Programare in C++, POO

3. SUPRADEFINIREA OPERATORILOR

3.1 Operatori ca functii ale clasei

Definirea clasei "Byte" poate fi privita ca o extindere a limbajului cu un nou tip de date. Ideal ar fi ca noile tipuri de date definite prin clase sa aiba o utilizare identica sau cat mai apropiata de cea a tipurilor predefinite.

Exprimarea operatiei de adunare dintre doua numere "Byte" sau dintre un intreg si un "Byte" printr-o functie cu numele "plus" sau "add" este mult diferita de exprimarea adunarii dintre doua numere intregi (prin operatorul '+'). De aceea s-a introdus in limbajul C++ posibilitatea de a "supraincarca" si operatorii limbajului cu noi sensuri, legate de noile tipuri de date definite prin clase.

Ideea nu este total noua, daca ne gandim ca o serie de operatori din limbajul C (si din limbajul Pascal) au deja mai multe utilizari diferite, in functie de tipul operanzilor. De exemplu, operatorul '+' are in Pascal sensurile de : "adunare de numere", "concatenare de siruri", "reuniune de multimi". Ceea ce apare nou in C++ este ca si utilizatorii pot adauga alte sensuri operatorilor existenti (dar nu pot adauga noi operatori).

Un operator este considerat in C++ ca o functie cu un nume special, dar supus tuturor regulilor referitoare la functii. Numele unei functii operator consta din cuvantul "operator" urmat de unul sau doua caractere speciale, prin care se foloseste operatorul.

Exemplul urmator arata cum se poate supradefini operatorul '+' pentru adunarea de obiecte de tip "Byte".

```
 // operatorul "+" supradefinit
class Byte {
 int b;
public:
 Byte (int v=0) { assert (v < 255); b=v; }
 Byte & operator + (Byte x) { b=x.b+b; assert (b < 256); return *this;}
 Byte & operator + (int k) { b=b+ k; assert (b < 256); return *this;}
 void print () { cout << b << endl; }
};

void main () {
 Byte b1=1, b2=2,b3=3,b4;
  b1=b1 + 5;  b1.print() ;   // 6
  b2=b2 + b1; b2.print() ;   // 8
  (b1+b2).print();       // 14
  b4=b1+b2+b3; b4.print();   // 25
}
```

Expresia

```
b1 = b1 + b2;
```

este o forma alternativa (si preferabila) pentru expresia

```
b1 = b1.operator + (b2);
```

Aceasta forma de (supra)definire a operatorului binar de adunare nu este si cea mai buna, din urmatoarele motive:

- Primul operand este intotdeauna modificat, ceea ce nu se intampla la adunarea de numere de un tip aritmetic predefinit.
- Primul operand trebuie sa fie de tip Byte, deci expresia (7 + b1) nu este permisa. Un operator definit ca functie membru nu are proprietatea de comutativitate, dar exista posibilitatea de a supradefini operatori comutativi prin functii "prieten".

Operatorul binar '+' putea avea si rezultat de tip "Byte", dar in nici un caz de tip "void". Rezultatul operatorului este un obiect temporar care poate fi folosit in alte expresii. Exemple:

```
cout << b1 + b2;
b4= b1+b2+b3;
```

Definitia operatorului "+" cu operator de tip "int" este inutila si poate lipsi deoarece in expresia urmatoare

```
b2= b1 + 7;
```

compilatorul incearca o conversie automata a constantei 7 la tipul "Byte" folosind constructorul clasei; deci ar fi fost ca si cum s-ar fi scris:

```
b2 = b1 + Byte(7);
```

Rezultatul unei functii operator poate fi de orice tip, diferit de "void". De multe ori, rezultatul unui operator este o referinta la clasa din care face parte.

Expresia *this desemneaza chiar obiectul pentru care se executa functia membru. Orice apel de functie dintr-o clasa primeste ca parametru implicit adresa obiectului. O functie operator '+' se poate rescrie si astfel:

```
Byte & Byte:: operator + (Byte * this, int k) {
   this->val= this->val + k;
   return *this;
 }
```

Exemplul urmator arata cum se poate supradefini operatorul "+=" de adunare si atribuire:

```
 Byte & Byte:: operator += (Byte b) {
    val = val + b.val;  // val += val.b
    return *this;
 }
```

Definirea unui operator de incrementare pentru clasa Byte se poate face astfel:

```
Byte & Byte:: operator ++ () {
     ++val;
     return *this;
 }
```

Utilizarea acestui operator este numai inaintea operandului:

```
Byte b1=1;
cout << ++b1 << endl;
```

Operatorii unari definiti prin functii membre nu au nici un parametru, iar operatorii binari au un singur parametru.

Supradefinirea operatorilor prin functii ale clasei se recomanda pentru operatorii unari care modifica valoarea operandului, pentru operatorii binari care modifica primul operand (atribuire simpla sau combinata cu alti operatori) si pentru operatori de comparatie.

Exemplul urmator arata cum se pot supradefini operatori de comparatie pentru obiecte de tip Byte:

```
    // comparatie la mai mic
   int Byte::operator < (Byte b) {
     return val < b.val;
   }
   // comparatie la inegalitate
  int Byte::operator != ( Byte b) {
    return val != b.val;
  }
     // utilizare
   ...
   Byte b1, b2;
   if ( b1 != b2) cout << " diferite" << endl;
   if ( b1 < b2) cout << b1 << "<" << b2 << endl;
```

Aproape toti operatorii C pot fi supradefiniti, inclusiv operatori unari (++, -- de exemplu) si unii operatori mai speciali cum sunt cel de selectare element dintr-un vector [], operatorul functional (), operatorul pentru conversie de tip ("cast"), operatorii "new" si "delete".

Exemplul urmator arata cum se poate defini un operator "int" pentru conversie de la clasa "Byte" la tipul "int" si cum se poate folosi acest operator pentru afisarea unor obiecte de tip "Byte".

```
  // supradefinire operator "cast"
class Byte {
 int i;
public:
 Byte (int v=0) { assert (v < 255); i=v; }
 operator int () { return i;}
};

void main () {
 Byte x1=5, x2, x3(16); int x;
  cin >> x;   // nu  cin >> (int) x;
```

```
  x2=x1+6.5;  // x2 = Byte ((int)x1 + (int)6.5)
  cout << "x2=" << x2 << endl;
  cout << sqrt(x3);  // sqrt ( (int)x3 );
}
```

3.2 Supradefinirea operatorului de atribuire

Cel mai bun exemplu de operator definit printr-o functie membru este cel al operatorului de atribuire, care modifica valoarea primului operand. Supradefinirea operatorului de atribuire este necesara pentru clasele ce contin pointeri (sau contin obiecte cu pointeri), pentru a inlocui copierea "superficiala" ("shallow copy") realizata de operatorul de atribuire generat automat de compilator cu o copiere "profunda" ("deep copy").

Supradefinirea operatorului de atribuire este necesara pentru aceleasi clase in care este necesara si supradefinirea constructorului de copiere.

In cazul unui vector alocat dinamic sau a unui sir de caractere (care este tot un vector, de caractere), copierea superficiala inseamna copierea adresei dintr-un obiect in altul, iar copierea profunda inseamna copierea datelor la o alta adresa in noul obiect. Altfel spus, prin copiere profunda obtinem obiecte diferite ce contin adrese diferite, in timp ce copierea superficiala conduce la situatia unor obiecte diferite care contin o aceeasi adresa.

Exemplul urmator arata o astfel de atribuire:

```
String s1("unu"), s2("doi");
s1= s2;
 // efecte nedorite ale atribuirii
cout << s1 ;  // se afiseaza "doi"
{ String s3 ("3");  s1 = s3 ; }    // la iesire se distruge s3
  // la distrugerea lui s1 apare eroare de pointer nul
```

Dupa aceasta atribuire, s1 si s2 contin adresa sirului "doi", iar orice modificare in acest sir afecteaza obiectul s1. Sirul "unu" devine inaccesibil si nu mai poate fi sters din memorie, pentru ca s-a pierdut adresa lui. Solutia acestei probleme este sa se aloce memorie pentru obiectul creat (prin atribuire sau prin construire) si sa se copieze datele din obiectul initial la noua adresa.

```
// operator de atribuire pentru clasa Vector
class Vector {
     int * vec;          // adresa vector alocat dinamic
  int d, dmax;       // dimensiune vector
public:
     Vector (int max=10) {
    dmax=max; vec=new int [dmax];
  }
  ~Vector () { delete [dmax] vec; }
  Vector & operator = (Vector & v) {
    if (*this != v) {    // evita operatii inutile
      if (dmax != v.dmax) {
        delete [dmax] vec;
        dmax=v.dmax;
        vec= new int [dmax];
      }
      d=v.d;  // numar de elemente in vector
```

```
        for (int i=0;i<d;i++)
          vec[i]=v.vec[i];
      }
      return * this;
    }
```

3.3 Supradefinirea operatorilor prin functii prieten

Pentru ca un operator sa poata avea ca prim operand o expresie de un alt tip decat tipul clasei, este necesar ca functia operator sa fie definita (si supradefinita) cu doi operanzi, al caror tip se specifica explicit. Deci o astfel de functie nu poate fi o functie membru, care primeste implicit si nu explicit unul din operanzi. Daca datele clasei sunt publice sau clasa contine metode de acces la datele "private", atunci se pot folosi functii externe (nemembre) pentru supradefinirea unor operatori. Exemplu:

```
    // alta definitie a operatorului de egalitate
  int operator == (Byte b1, Byte b2) {
    return ( b1.getVal() == b2.getVal() );
  }
    // utilizare
  ...
  if (b1==b2) cout << "egale" << endl;
```

Daca datele clasei nu sunt publice (conform principiului incapsularii datelor) atunci o functie externa clasei nu are acces la aceste date.

Solutia acestei dileme este o functie "prieten", adica o functie externa, dar cu drept de acces la datele locale ale unei clase; aceste functii trebuie declarate prin cuvantul cheie "friend" in clasa care le acorda drepturi. Utilitatea functiilor "prieten" nu se reduce la supradefinirea unor operatori binari, dar nu trebuie abuzat de ele pentru ca reprezinta o bresa in sistemul de protectie a datelor dintr-o clasa fata de modificari din afara clasei.

Exemplul urmator arata cum se pot defini mai multi operatori de adunare prin functii prieten si posibilitatile create pentru expresii permise cu acesti operatori.

```
   // operatori ca functii prieten
class Byte {
 int val;
public:
 Byte (int v=0) { assert (v < 255); val=v; }
 operator int () { return val;}
 friend Byte operator + (Byte ,Byte );
 friend Byte operator + (int, Byte );
 friend Byte operator + (Byte,int );
};

Byte operator + (Byte b1, Byte b2) {
 Byte b ;
 b.val=b1.val+b2.val;
 assert ( b.val > 0 && b.val < 256);  return b;
}

Byte operator + ( int k, Byte b2) {
 Byte b ;
```

```
 b.val=k+b2.val;
 assert ( b.val > 0 && b.val < 256); return b;
}

Byte operator + ( Byte b1, int k) {
 Byte b ;
 b.val=b1.val + k;
 assert ( b.val > 0 && b.val < 256); return b;
}

void main () {
 Byte b1(1), b2(2), b3;
  b3= b1 + b2; cout << b3 << ' ';  // b1 si b2 nu se modifica
  b2= b2 + 3;  cout << b2 << ' ';
  b1= 5 + b1;  cout <<  b1 << ' ';
}
```

Utilizarea de functii prietene sau de functii externe pentru operatori se recomanda pentru:

- operatori binari care nu trebuie sa modifice operanzii (operatori de comparatie, de exemplu);
- operatori binari care produc un rezultat, diferit de cei doi operanzi (ramasi nemodificati) : operatori aritmetici, logici, s.a.
- operatori de citire-scriere (>> si <<).

3.4 Supradefinirea operatorilor de citire-scriere

Pentru uniformitatea operatiilor de citire si de scriere a datelor de orice tip este preferabil sa putem folosi operatorii << si >> pentru obiecte.

Acest lucru este posibil prin "supraincarcarea" acestor operatori cu noi sensuri.

Problema este ca primul operand este neaparat un obiect de tipul "istream" (de obicei, obiectul "cin") sau de tipul "ostream" (pentru consola, obiectul "cout"), deci functiile pentru acesti operatori nu pot fi membre ale altor clase (de exemplu, ale clasei "Byte"). Nici modificarea definitiilor claselor "istream" si "ostream" de catre utilizatori, prin adaugarea de noi functii, nu este acceptabila.

Solutia foloseste, din nou, doua functii "prieten" cu clasa "Byte" pentru supradefinirea operatorilor << si >> ; primul parametru va fi de tip referinta la clasa "ostream" sau "istream", iar al doilea parametru va fi de tip "Byte" sau "referinta la Byte".

Clasele "ostream" si "istream" sunt definite in fisierul IOSTREAM.H. Exemplu:

```
  // operatori de citire / scriere la consola
class Byte {
 int val;
public:
 Byte (int v=0) { assert (v < 255); val=v; }
 operator int () { return val;}
 friend istream & operator >> (istream& ,Byte& );
 friend ostream & operator << (ostream&, Byte&);
```

```
};

istream & operator >> (istream& is, Byte& b) {
 int x;
 is >> x;
 assert ( x >= 0 && x < 256);  b.val=x;
 return is;
}

ostream & operator << ( ostream& os, Byte& b) {
 return os << b.val;
}

void main () {
 Byte b,c;
 do {
  cin >> b; cout << b << ' ';
  } while (cin);
 cin >> b >> c ;
 cout << b << c << endl;
}
```

Uneori operatorii '<<' si '>>' se pot defini si prin functii externe oarecare ("ne-prieten"), daca exista alte functii care permit accesul la datele clasei. Exemplu:

```
  // operatori ca functii externe
class Byte {
 int i;
public:
 Byte (int v=0) { assert (v < 255); i=v; }
 operator int () { return i;}
};

istream & operator >> (istream& is, Byte& b) {
 int x;
 is >> x;
 assert ( x >= 0 && x < 256);  b= Byte(x);
 return is;
}

ostream & operator << ( ostream& os, Byte & b) {
 return os << (int) b;
}

void main () {
 Byte b;
 do {
  cin >> b; cout << b << ' ';
  } while (cin);
}
```

Functiile "prieten" pot fi externe claselor sau pot apartine unei clase. Daca o clasa F contine functii ce trebuie sa aiba acces la date dintr-o clasa B, atunci vom defini clasa F ca o clasa 'prieten" cu B.

3.5 Supradefinire operatori [] si ()

Operatorul [] este folosit in limbajul C pentru selectarea unui element dintr-un vector. Expresia t[i] este privita ca o expresie cu un operator si doi operanzi: t si i.

Definit ca functie membru, acest operator are ca parametru un intreg ce reprezinta pozitia elementului in vector si ca rezultat valoarea elementului (de tipul componentelor vectorului). Pentru ca valoarea selectata din vector sa poata fi modificata (sa poata apare in stanga unei atribuiri) este necesar ca rezultatul sa fie de tip referinta la tipul componentelor.

Exemplul urmator arata cum se poate defini si utiliza operatorul [] pentru a face si verificarea incadrarii indicelui in limitele permise.

```
    // supradefinire []
  class Vector {
    int * vec;         // adresa vector alocat dinamic
    int d, dmax;       // dimensiune vector
  public:
    Vector (int max=10) {
      dmax=max; vec=new int [dmax];
    }
    ~Vector () { delete [dmax] vec; }
    int& operator [] (int i) { assert (i>=0 && i<d); return vec[i]; }
  };
```

Faptul ca operatorul [] are un rezultat de tip referinta ne permite sa folosim o expresie de forma t[i] si in partea stanga a unei atribuiri, nu numai in partea dreapta a operatorului de atribuire. Exemplu:

```
    // utilizare operator []
  void main () {
    Vector a(3), b(3);
    for (int i=0;i<3;i++) [
      a[i] = 10*i;
      b[i]=a[i];
      cout << b[i] << " ";
    }
  }
```

Pentru a separa cele doua utilizari ale operatorului [] putem declara doi operatori diferiti astfel:

```
... // in clasa Vector
  int & operator [] (int i) { return vec[i];}
  const int & operator [] (int i) const { return vec[i];}
  ...  // alte metode
```

O functie cu un parametru formal "const" va putea primi acum un element dintr-un vector:

```
    // o functie cu parametru nemodificabil
  void print ( const int x ) {
    cout << x << " ";
  }
    // utilizare
  Vector a(3);
  print ( a[0]);
```

Functia operator [] nu poate avea decat un singur argument si nu se poate folosi pentru a selecta un element dintr-o matrice, decat daca matricea este un obiect "Vector" cu componente de

tip "Vector".

Exista insa un alt operator in C care poate avea mai multe argumente, separate prin virgule, si care produce o valoare.

Operatorul () este folosit in C pentru apelarea de functii, deci pentru a produce o valoare pe baza mai multor valori (argumentele efective). De exemplu, f(i,j) va produce o valoare de tipul functiei "f".

Redefinirea operatorului () in C++ este utila atat la definirea unor clase de tip matrice (cu doua sau mai multe dimensiuni), cat si la definirea unor iteratori, care produc valoarea urmatoare dintr-o colectie.

Vom exemplifica aici numai folosirea operatorului () intr-o clasa "Mat" cu o matrice alocata dinamic, desi selectarea unui element din matrice corespunde notatiei din Fortran si Basic si nu notatiei din C.

```
class mat {
protected:
 double ** m;
 int nl, nc;
public:
 mat (int nli,int nci, double vali=0);
 ~mat ();
 double & operator () (int lin, int col);
};

double & mat :: operator () (int lin, int col) {
  assert (lin < nl && col < nc);
    return ( m[lin][col]);
}

mat:: mat (int nli,int nci, double vali) {
 m= new double * [nli];
 for (int x=0; x <nli;x++)
   m[x] = new double [nci];
 nl=nli; nc = nci;
 for (int i=0; i<nl; i++)
   for (int j=0; j <nc; j++)
     m[i][j] = vali;
}

mat:: ~mat () {
  for (int x=0; x < nl; x++)
    delete m[x];
  delete m;
}

void main () {
 const int n=3;
 int i,j ;
 mat A (n,n);
 for ( i=0;i<n;i++)
  for ( j=0; j<n; j++)
    A(i,j) = j + i* n + 1;
}
```