)
              for (j=0;j<C.Dim2;j++)
                C.tab[C.indtab(i,j)]=tab[indtab(j,i)];
        return C;
       }
      }
double matrice::elem(int i,int j) const
      { if (erind(i,j))        return 0;
        return tab[indtab(i,j)];}
int matrice::dimtab() const
      { return Dim1*Dim2; }

/*PTR. MATRICI SIMETRICE:
double matrice::operator ()(int i, int j)
      { if (erind(i,j))  return 0;
         return tab[indtab(i,j)];
}*/
```

```
// FISIERUL test_matrice.cc
#ifndef _iostream_h
#include <iostream.h>
#define _iostream_h
#endif

#ifndef _matrice_h
#include "matrice.h"
#define _matrice_h
#endif

void main()
{int M,N; cout <<"Nr. linii:"; cin>>M; cout <<"Nr. coloane:"; cin>>N;
{matrice A(M,N),B(4,4);matrice C();int i,j; double val;
// introduc matr. A(M,N)
for (i=0;i<M;i++)
        for (j=0;j<N;j++) {
                cout<<"A["<<i<<","<<j<<"]="; cin>>val;
                A.pune(i,j,val); }
cout<<"Matr. introdusa:\n";cout<<A<<'\n';matrice E(A); //apel constr. copiere
cout<<"E="<<E<<'\n'; matrice D=A;                       //constr. copiere
cout<<"D="<<D<<'\n'; matrice F(M,N);cout<<"Inainte de atrib. F=\n";
cout<<F<<'\n';F=A;cout<<"Dupa atrib.F=\n"<<F<<'\n';int comp=F.comparare(A);
```

```
if (comp==0)    cout<<"Matrici identice\n!";
else if (comp==2)       cout<<"Matrici de dim. diferite!\n";
else    cout<<"Matr. cu elem. diferite!\n";
E.pune(0,0,100.5); comp=E.comparare(A);
if (comp==0)    cout<<"Matrici identice\n!";
else if (comp==2)       cout<<"Matrici de dim. diferite!\n";
else    cout<<"Matr. cu elem. dif!\n";
cout<<"A+A="<<(A+A)<<'\n';cout<<"A-A="<<(A-A)<<'\n';
A+=E; cout<<"A="<<A<<'\n';cout<<"D=A"<<(D=A)<<'\n';cout<<"A="<<A<<'\n';
cout<<"A*A="<<(A*A)<<'\n'; cout<<(A.transp())<<'\n';
}
matrice G(5);     }
```

11.11. CONVERSII

Există următoarele tipuri de conversii:
- Conversii ***implicite***;
- Conversii ***explicite***.

Conversiile implicite au loc în următoarele situații:
- În cazul *aplicării operatorului de atribuire*: operandul drept este convertit la tipul operandului stâng.
- La *apelul unei funcții*: Dacă tipul parametrilor efectivi (de apel) diferă de tipul parametrilor formali, se încearcă conversia tipului parametrilor efectivi la tipul parametrilor formali.
- La revenirea dintr-o funcție: Dacă funcția returnează o valoare în programul apelant, la întâlnirea instrucțiunii ***return*** *expresie;* se încearcă conversia tipului expresiei la tipul specificat în antetul funcției.

Conversiile explicite pot fi :

a) tip_predefinit_1 -> tip_predefinit_2
b) tip_predefinit -> tip_definit_de_utilizator (clasă)
c) clasă -> tip_predefinit
d) clasă_1 -> clasă_2

11.11.1. CONVERSII DIN TIP PREDEFINIT1 ÎN TIP PREDEFINIT2

Pentru realizarea unor astfel de conversii, se folosește operatorul unar de conversie explicită (cast), de forma:

`(tip) operand`

Exemplu:

```
int k; double x;
x = (double) k / (k+1);
```

/* În situația în care se dorește obținerea rezultatului real al împărțirii întregului k la k+1, trebuie realizată o conversie explicită, vezi capitolul 2.7. */

În limbajul C++ același efect se poate obține și astfel:

```
x= double (k) / (k+1);
```

deoarece se apelează explicit constructorul tipului double.

11.11.2. CONVERSII DIN TIP PREDEFINIT ÎN CLASĂ

Astfel de conversii se pot realiza atât ***implicit***, cât și ***explicit***, în cazul în care pentru clasa respectivă există un constructor cu parametri impliciți, de tipul predefinit.

Exemplu: Pentru clasa `fracție` definită în cursurile anterioare:

```
class  fracție{
      int nrt, nmt;
public:
      fracție( int nrt = 0, int nmt = 1);
      //...
```

```
};
fracţie f;
f = 20;
```
/* Conversie IMPLICITĂ: înaintea atribuirii se converteşte operandul drept (de tip `int`) la tipul operandului stâng (tip `fracţie`). */
```
f = fractie(20);
```
/*Conversie EXPLICITĂ: se converteşte întregul 20 într-un obiect al clasei fracţie (nrt=20 şi nmt=1). */ }

11.11.3. CONVERSII DIN CLASĂ ÎN TIP PREDEFINIT

Acest tip de conversie se realizează printr-un operator special (cast) care converteşte obiectul din clasă la tipul predefinit. Operatorul de ***conversie explicită*** se supraîncarcă printr-o ***funcţie membră nestatică***.

```
nume_clasa:: operator nume_tip_predefinit( );
```

La aplicarea operatorului se foloseşte una din construcţiile:

```
(nume_tip_predefinit)obiect;
nume_tip_predefinit (obiect);
```

Exemplu: Pentru clasa fracţie, să se supraîncarce operatorul de conversie explicită, care să realizeze conversii **`fracţie -> int`**.

```
#include <iostream.h>
class fracţie{
        long nrt, nmt;
public:
        fracţie(int n=0, int  m=1)        {nrt=n; nmt=m;}
        friend ostream &operator<<(ostream &, const fracţie &);
        operator int( ) {return nrt/nmt;}        //conversie fracţie -> int
};
ostream &operator<<(ostream &ies, const fracţie &f)
      {ies<<'('<<f.nrt<<'/'<<f.nmt<<")\n"; return ies;}

void main()
{     fracţie a(5,4), b(3), c;  int  i=7, j=14;    c=a;   c=7;
      cout<<(fracţie)243<<'\n';  cout<<"(int)a="<<(int)a<<'\n';
      //conversie explicită
      cout<<"int(a)="<<int(a)<<'\n';   //conversie explicită
      int x=a;               //conversia se face implicit, înainte de atribuire
}
```

11.11.4. CONVERSII DIN CLASĂ1 ÎN CLASĂ2

Conversia din *tip_abstract_1* în *tip_abstract_2* (din `clasă1` în `clasă2`), se realizează cu ajutorul unui constructor al clasei2, care primeşte ca parametri obiecte din clasa1 (fracţie -> complex).

Exemplu:

```
#include <iostream.h>
class fracţie{
       int  nrt, nmt;
public:
        fracţie(int nr=0, int nm=1) {nrt=nr; nmt=nm;}
        operator int() const {return nrt/nmt;} //conversie fracţie -> int
        friend ostream &operator<<(ostream&, const fracţie&);
        friend class complex;
/*cls. complex este clasa prietena ptr. clasa fracţie, fiecare funcţie din complex este prietena pentru fracţie*/
        int întreg(){return nrt/nmt;}
};
ostream &operator <<(ostream &ostr, const fracţie &f)
      {ostr<<'('<<f.nrt<<'/'<<f.nmt<<")\n";return ostr;}
```

```
class complex{
        double re, im;
public:
        complex (double r=0, double i=0)        {re=r; im=i;}
        complex (fracţie &f) {re=(double)f.nrt/f.nmt; im=0;}
            // conversie fracţie->complex
        operator double() const  {return re;}//conversie complex->double
        friend ostream &operator<<(ostream &, const complex &);
};
ostream &operator<<(ostream &ies, const complex &z)
      {ies<<'('<<z.re<<','<<z.im<<")=\n"; return ies;}

void main()
{     complex a(6.98, 3.2), b(9), c, d, e, q, s; int i=12, j=5, k;
      double x=1234.999, y=74.9897, u, z; c=i; fractie r(7, 3), r1(9), t;
      d=x; //conversie double->complex (constructor complex cu arg. double)
      e=complex(y,j); //apel explicit al constr. complex; întâi conversie j de la int la double
      k=a; //conversie complex->double->int
      u=a; //conversie complex->double
      z=(double)a/3; //conversie complex->double
      cout<<"r="<<r<<" q="<<q<<'\n';
      cout<<"int(r)="<<int(r)<<" (int)r="<<(int)r<<'\n';
      //conversie fractie->int
      cout<<"r="<<r<<'\n';    cout<<r.intreg()<<'\n';
      s=r;               // conversie fracţie->complex
      cout<<"(complex)r="<<(complex)r<<" complex (r)="<<complex (r)<<'\n';
      // conversie fracţie->complex
}
```

ÎNTREBĂRI ŞI EXERCIŢII

Chestiuni teoretice

1. Cum se realizează conversia din clasă1 în clasă2? Daţi un exemplu.
2. Prin ce modalităţi se pot supraîncărca operatorii?
3. În ce situaţii se realizează conversiile implicite?
4. Cum se poate realiza conversia dintr-un tip abstract (clasă) într-un tip predefinit? Exemplu.
5. Ce observaţii puteţi face în legatură cu aritatea unui operator şi modul de supraîncarcare a acestuia?
6. Cum se realizează conversia din tip predefinit în clasă?
7. Ce restricţii impune mecanismul de supraîncărcare a operatorilor?
8. În cazul supradefinirii metodelor, cum se poate realiza selecţia unei metode ?

Chestiuni practice

1. Pentru toate tipurile de date implementate, să se completeze programele de test, astfel încât să se verifice toţi operatorii supraîncărcaţi.
2. Pentru clasa fracţie, să se supraîncarce operatorul unar ++ printr-o funcţie membră şi operatorul -- printr-o funcţie prietenă. Să se completeze funcţia main, astfel încât să se testeze toţi operatorii supradefiniţi.
3. Fie clasa complex, cu datele membre parte reala şi parte imaginară. Să se supraîncarce operatorul extractor. Să se supraîncarce operatorul binar de împărţire, care realizează operaţii de forma c/d, unde c este complex şi d este real. Să se supraîncarce operatorul de scădere printr-o funcţie membră.

4. Fie clasa fracție, cu membrii numitor şi numărător. Să se supraîncarce operatorul / binar astfel încât să se poata realiza operații de forma b/f, unde b este întreg, iar f este fracție. Să se supraîncarce operatorul ++ care realizează incrementarea unei fracții.
5. Fie clasa şir, declarată conform modelului din curs. Să se defineasca pentru aceasta un constructor de copiere. Să se supraîncarce operaratorul == care compară două şiruri.
6. Fie clasa fracție, cu membrii numitor şi numărător. Să se supraîncarce operatorul insertor. Să se supraîncarce operatorul binar de înmulțire, printr-o funcție membră.
7. Fie clasa complex. Să se supraîncarce operatorul de înmulțire a 2 numere complexe, printr-o funcție prietenă. Să se supraîncarce operatorul de înmulțire, astfel încât sa fie posibile operații de forma c*a, unde a este un întreg, iar c este un obiect de tip abstract complex. Să se supraîncarce operatorul de împărțire a două obiecte complexe, printr-o funcție membră. Să se supraîncarce operatorul – unar care schimbă semnul părților reale şi imaginare ale unui complex.
8. Fie clasa şir. Să se supraîncarce operatorul + care realizează concatenarea a 2 şiruri. Să se implementeze metoda `caută_nr_apariții` care caută de câte ori apare un caracter transmis ca argument într-un şir şi returnează numărul de apariții sau 0 în cazul în care acel caracter nu este găsit. Să se supraîncarce operatorul `!` care transformă caracterele din conținutul şirului din litere mari în litere mici. Să se definească destructorul pentru şir. Să se supraîncarce operatorul binar ŞI logic (pe cuvânt) care din două şiruri s1 şi s2, construieşte un alt şir, care conține caracterele comune lui s1 si s2. Să se supradefinească operatorul != care testează existența unui caracter (dat ca argument) într-un şir. Dacă acel caracter este conținut în şir, se returnează valoarea 1, altfel – valoarea 0. Să se supraîncarce operatorii relaționali care compară lexicografic conținutul a două şiruri. Să se supraîncarce operatorul – unar care realizează conversia tuturor caracterelor alfabetice din conținutul unui obiect de tip şir, din litere mari în litere mici.
9. Fie clasa vector. Să se supraîncarce operatorul + care realizează adunarea a 2 vectori. Să se supraîncarce operatorul * care realizează produsul scalar a 2 vectori.
10. Să se definească tipul abstract dată calendaristică. Data calendaristică se va introduce sub forma zz/ll/aaaa, fiind validată (se va ține cont de anii bisecți). Se vor supraîncărca operatorii insertor, extractor, a+= (adună la dată un număr de zile),-= (scade dintr-o dată un număr de zile), == (compară 2 date), - (returnează numărul de zile dintre două date), + (adună două date), ++ (incrementează luna), şi -- (decrementează luna).
11. Să se adauge la clasele punct şi segment metode de desenare, de rotire a unui segment în jurul vârfului.
12. Să se scrie un program care translatează coordonatele vârfurilor unui triunghi, şi desenează triunghiul înainte şi după translație, folosindu-se o clasă punct. Se va modifica ulterior programul declarându-se o clasa triunghi care conține ca membri 3 obiecte de tipul punct. Se va modifica programul, astfel încât clasa triunghi să aibă ca membru un vector de puncte.
13. Pentru clasa şir, să se supraîncarce următorii operatori:

    `int intr_p(const sir &s) const;`

    Determină prima apariție a şirului s în şirul curent. Dacă s este subşir, returnează indicele primului caracter al acestei intrări; altfel, returnează -1.

    `int intr_n(const sir &s, const unsigned n) const;`

    Determină a n-a apariție a şirului s în şirul curent. Returnează indicele primului caracter al acestei intrări; altfel, returnează -1.

    `sir stregcar (unsigned i, unsigned n) const;`

    Returnează şirul rezultat prin ştergerea din obiectul curent a cel mult n caractere, începând cu poziția i. Dacă i sau n sunt eronate, se returnează obiectul curent nemodificat.

    `sir operator - (const sir &s) const;`

    Returnează obiectul rezultat prin eliminarea sufixului s din şirul curent. Returnează şirul curent dacă s nu este sufix, 0 pentru memorie insuficientă.

    `sir operator % (const sir &s) const;`

    Returnează obiectul rezultat prin eliminarea prefixului s din şirul curent. Returnează şirul curent dacă s nu este prefix, 0 pentru memorie insuficientă.

    `sir operator * (const sir &s) const;`

    Returnează obiectul rezultat prin eliminarea primei intrări a lui s în şirul curent.

    `sir operator / (const sir &s) const;`

    Returnează obiectul rezultat prin eliminarea ultimei intrări a lui s în şirul curent.

    `sir operator( ) (const sir &s1, const sir &s2, int poz);`

Returnează obiectul rezultat prin înlocuirea unei intrări a lui s1, cu s2. Dacă poz este o, se substiuie prima intrare, altfel - ultima intrare. Dacă s1 nu este subşir al obiectului curent, se returnează obiectul curent.

```
sir operator( ) (const sir &s1, const sir &s2) const;
```

Returnează obiectul rezultat prin înlocuirea în obiectul curent a tuturor intrărilor lui s1, cu s2.