6. REDEFINIREA OPERATORILOR ȘI FUNCȚIILOR

Limbajul C++ are posibilitatea redefinirii funcțiilor și operatorilor. Mai exact, unei funcții (metode) i se conferă posibilitatea de a "înțelege" și trata în mod corespunzător mai multe liste diferite de intrare. De asemenea, operatorilor uzuali li se pot atribui semantici noi. În acest capitol se vor aborda următoarele subiecte: redefinirea unei funcții, redefinirea operatorilor, restricții referitoare la funcțiile de tip **operator**, funcții de tip **operator** și prieten al clasei, prezentarea unor operatori speciali și problemele folosirii lor etc.

6.1. Redefinirea unei funcții

O funcție redefinită (*overloaded function*) este o funcție cu mai multe definiții. În capitolele anterioare s-a remarcat că pentru anumite funcții membre ale unor clase existau mai multe declarații (prototipuri) și definiții (implementări). Considerăm un exemplu în care redefinim funcția denumită suma() în trei ipostaze. Primele două se referă la adunarea a doi parametri de tip **int** și respectiv **double**, iar ultima se referă la concatenarea a două șiruri.

```
// Program P6_1.CPP   Redefiniri de funcții
# include <stdio.h>
# include <string.h>
int suma(int i1, int i2) { return i1+i2; }
double suma (double op1, double op2) { return op1+op2; }
char *suma(char *sir1, char *sir2) { return strcat(sir1, sir2); }
void  main (void)
{       int  k1 = 10, k2 = 20;
        printf("Suma a 2 numere intregi este = %d\n", suma(k1, k2));
        double x1 = 1.1; double x2 = 2.2;
        printf("Suma a 2 numere reale este = %f\n", suma(x1, x2));
        char *sir1 = "abc";
        char *sir2 = "def";
        printf("Rezultatul alipirii lui sir1 cu sir2 este = %s\n", suma(sir1, sir2));
}
```

În funcția main() se apelează pe rând cele trei forme ale funcției suma(). În interiorul programului se creează lanțul necesar de legături și în funcție de tipul argumentelor, la momentul execuției programului, se cheamă metoda adecvată tratării acestui mesaj. Sunt lansate trei mesaje și anume k1, k2 (de tip **int**, **int**), apoi x1, x2 (de tip **double**, **double**) și în final sir1, sir2 când de fapt are loc concatenarea. Există, în acest exemplu, o totală coincidență a tipurilor argumentelor cu cele ale parametrilor.

Vom considera acum un exemplu în care, pe lângă asocierea metodei adecvate de tratare a mesajului de prelucrat, compilatorul va fi obligat să efectueze și conversiile necesare. Se va redefini și evident apela o funcție denumită valpolin() care va determina valoarea unui polinom $P(x) = a_0x^n+a_1x^{n-1}+...+a_n$ într-un anume punct x. Se va recurge la funcția poly() al cărei prototip definit în fișierul antet <**math.h**> este:

```
        double poly(double x, int grad_polinom, double vector_coeficienti[]);
```

Listingul programului este următorul:

```
// Program P6_2.CPP    Valoarea unui polinom
# include <stdio.h>
# include <math.h>

double valpolin (double x, int n, double *c);      // Primul prototip al funcției valpolin()
long valpolin (int x1, int n1, double *c1);        // Al doilea prototip al funcției valpolin()
```

```
void  main (void)
{       double coef[] = {1., -4., 6., -4., 1.};          // Un polinom de gradul 4, P4(x)
        double rez;
        double coef1[] = {-1., 1., -1., 1., -1., 1.};    // Un polinom de gradul 5, P5(x)
        long rez1;
        rez = valpolin (10., 4, coef);
        printf ("Rezultatul interpolarii polinomului P4(10.) este = %lf\n", rez);
        rez1 = valpolin (10, 5, coef1);
        printf ("Rezultatul interpolarii polinomului P5(10) este = %ld\n", rez1);
}
double valpolin (double x, int g, double c[])
        { return poly (x, g, c); }
long valpolin (int x, int g, double c[])
        { return (long) poly (x, g, c); }
```

Alegerea metodei de prelucrare se face în funcție de tipul parametrilor din lista de apel a metodei. *Mai exact, se încearcă găsirea unei metode care are același prototip cu tipurile argumentelor prezentate în mesaj*. Astfel, pentru prelucrarea mesajului (10., 4, coef), adică (**double**, **int**, **double**[]) prelucrarea se realizează cu prima formă a lui valpolin(), iar pentru prelucrarea mesajului (10, 5, coef1), adică (**int**, **int**, **double**[]) prelucrarea se realizează cu cea de-a doua formă a lui valpolin().

6.2. Redefinirea operatorilor

În acest paragraf este vorba de funcții care au un șablon în care apare cuvântul cheie **operator** de forma următoare:

```
tip_rezultat     operator  op (lista_de_argumente)
{
        Corpul funcției
}
```

unde **op** poate fi unul din următorii operatori: aritmetici (+, -, *, /, %), de atribuire (=, +=, -=, *=, /=, %=), de incrementare (++), de decrementare (--), logici (&&, ||, !), unari (+, -), operatori de manipulare la nivel de bit (&, |, ~, ^, <<, >>) etc.

Exemplul următor conține o clasă denumită ADUN, în care d este singura variabilă de tip privat. Listingul în care apar și funcții de tip **operator** este următorul.

```
// Program P6_3.CPP    Clasa cu operator +
# include <stdio.h>
class ADUN {
        double  d;
public:
        ADUN (double x = 0) {d = x;}                  // Constructorul clasei
        ADUN operator + (ADUN celalalt)  { return ADUN (d + celalalt.d); }
        // Primul d din această definiție înseamnă "this->d" , adică obiectul curent
        void Imp (char *mesaj = " ")
                { printf ("%s  d = %f \n", mesaj, d); }
};
void  main (void)
{       ADUN ob1 (1.11), ob2 (2.22);          // Se crează două instanțe ale clasei ADUN
        ADUN ob3 = ob1 + ob2;                 // Se adună cele două instanțe
```

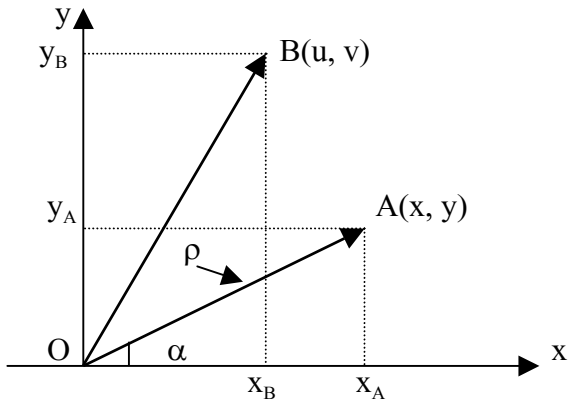
```
        ob3.Imp ("Obiectul rezultat ob3 = ob1 + ob2 contine valoarea: ");
}
```

Cum acest exemplu a conținut ca dată privată un singur scalar, în exemplul următor vom defini o clasă numită PUNCT. Aceasta se referă la obiecte (puncte) din sistemul cartezian xOy și conține următoarele operațiuni care se pot efectua cu coordonatele acestor obiecte:

- atribuirea, în sensul preluării conținutului unui punct A(x, y);
- adunarea și scăderea coordonatelor a două puncte A(x, y) și B(u, v);
- rotirea razei vectoare ρ a unui punct A(x, y) cu un unghi α (radiani).

Definirea clasei în care apar o serie de redefiniri ale unor operatori utilizați pentru efectuarea unor operații cu obiecte (puncte) din planul xOy este realizată în fișierul antet PUNCT.H

```
// Fișier PUNCT.H utilizat în programul P6_4.CPP
class   PUNCT {
protected:
        double  x,                      // abscisa
                y;                      // ordonata
public:
        PUNCT (double xx = 0, double yy = 0)       // Constructorul clasei
                { x = xx; y = yy; }
// Funcțiile membre ale clasei și redefinirile operatorilor
        PUNCT operator= (PUNCT);        // Atribuirea valorii coordonatelor x și y
        double distOA ();               // Lungimea razei vectoare ro a unui punct din plan
        double alfa ();                 // Unghiul razei ro cu axa Ox
        PUNCT operator + (PUNCT);       // A(x,y) + B(x,y), fără modificarea lui A(x,y)
        PUNCT operator - (PUNCT);       // A(x,y) - B(x,y), fără modificarea lui A(x,y)
        PUNCT operator ^ (double);      // Rotire cu un unghi alfa
        void Imp (char *mesaj = " ");
};
```



Programul propriu-zis care folosește acest fișier antet este următorul:

```
// Program P6_4.CPP  // Definițiile propriu-zise ale operatorilor
# include <stdio.h>
# include <math.h>
# include "punct.h"
const double epsilon = 0.1e-6;

PUNCT PUNCT :: operator = (PUNCT p) {
        x = p.x;  y = p.y;
```

```
        return *this;                   // Se întoarce chiar obiectul curent
}
double PUNCT :: distOA ()   { return sqrt (x*x + y*y); }   // ρ = √(x² + y²)
double PUNCT :: alfa ()     { return atan2 (y, x); }       // α = arctg(y/x)
PUNCT PUNCT :: operator + (PUNCT p)
        { return PUNCT (x+p.x, y+p.y);  }
PUNCT PUNCT :: operator - (PUNCT p)
        { return PUNCT (x-p.x, y-p.y);  }
PUNCT PUNCT :: operator ^ (double a) {
        double xrotit = distOA() * cos(alfa() + a);
        double yrotit = distOA() * sin(alfa() + a);
        return PUNCT (xrotit, yrotit);
}
void PUNCT :: Imp (char *m) {
        if (*m)
                printf ("%s ", m);
        printf ("abscisa x = %f, ordonata y = %f\n", x, y);
}
void main (void)
{
   PUNCT a, b(10., 10.), c(-5., -5.);
   double unghi = M_PI_4;                   // În <math.h> - pi/4
   b.Imp ("Punctul b de coordonate: ");
   c.Imp ("Punctul c de coordonate: ");
   a = b + c;
   a.Imp ("Punctul a = b+c de coordonate: ");
   a = b - c;
   a.Imp ("Punctul a = b-c de coordonate: ");
   printf ("Raza vectoare = %f\n", a.distOA());
   printf ("si azimutul = %f\n", a.alfa());
   a = a ^ unghi;
   a.Imp ("Punctul a, dupa rotire:");
   printf ("Raza vectoare = %f\n", a.distOA());
   printf ("si azimutul = %f\n", a.alfa());
}
```

În funcția main() se creează trei obiecte: a, b(10., 10.) și c(-5., -5). Folosind constanta M_PI_4, adică $\pi/4$ din fișierul <**math.h**>, se realizează operațiuni asupra punctelor b și c. În legătură cu modul de implementate a membrilor clasei PUNCT trebuie făcute următoarele observații:

1. Funcția de tipul **operator** de atribuire

```
        PUNCT PUNCT :: operator = (PUNCT p) {
                x = p.x;  y = p.y;
                return *this;     }     // Se întoarce chiar obiectul
```

recurge la operatorul **this**. Primul cuvânt PUNCT reprezintă tipul rezultatului acestei funcții, iar cel de-al doilea, denumirea clasei. Funcția preia coordonatele parametrului p (obiectului de tip PUNCT) și inițializează variabilele **private** x și y ale obiectului curent (obiectul **this**). Notația ***this** înseamnă transmiterea funcției apelante a conținutului acestui obiect. Notația **this** înseamnă întoarcerea adresei (referinței) obiectului curent.

2. În funcția membră a clasei și de tipul **operator**:

PUNCT PUNCT :: **operator** + (PUNCT p)

{ **return** PUNCT (x+p**.**x, y+p**.**y); }

expresia PUNCT (x+p**.**x, y+p**.**y) echivalează cu crearea unui obiect cu durata de viață limitată. Rezultatul său este pasat sub forma unei structuri de tip PUNCT. Deci, se creează în mod adhoc un obiect curent de tip PUNCT și când se ajunge la acolada "}", obiectul "dispare", eliberând resursele de memorie.

3. Se remarcă de asemenea o notație de forma:

double PUNCT :: distOA () { **return** sqrt (x*x + y*y); }

în care tipul rezultatului nu mai este o structură de tip PUNCT, ci un scalar de tip **double**.

6.3. Restricții referitoare la funcțiile de tipul operator

1. Nu se admite inventarea altor simboluri pentru operatori decât cei utilizați în C++. De exemplu, PUNCT PUNCT :: **operator @** (PUNCT) nu ar fi acceptat, deoarece simbolul @ nu este operator.

2. Nu se pot combina operatorii cunoscuți pentru a crea alți opratori. De exemplu, funcția **operator** PUNCT PUNCT :: **operator +-** (PUNCT, PUNCT) nu este legală.

3. Nu se poate modifica clasa operatorului, adică un operator binar nu poate deveni operator unar și invers. De exemplu, nu se poate scrie PUNCT PUNCT :: **operator --** (PUNCT, PUNCT) deoarece operatorul "--" este unar.

4. Nu se poate modifica prioritatea operatorilor.

5. Cel puțin unul din argumentele funcției de tipul **operator** trebuie să fie un obiect al clasei, sau funcția **operator** trebuie să fie o funcție membră a clasei. De exemplu, **double operator** + (**double**, **double**) nu poate fi acceptată nici ca funcție membră, nici ca nemembră. În funcție de faptul dacă o funcție **operator** este sau nu de tipul membu, forma de apel "externă" diferă. Menționăm totuși că această formă de apel se caracterizează printr-o maximă claritate. Totodată, precizăm că există și o formă "internă" (canonică) de apel, dar aceasta este mai ilizibilă. În tabelele următoare se prezintă ambele forme de scriere.

a) Pentru operatorul binar **op**

Tabelul 6.1a

Funcția	**Sintaxa formei externe de apel**	**Sintaxa formei interne (canonice) de apel**
membră	obiect1 **op** obiect2	obiect1**. operator op** (obiect2)
nemembră	obiect1 **op** obiect2	**operator op** (obiect1, obiect2)

b) Pentru operatorul unar **op**

Tabelul 6.1b

Funcția	**Sintaxa formei externe de apel**	**Sintaxa formei interne (canonice) de apel**
membră	**op** obiect1 obiect1 **op**	obiect1**. operator op** () obiect1**. operator op** ()
nemembră	**op** obiect1 obiect1 **op**	**operator op** (obiect1) **operator op** (obiect1)

Observații:

În contextul funcției membre a clasei de tipul **operator op** (binar), forma canonică este obiect1**. operator op** (obiect2). Se observă că deși operatorul este binar, listă de parametri nu conține doi parametri ci numai unul, și anume pe cel explicit. Cel de al doilea parametru se referă la obiectul curent, adică cel punctat prin operatorul **this**. Mai mult, în contextul operatorului unar **op**, în listă nu apare nici un parametru explicit. În acest caz, operatorul se referă la obiectul implicit, **this**. De exemplu, funcția **operator +** din exemplul de mai sus este definită astfel:

```
PUNCT PUNCT :: operator + (PUNCT p)
        { return PUNCT (x+p.x, y+p.y);  }
```

Instrucțiunea ce formează corpul funcției se poate scrie sub forma:

```
        return PUNCT (this -> x + p.x, this -> y + p.y);
```

6. Referitor la operatorii unari **++** și **--** trebuie precizat că atunci când aceștia sunt redefiniți, notația **operator ++** și **operator --** nu va putea fi extinsă și la formele **++ operator** și **-- operator**. În limbajul C++, se va păstra *numai* sensul operației **a++**, respectiv **a--**, *nu* și **++a**, respectiv **--a**.

7. Această observație este legată de redefinirea operatorilor **&&** și ||. Se știe că o expresie de forma a && b încetează să mai fie evaluată dacă **a** are valoarea 0 (FALSE) și că o expresie de forma a || b, de asemenea încetează să mai fie evaluată dacă **a** are valoarea diferită de zero (TRUE). Aceste reguli nu se mai aplică în cazul redefinirii acestor operatori: evaluarea va continua indiferent de valoarea obiectului **a**.

6.4. Funcții de tip friend și operator

Considerăm o clasă ce conține vectori alcătuiți din câte trei elemente. Asupra acestora acționează o serie de funcții ale căror prototipuri și definiții sunt prezentate în listingul următor:

```
// Program P6_5.CPP    Funcții de tip friend și operator
# ifndef VECTOR_H
# define VECTOR_H
# include <iostream.h>

class VECT {                    // Definirea clasei denumită VECT
        double v[3];
public:
   VECT ()                      // Funcția constructor
   { v[0] = v[1] = v[2] = 0.0; }
   ~VECT () {}                  // Funcția destructor - elimină întregul vector
   VECT (double x1, double x2, double x3);      // x(a,b,c) - vectorul x are elementele reale
   VECT (VECT &);                               // x = y ; vectorul x preia valoarea vectorului y
   VECT (double);               // Elementele lui x se actualizează cu o constantă reală
   VECT (int);                  // Elementele lui x se actualizează cu o constantă întreagă
   VECT operator = (VECT);                      // x = y ; vectorul x preia valoarea vectorului y
   VECT operator = (double);                    // Elementele lui x vor fi o constantă reală
   VECT operator = (int);                       // Elementele lui x vor fi o constantă întreagă
   friend VECT operator + (VECT, VECT);         // x = a+b
   friend VECT operator - (VECT, VECT);         // x = a-b
   friend VECT operator * (VECT, VECT);         // x = a*b
   friend VECT operator / (VECT, VECT);         // x = a/b
   friend VECT operator + (VECT);               // x = +a
   friend VECT operator - (VECT);               // x = -a
   VECT operator += (VECT);                     // a += b
   VECT operator -= (VECT);                     // a -= b
   VECT operator *= (VECT);                     // a *= b
   VECT operator /= (VECT);                     // a /= b
   double &operator[](int);                     // double d = a[i]
   friend int operator == (VECT, VECT);         // int x = (a == b) - egalitatea a doi vectori
   friend int operator != (VECT, VECT);         // int x = (a != b) - ingalitatea a doi vectori
```

```
   friend ostream &operator << (ostream &, VECT);        // cout << obiect tip VECT
   double ps(VECT);                                      // x = a.ps(b) - produsul scalar a doi vectori
};

// Definițiile funcțiilor membru
// v = w. Valorile elementelor vectorului w sunt atribuite vectorului v
VECT :: VECT (VECT &w)                    // Funcție de atribuire (membră a clasei VECT)
{       v[0] = w.v[0];
        v[1] = w.v[1];
        v[2] = w.v[2];
}
// VECT v = scalar_v.m. Vectorului v i se atribuie o valoare scalară în virgulă mobilă
VECT :: VECT (double d)                   // Funcție de atribuire (membră a clasei VECT)
{       v[0] = v[1] = v[2] = d;
}
// VECT v = scalar_întreg. Vectorului v i se atribuie o valoare scalară întreagă
VECT :: VECT (int k)                      // Funcție de atribuire (membră a clasei VECT)
{       v[0] = v[1] = v[2] = k;
}
// VECT (x, y, x). Valorile vectorului v sunt inițializate cu trei numere reale
VECT :: VECT (double x, double y, double z)       // Funcție de atribuire (membră a clasei VECT)
{       v[0] = x;
        v[1] = y;
        v[2] = z;
}
// v = w. Valorile elementelor vectorului w sunt atribuite vectorului v
VECT VECT :: operator = (VECT w)          // Funcție de atribuire tip operator (membră a clasei VECT)
{       v[0] = w.v[0];
        v[1] = w.v[1];
        v[2] = w.v[2];
        return *this;                     // Se returnează vectorul curent v
}
// VECT v = scalar_v.m. Vectorului v i se atribuie o valoare scalară în virgulă mobilă
VECT VECT :: operator = (double d)        // Funcție de atribuire tip operator (membră a clasei VECT)
{       v[0] = v[1] = v[2] = d;
        return *this;                     // Se returnează vectorul curent v
}
// VECT v = scalar_întreg. Vectorului v i se atribuie o valoare scalară întreagă
VECT VECT :: operator = (int i)           // Funcție de atribuire tip operator (membră a clasei VECT)
{       v[0] = v[1] = v[2] = (double) i;
        return *this;                     // Se returnează vectorul curent v
}
// x = v1 + v2. Suma a doi vectori
VECT operator + (VECT v1, VECT v2)        // Funcție de tip friend si operator
{   VECT  elem ( v1.v[0] + v2.v[0],
                 v1.v[1] + v2.v[1],
                 v1.v[2] + v2.v[2] );
            return elem;                  // Se returnează referinta la obiectul local elem
```

```
}
// x = v1 - v2. Diferența a doi vectori
VECT operator - (VECT v1, VECT v2)              // Funcție de tip friend si operator
{   VECT  elem ( v1.v[0] - v2.v[0],
                 v1.v[1] - v2.v[1],
                 v1.v[2] - v2.v[2] );
          return elem;                          // Se returnează referinta la obiectul local elem
}
// x = v1 * v2. Produsul elementelor corespondente a doi vectori
VECT operator * (VECT v1, VECT v2)              // Funcție de tip friend si operator
{   VECT  elem ( v1.v[0] * v2.v[0],
                 v1.v[1] * v2.v[1],
                 v1.v[2] * v2.v[2]);
          return elem;                          // Se returnează referinta la obiectul local elem
}
// x = v1 / v2. Raportul elementelor corespondente a doi vectori
VECT operator / (VECT v1, VECT v2)              // Funcție de tip friend si operator
{   VECT  elem ( v1.v[0] / v2.v[0],
                 v1.v[1] / v2.v[1],
                 v1.v[2] / v2.v[2]);
          return elem;                          // Se returnează referinta la obiectul local elem
}
// x = +u. Vectorului x i se atribuie valorile vectorului +u
VECT operator + (VECT u)                        // Funcție de tip friend si operator +
{       VECT elem (+u.v[0], +u.v[1], +u.v[2]);
        return elem;                            // Se returnează referinta la obiectul local elem
}
// x = -u. Vectorului x i se atribuie valorile vectorului -u
VECT operator - (VECT u)                        // Funcție de tip friend si operator -
{       VECT elem (-u.v[0], -u.v[1], -u.v[2]);
        return elem;                            // Se returnează referinta la obiectul local elem
}
// v  += v1 sau v = v +v1
VECT VECT :: operator += (VECT v1)              // Funcție membră de tip operator +=
{       VECT  elem (  v[0] += v1.v[0],
                      v[1] += v1.v[1],
                      v[2] += v1.v[2]);
              return *this;                     // Se returnează obiectul curent elem
}
// v  -= v1 sau v = v - v1
VECT VECT :: operator -= (VECT v1)              // Funcție membră de tip operator -=
{       VECT  elem (  v[0] -= v1.v[0],
                      v[1] -= v1.v[1],
                      v[2] -= v1.v[2]);
              return *this;                     // Se returnează obiectul curent elem
}
// v  *= v1 sau v = v *  v1 (produsul elementelor corespondente)
VECT VECT :: operator *= (VECT v1)              // Funcție membră de tip operator *=
```

```
{       VECT  elem ( v[0] *= v1.v[0],
                     v[1] *= v1.v[1],
                     v[2] *= v1.v[2]);
              return *this;                          // Se returnează obiectul curent elem
}
// v /= v1 sau v = v / v1 (raportul elementelor corespondente)
VECT VECT :: operator /= (VECT v1)                   // Funcție membră de tip operator /=
{       VECT  elem ( v[0] /= v1.v[0],
                     v[1] /= v1.v[1],
                     v[2] /= v1.v[2]);
              return *this;                          // Se returnează obiectul curent elem
}
// d = u[j].  Variabila reala d preia continutul elementelor vectorului u
double &VECT :: operator [] (int j)                  // Funcție membră de tip operator []
{
       return v[(j >= 0) && (j < 3) ? j : 0];
}
// x = (v1 == v2). Intregul x preia rezultatul testului de egalitate a doi vectori
int operator == (VECT v1, VECT v2)                   // Funcție de tip friend si operator
{       return (v1.v[0] == v2.v[0] &&
                v1.v[1] == v2.v[1] &&
                v1.v[2] == v2.v[2]);
}
// x = (v1 != v2). Intregul x preia rezultatul testului de inegalitate a doi vectori
int operator != (VECT v1, VECT v2)                   // Funcție de tip friend si operator
{       return (v1.v[0] != v2.v[0] ||
                v1.v[1] != v2.v[1] ||
                v1.v[2] != v2.v[2]);
}
// cout << w. Afișarea conținutului unui vector
ostream & operator << (ostream& iesire, VECT w)      // Funcție de tip friend si operator
{       return iesire << "adica [" <<
              w.v[0] << "," <<
              w.v[1] << "," <<
              w.v[2] << "]";
}
// Produsul scalar  x = v1.ps(v2)
double VECT :: ps (VECT v1)                          // Funcție membra a clasei VECT
{       return (v[0]*v1.v[0] + v[1]*v1.v[1] + v[2]*v1.v[2]);
}
# endif

void main (void)
{       VECT x, y, a(1., 2., 3.), b(-1., -2., -3.);   // Se ceează obiectele x, y, a și b
        int rez;
        double d;
        x = -a;                                  // Vectorului x i se atribuie valorile vectorului -a
        y = +b;                                  // Vectorului y i se atribuie valorile vectorului +b
```

```
        cout << "x = - a, " << x << endl;
        cout << "y = + b, " << y << endl;
        x = a + b;
        y = a - b;
        cout << "x = a + b, " << x << endl;
        cout << "y = a - b, " << y << endl;
        x = a * b;
        y = a / b;
        cout << "x = a * b ," << x << endl;
        cout << "y = a / b , " << y << endl;
        x += x;
        y -= y;
        cout << "Dublarea lui x, x = x + x, " << x << endl;
        cout << "Scaderea y = y - y, " << y << endl;
        rez = a == b;                          // rez preia rezultatul de egalitate a vectorilor a si b
        cout << "rez = (a == b) care da " << rez << endl;
        rez = a != b;                          // rez preia rezultatul de inegalitate a vectorilor a si b
        cout << "rez = (a != b) care da " << rez << endl;
        d = a.ps (b);                          // d preia produsul scalar al vectorilor a si b
        cout << "Produsul scalar  (a.b)  este " << d << endl;
//
        cout << "\nVectorul x are valoarea anterioara, " << x << endl;
        d = x[2];
        cout << " d = x[2], adica " << d << endl;
        int i = 5;
        x = i;
        y = -i;
        cout << "\n x = i, " << x << endl;
        cout << "\n y = -i, " << y << endl;
}
```

Operațiile permise cu obiecte de tip VECT sunt atât cele binare cât și cele unare. Inițializările unui vector pot fi atât implicite (realizate prin intermediul funcției constructor), cât și explicite (realizate prin intermediul unor funcții membre ale clasei). Eliminarea din memorie a unui obiect de tip vector se face cu ajutorul destructorului ~VECT(). Se admit, de asemenea, și următoarele operațiuni:

a) Atribuirea: x = w;

Prototipul funcției de atribuire este VECT (VECT & w) sau mai pe scurt VECT (VECT &). Se vede că parametrul de atribuit w este o referință la un obiect de tipul VECT.

```
// Definiția funcției este:
        VECT::VECT (VECT &w)
        {       v[0] = w.v[0];
                v[1] = w.v[1];
                v[2] = w.v[2];
        }
```

Cele trei instrucțiuni de atribuire de mai sus inițializează elementele vectorului v al obiectului curent, **this**.

b) Atribuirea unui scalar real sau întreg tuturor elementelor vectorului v cu funcțiile VECT (**double**) și respectiv VECT (**int**).

c) Atribuirea efectuată prin intermediul funcției **operator =** pentru cele trei cazuri de mai sus, utilizând respectiv funcțiile:

VECT **operator =** (VECT) ; // Valorile unui vector sunt atribuite unui alt vector

VECT **operator =** (**double**) ; // Elementelor unui vector li se atribuie o valoare reală

VECT **operator =** (**int**) ; // Elementelor unui vector li se atribuie o valoare intreagă

Este evident faptul că cele două modalități de atribuire prezentate mai sus sunt echivalente.

d) O serie de funcții de tip **operator** sunt și de tipul **friend**. În acest sens există următoarele reguli pe care trebuie să le avem în vedere:

R1. Funcția de tip **friend** care are în lista de intrare un singur parametru folosește un operator **op** ce trebuie să fie de tip unar. Dacă funcția are o listă de intrare formată din doi parametri, operatorul **op** va trebui să fie unul de tip binar.

R2. În contextul unei funcții **operator** și membră a clasei, dacă **op** este unar, lista de intrare va trebui să fie vidă, iar când **op** este binar, lista va trebui să fie formată dintr-un singur parametru.

În cele ce urmează vom analiza sintaxa definirii acestor funcții. Astfel, sintaxa definirii unei funcții de tipul **operator** și care este și membră a clasei este:

tip_rezultat nume_clasă :: **operator op** (listă_de _parametri)

{ Corpul funcției}

Sintaxa folosită la definirea unei funcții **operator** care este admisă numai ca funcție de tip **friend** a clasei este:

friend tip_rezultat **operator op** (listă_de _parametri)

{ Corpul funcției}

Menționăm că utilizarea funcțiilor de tip **friend** în acest exemplu a fost motivată de realizarea unor operații cu vectori de tipul a+b, a-b, a*b, a/b sau a =+b, a=-b.

e) Funcția **friend** ostream & **operator <<** (ostream & VECT) este o funcție care se ocupă de listarea conținutului obiectului de tip VECT. Se redefinește de fapt operatorul <<, al cărui sens, în limbaj C/C++ este de deplasare spre stânga. Aici are rolul de transmitere a informațiilor la fișierul logic de ieșire.

f) Notația **double** & **operator []** (**int**) înseamnă redefinirea operatorului de indexare []. În sens normal, operatorul [] este folosit pentru adresarea unui element al vectorului. În contextul prezentat, acestuia i se păstrează aceeași semantică, deci se urmărește adresarea unui element din vectorul v (dată de tip **private**). Indicele este pasat ca argument de intrare de tip **int**.

g) În exemplul anterior, funcțiile declarate de tip **friend**,

friend VECT **operator op** (VECT, VECT);

op având valorile +, -, *, /, == și !=, se puteau defini și ca funcții membre ale clasei. În acest caz, în funcția main() trebuia folosită notația:

obiect1**.operator op** (obiect2).

h) Observația următoare se referă la operatorul **this** utilizat ca ***this**. Fie funcția de tipul **operator** și membră a clasei:

```
VECT VECT :: operator = (VECT w)
{       v[0] = w.v[0];
        v[1] = w.v[1];
        v[2] = w.v[2];
        return *this;   }
```

Valorile elementelor vectorului w sunt atribuite vectorului v. Notația ***this** înseamnă obținerea întregului obiect drept rezultat al unei instrucțiuni de atribuire (x = y).

i) Acestă ultimă observație se referă la o serie de funcții de tipul **operator** și **friend**, ca de exemplu funcția

VECT **oprerator +** (VECT v1, VECT v2) ;

În cadrul acestei funcții s-a creat o variabilă locală (un obiect local de tip VECT) denumită elem. Prin intermediul instrucțiunii **return** se pasează în exteriorul funcției referința la acest obiect local.

6.5. Trei operatori speciali și problemele folosirii lor

Ne vom referi la: operatorul de atribuire, =; operatorul de indexare, [] și operatorul de apel, ().

6.5.1. Operatorul de atribuire, =

În exemplul următor vom prezenta două modalități de utilizare a operatorului de atribuire: una neglijentă și care conduce la erori și una corectă, conducând evident la rezultate corecte.
Varianta incorectă de utilizare a operatorului de atribuire este următoarea:

```
// Program P6_6a.CPP   O problemă de finețe datorită utilizării neglijente a funcției strcpy()
# include <stdio.h>
# include <string.h>
# define MAXLEN 100
char *s1 = new char[MAXLEN];
char *s2 = new char[MAXLEN];

void  main (void)
{       strcpy(s1, "abc");                                  // Sirul s1 = abc
        strcpy(s2, "1234");                                 // Sirul s2 = 1234
        printf("\n Sirul punctat de s1 este: %s", s1);      // Se listează șirul s1
        printf("\n Sirul punctat de s2 este: %s", s2);      // Se listează șirul s2
        printf("\n\n Valoarea initiala a lui s1 = %p\n", s1); // Valoarea pointerului s1
        printf("Valoarea initiala a lui s2 = %p\n", s2);    // Valoarea pointerului s2
        s1 = s2;                                  // Se comandă copierea conținutului șirului s2 în s1
        printf("\n Noua valoarea a lui s1 = %p\n", s1);
        printf(" Noua valoarea lui s2 = %p\n", s2);
        printf("\n Sirul punctat de s1 este: %s", s1);
        printf("\n Sirul punctat de s2 este: %s", s2);
        delete s2;
        delete s1;
}
```

Pointerii s1 și s2 vor puncta, fiecare, câte un caracter. Instrucțiunea de atribuire s1 = s2; din main() nu realizează ceea ce doream, adică copierea conținutului obiectului s2 în obiectul s1. În realitate, așa cum se vede prin execuția programului, se copiază numai valoarea pointerului s2 în s1. Mai exact, după execuția acestui program se vor obține următoarele mesaje:

```
        Sirul punctat de s1 este: abc
        Sirul punctat de s2 este: 1234
        Valoarea initiala a lui s1 = 1EE8 (o valoare în hexazecimal)
        Valoarea initiala a lui s2 = 1F50
```

După comanda solicitată de noi:

```
        Noua valoarea a lui s1 = 1F50
        Noua valoarea a lui s2 = 1F50
        Sirul punctat de s1 este: 1234
        Sirul punctat de s2 este: 1234
```

Ultimele două instrucțiuni din funcția main() vor șterge cei doi punctatori s1 și s2 și nu cele două obiecte punctate de s1 și s2. Rezultatele obținute sunt deci incorecte. Pentru a rezolva corect această problemă, considerăm definiția clasei SIR, având structura din exemplul următor:

```
// Program P6_6b.CPP   Utilizarea îngrijită a operației de copiere a unor obiecte
# include <stdio.h>
# include <string.h>
# define MAXLEN 100
class SIR {
        char *p;
public:
        SIR(char *q) {p = new char[MAXLEN];    // Se alocă exact MAXLEN octeți
        strcpy(p, q);}                  // Se copiază șirul q la adresa p
        ~SIR() {delete p;}
        void operator = (SIR &);        // Se declară un membru de tip operator de atribuire
};
void SIR :: operator = (SIR &s)         // Definiția funcției membru
        {strcpy (p, s.p);}
// Programul apelant
void  main (void)
{       SIR s1("Primul text "), s2("si al doilea text");
        printf("\n s1 = %s\n", s1);
        printf(" s2 = %s\n", s2);
        printf(" Valoarea lui s1 = %p\n", s1);
        printf(" Valoarea lui s2 = %p\n", s2);
        s1 = s2;                        // Se comandă copierea conținutului obiectului s2 în s1
        printf("\n s1 = %s\n", s1);
        printf(" s2 = %s\n", s2);
        printf(" Valoarea lui s1 = %p\n", s1);
        printf(" Valoarea lui s2 = %p\n", s2);
}
```

În acest exemplu se redefinește operatorul de atribuire =. Astfel, funcția **void operator =** (SIR &) conține instrucțiunea strcpy (p, s.p); care realizează copierea conținutului obiectului s2 în obiectul s1. Valorile pointerilor s1 și s2 rămân nemodificate. Acest lucru se vede prin execuția programului care va afișa mesajele:

```
        s1 = Primul text
        s2 = si al doilea text
        Valoarea lui s1 = 1E88 (o valoare în hexazecimal)
        Valoarea lui s2 = 1EF0
```

După comanda solicitată de noi:

```
        s1 = si al doilea text
        s2 = si al doilea text
        Valoarea lui s1 = 1E88
        Valoarea lui s2 = 1EF0
```

În finalul executării funcției main(), destructorul clasei ~SIR va elibera ambele zone de memorie alocate celor două obiecte. De data aceasta, rezultatele obținute sunt corecte.

Observație. Instrucțiunea s1 = s2; este echivalentă cu instrucțiunea s1.**operator =** (s2);

6.5.2. Operatorul de indexare, []

Un exemplu de utilizare a operatorului de indexare a fost dat în cazul clasei VECT. În exemplul următor ne propunem extragerea valorii elementului [i] dintr-un vector utilizând funcția **operator []**.

```
// Program P6_7.CPP  Acces la obiectele clasei prin intermediul funcției operator []
# include <string.h>
# include <iostream.h>
# include <limits.h>
const char BLANC = 0x20;
class SIR {
        unsigned int  l;
        char text [UCHAR_MAX + 1];
public:
        SIR (short marime, char *q);          // Constructorul
        char & operator [] (int);             // Declarația funcției membru
};

SIR :: SIR (short marime, char *q)            // Implementarea constructorului
{       if (marime > UCHAR_MAX)
                marime = UCHAR_MAX;           // Lungimea șirului se limitează la UCHAR_MAX
                                              // caractere
        l = marime;
        strncpy(text, q, l);                  // Se copiază l caractere din vectorul q, într-un vector
        text[l] = '\0';                       // care începe la adresa text
}

char & SIR :: operator[] (int i)              // Implementarea funcției membru
{       if (i >= 0 && i < l)
          {     return text[i];
                cout << text[i];
          }
        else {
                cerr << "Indice in afara limitelor, adica " << "i > " << l << endl;
        }
}
// Programul apelant
void  main (void)
{
        SIR s(10, "1234567890");                  // marime =10
        cout << "s[9] = " << s[9] << endl;        // Se afișează s[9] = 0
        s[9] = BLANC;
        cout << "s[9] = " << s[9] << endl;        // Se afișează s[9] =
        char c = s[13];
        cout << "s[13] = " << c << endl;          // Indice in afara limitelor, adica 13 > 10
}
```

Clasa denumită SIR definită în acest exemplu conține ca date de tip privat lungimea efectivă a șirului notată l și un masiv, denumit text, de maximum 256 caractere (Într-adevăr, constanta UCHAR_MAX definită în <**limits.h**> are valoarea 255). Se vede că în funcția de tipul operator

char & operator[] (int) apare și operatorul de referință &. Această funcție preia ca argument de intrare de tip **int**, indexul i, și după o verificare prealabilă a încadrării lui între limitele 0 și l, întoarce o referință la caracterul din șirul punctat de indexul i.
În funcția main() se creează mai întâi obiectul s de 10 octeți și se inițializează cu valorile precizate. Prin instrucțiunea s[9] = BLANC; se apelează de fapt funcția de tip **operator []**, prin care se înscrie valoarea 0x20 (corespunzătoare blancului) în al zecelea octet al obiectului s. Instrucțiunea este echivalentă cu s[9] = ` `;
Instrucțiunea **char** c = s[13]; conduce la folosirea funcției **& operator []** care va încerca să preia valoarea caracterului cu indicele i = 13. Cum indicele i este în afara gamei admise pentru acest obiect [0, 9], se va da controlul ramurii **else** a funcției.
Cele două instrucțiuni de mai sus se traduc de fapt astfel:

s.**operator []** (9) = " ";
char c = s. **operator []** (13);

Operatorul de atribuire este de tipul binar. Unul din operanzii expresiei s[i] este indicele i, iar celălalt, care este ascuns, este chiar obiectul s (acest obiect). În acest exemplu s-a recurs la operatorul **&** cu scopul de a putea întrebuința funcția **operator []** în cele două modalități posibile, adică: a[i] = valoare; și valoare = a[j];
Restricția ce se impune operatorului de indexare [] este aceea că funcția care îl redefinește nu poate fi decât funcție membră a unei clase.

6.5.3. Operatorul de apel, ()

Precizăm că și în cazul redefinirii operatorului de apel () a unei funcții există aceeași restricție ca și în contextul redefinirii operatorului de indexare [], deoarece și acesta este tot binar. Apelul f(i) se va traduce prin notația f.**operator ()** (i). O aplicație tipică de redefinire a acestui operator, la care ne vom referi în continuare, este aceea a utilizării funcției de tip "iterator" al obiectelor unei clase (*iterator function*). Precizăm că o funcție de tipul iterator nu posedă o sintaxă specială în raport cu celelalte funcții și nu este specifică numai acestui limbaj. Scopul ei este de ascundere a detaliilor de implementare a obiectelor unei clase. Există și o clasă de tip iterrator (*iterator class*). O utilizare tipică a funcției și mai ales a clasei de tip iterator este și aceea referitoare la structurile cunoscute: liste, arbori, stive. Funcțiile care tratează elementele (nodurile) acestor structuri vor trebui să ascundă în ele o serie de detalii de implementare referitoare la obiectele clasei care operează cu aceste structuri. Pe utilizatorul unei astfel de clase, la un moment dat, nu trebuie să-l mai intereseze aceste detalii. El dorește să utilizeze clasa în scopul pentru care a fost creată, neavând neapărat în vedere cum se modifică punctatorii la adăugarea sau eliminarea unui element. Pentru a face posibilă reutilizarea codului pe structuri noi, se recurge la clase de tip iterator care conțin funcții de tipul iterator. Pentru exemplificare, vom pleca de la ultimul listing de mai sus și ne propunem să construim obiecte folosind o clasă separată SIR_IT, de tipul iterator. Listingul este următorul:

```
// Program P6_8.CPP   Acces la obiectele clasei prin intermediul clasei iterator SIR_IT
# include <string.h>
# include <iostream.h>
# include <limits.h>
const char BLANC = 0x20;
class SIR {
        unsigned int  l;
        int nr_biti;
        char *p;
public:
```

```
        SIR (short marime, char *q, int talie = 8);          // Constructorul
        ~SIR() {delete p;}                                   // Destructorul
        friend class SIR_IT;          // La această clasă are acces o clasă de tip iterator SIR_IT
        char & operator [] (int);     // Declarația funcției membru de tipul operator de indexare
};
// Declarația clasei SIR_IT
class SIR_IT {
        int poz_bit;                  // Poziția bitului curent
        SIR *str;                     // în șirul punctat de str cu alura obiectelor SIR
public:
        SIR_IT (SIR &s) {             // Constructorul va anula în mod automat variabila poz_bit,
                                      // adică primul bit din primul caracter punctat prin p
                poz_bit = 0;
                str = &s;
        }
        int operator() (void);        // Declararea unei funcții de tipul iterator
};
// Implementarea funcțiilor
// Implementarea constructorului din clasa de bază SIR
SIR :: SIR (short marime, char *q, int talie)
{       if (marime > UCHAR_MAX)
                marime = UCHAR_MAX;
        l = marime;
        nr_biti = talie;
        p = new char[l+1];            // +1 pentru caracterul '\0'
        strncpy(p, q, l);             // Se copiază l caractere din vectorul q, începând de la
        *(p+l) = '\0';                // adresa p
}
// Implementarea funcției de tipul iterator, membră a clasei SIR_IT
// Aceasta funcție întoarce valoarea bitului curent, adică valoarea 1,
// dacă bitul = 1, nulă, dacă bitul = 0, și -1 la ajungerea la sfârșitul șirului
int SIR_IT :: operator() (void)
{
        int bit, byte;
        byte = poz_bit / str->nr_biti;
        bit = poz_bit % str->nr_biti;
        if (byte >= str->l)
                return -1;            // Sfârșit de șir
        else {
                bit = *(str->p + byte) & (0x80 >> bit);
                poz_bit ++;
           return ((bit > 0) ? 1 : 0);
        }
}
// Programul apelant
void  main (void)
{
        SIR pattern(3, "abc");        // Șirul pattern conține trei elemente: a, b, c
```

```
        SIR_IT bit_cu_bit (pattern);   // Se formează obiectul de tip iterator bit_cu_bit
        int b;
        cout << endl;
        while ((b = bit_cu_bit()) != -1)
                cout << "<" << b << ">" ;
        cout << endl;
}
```

Scopul clasei SIR_IT constă în "palparea" bit cu bit a variabilei private punctate prin intermediul pointerului p. Acesta punctează un șir de elemente, fiecare având opt biți (nr_biți = 8). Clasa SIR_IT conține declarația și definiția (implementarea) funcției **operator ()**. Fiind membră a clasei SIR_IT, aceasta este o funcție de tipul iterator. Funcția întoarce valoarea bitului curent prin intermediul instrucțiunii **return** ((bit > 0) ? 1 : 0); În secțiunea privată a clasei se definește și punctatorul de tip SIR notat cu **str**. Variabilele byte și bit vor conține elementul șirului (un element are nr_biți) și deplasarea (offsetul) în cadrul elementului. Aflarea valorii bitului situat la offsetul bit se realizează printr-o operație de mascare. Masca este constanta 0x80 (adică 10000000b) după plasarea lui 1 glisabil pe bitul care ne interesează. Operația este efectuată de expresia 0x80 >> bit.
În funcția main() se creează obiectul pattern care conține elementele abc adică 0x61, 0x62, 0x63 în hexazecimal. În urma execuției acestui program se va obține următorul șir de biți: 0110 0001 0110 0010 0110 0011 ce corespunde valorilor din obiectul pattern.

6.6. Conversii implicite și explicite

De multe ori, în limbajul C++ au loc, în mod transparent, conversii implicite. De exemplu, în ultimul program de mai sus, variabila l are tipul **unsigned short**, iar variabila marime din constructor are tipul **int**. La executarea instrucțiunii l = marime; va avea loc o conversie implicită **int** --> **unsigned short**. Conversiile se pot realiza prin:

- funcții de conversie;
- operatori de tip **cast**.

6.6.1. Conversii realizate prin intermediul funcțiilor

De fapt, funcția ce realizează o astfel de conversie este funcția de tip constructor. Pentru exemplificare, considerăm următorul program:

```
// Program P6_9a.CPP   Un prim mod de inițializare a obiectelor
# include <math.h>
# include <iostream.h>
class TINTA {
public:
        double pozitie;         // pozitie = poziția unei ținte (criptată), astfel:
                                // 4 cifre      - distanța
                                // 5            - unghiul de înălțare theta
                                // 5            - azimutul phi
        TINTA (unsigned int dd, float ii, float aa);  // Constructorii clasei
        TINTA (double diaz) { pozitie = diaz; }
};
// Implementarea constructorului (decriptorului)
TINTA :: TINTA (unsigned int dd, float ii, float aa)
{
        double const d = 10000000000.;
        double const i = 10000000.;
```

```
        double const a = 100.;
        pozitie = dd*d + ii*i + aa*a;
        pozitie = floor(pozitie);        // Funcția floor() întoarce cel mai mare întreg, mai mic decât
}                                        // numărul precizat
// Programul apelant
void  main (void)
{
  TINTA ufo1(100, 40.12, 270.34), ufo2(1500413217123.l);
  cout << "Pozitia este: " << ufo1.pozitie << ", respectiv  " << ufo2.pozitie << endl;
}
```

Acest program conține doi constructori. În cazul obiectului ufo1, datele celor trei coordonate activează constructorul care are în lista sa de intrare trei parametri. Vor avea loc totuși conversii "subterane" și anume:

100 (**signed int**) trece în 100 (**unsigned int**);
dd (**unsigned int**) trece în **double**;
ii și aa (**float**) trec în **double**.

În cazul obiectului ufo2, mesajul criptic 1500413217123.l este supus conversiei din tipul **double** în tipul TINTA.

6.6.2. Conversiile de tipul cast

Modificăm exemplul de mai sus, în sensul introducerii în cadrul clasei TINTA a unei funcții de tipul operator **cast**, în care **cast** devine **double ()**.

```
// Program P6_9b.CPP   Un alt mod de inițializare a obiectelor
# include <math.h>
# include <iostream.h>
# include <stdio.h>
class TINTA {
        double pozitie;         // pozitie = poziția unei ținte (criptată), astfel:
                                // 4 cifre      - distanța
                                // 5            - unghiul de înălțare theta
                                // 5            - azimutul phi
public:
        TINTA (unsigned int dd, float ii, float aa);  // Constructorii clasei
        TINTA (double diaz) { pozitie = diaz; }
        operator double() { return pozitie; }
};
// Implementarea constructorului (decriptorului)
TINTA :: TINTA (unsigned int dd, float ii, float aa)
{       double const d = 10000000000.;
        double const i = 10000000.;
        double const a = 100.;
        pozitie = dd*d + ii*i + aa*a;
        pozitie = floor (pozitie);
}
// Programul apelant
void  main (void)
{       TINTA ufo1(100, 40.12, 270.34), ufo2(1500413217123.l);
        double poz1, poz2;
```

```
        poz1 = double (ufo1);
        poz2 = double (ufo2);
        printf("Pozitiile tintelor: ufo1 = %2.14g , respectiv ufo2 = %2.14g\n", poz1, poz2);
}
```

În funcția main() s-au folosit instrucțiuni de forma:

```
        poz1 = double (ufo1);
        poz2 = double (ufo2);
```

îc care apare funcția operator **double()** și care realizează o conversie a obiectului de tip TINTA în tipul **double**.

În limbajul C, operațiile de conversie utilizând operatorul **cast**, se utilizează sub forma:

```
        int i;
        double  d, e:
        d = (double) i;
        e = (double) (4/10);
```

În C++, utilizând operatorul **cast**, ultima conversie se scrie astfel e = **double** (4/10). (*Observație.* (**double**) 4/10 este diferit de expresia (**double**) (4/10). Aceasta pentru că *operatorul* **cast** *are prioritatea mai mare decât operatorul* "/"). De reținut este faptul că notația **double** (4/10) este un apel de funcții de tip **cast**. Aceste exemple sunt plasate în listingul de mai jos:

```
// Program P6_9c.CPP   Utilizarea operatorului "cast"
# include <stdio.h>
void  main (void)
{
        double num1, num2, num3;
        int i = 7;
        num1 = double (i/10);
        num2 = (double) i/10;    // Operatorul cast are prioritate superioara fata de operatorul "/"
        num3 = (double) (i/10);
        printf("num1 = %lf,  num2 = %lf, num3 = %lf\n", num1, num2, num3);
        float j = 7;
        num1 = double (j/10);
        num2 = (double) j/10;    // Operatorul cast are prioritate superioara fata de operatorul "/"
        num3 = (double) (j/10);
        printf("num1 = %lf,  num2 = %lf, num3 = %lf\n", num1, num2, num3);
}
```