8 Funcții de administrare (II)

8.1 Supraîncărcarea operatorilor

Pentru a uşura sintaxa de utilizare a claselor, putem implementa operațiile nu prin funcții membre, ci prin supraîncărcarea unor operatori. Acest lucru ar putea face posibilă scrierea următorului tip de cod:

Exemplu:

```
String InBuf;
...
while ( cin >> InBuf )
{
   if( !Inbuf ) return;
   if (InBuf == "gata" ) return;
   // prelucrare
   cout << "Sirul este :" << InBuf;
}
...
```

Un operator nu trebuie neapărat să fie membru, însă în acest caz trebuie să aibă cel putin un argument de tip clasă. Un operator se defineşte ca şi orice altă funcție având în locul numelui cuvântul `operator` urmat de unul din operatorii predefiniți posibil de supraîncărcat. Aceşti operatori sunt:

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

Exemplu:

```
class String
{
   public:
      String& operator = ( const String& );
      String& operator = ( const char* );
      int operator == ( String& );
...
};

String& String::operator = (const String& s)
{
   // asignare siruri
   len = s.len;
   delete str;     // elibereaza zona alocata
   str = new char[len+1];
   strcpy( str, s.str );
   return *this;
}

String::operator == ( String& s )
{
   // returneaza 1 daca sunt egale
   return( strcmp( str, s.str ) == 0 );
}
```

Observații:

- operatorii tipurilor predefinite nu pot fi modificați (nu supraîncărcare, nu adăugare alții noi);

- prin supraîncărcarea operatorilor se păstreaza *aritatea* acestora (binar, unar).

Un operator poate fi definit fie ca membru, fie ca nemebru al clasei. Prin convenție, primul operand (operandul stang) al unui operator membru este obiectul invocator.

Exemplu:

```
class String
{
   friend String& operator +( String&, String& );
...
};
```

sau

```
class String
{
   public: String& operator +( String& );
...
};
```

Cazul în care pentru o clasă sunt definite ambele instanțe, poate genera situații ambigue.

```
class String
{
   friend String& operator +( String&, String& );
   public:
      String& operator +( String& );
...
};
String a("hobby"), b("computer");
String c = a+b;         // ambiguitate
```

Observație:

- sunt patru operatori care trebuie definiți ca membri:
  asignarea `=`, indexarea `[]`, iterare `()`, selectorul `->`.

Dacă operandul stâng nu este obiectul invocator, atunci operatorul trebuie definit ca functie nemembră. În plus, dacă accesează membri nepublici, operatorul respectiv trebuie declarat `friend`.

Exemplu:

```
class String
{
   friend ostream& operator << (ostream& os, String &s)
   {
      return (os<<s.str);
   }
};
```

Utilizatorii clasei `String` trebuie să aibă acces de citire şi scriere la caractere individuale ale membrului `str`. Cu alte cuvinte, în utilizarea obiectelor de tip `String` trebuie să existe suport pentru următorul tip de cod:

Exemplu:

```
String propozitie("Astazi este miercuri.");
String tempBuf(propozitie.getLen() );
for( int i=0; i<propozitie.getLen(); ++i )
   tempBuf[i]=propozitie[i];
```

`String::getLen` este o simplă funcție de acces care returnează lungimea obiectelor de tip `String`:

```
inline String::getLen()
{
   return len;
}
```

Deci operatorul de indexare `[]` trebuie să poată apărea de ambele părți ale unei atribuiri. Ca să poata apărea în stânga, valoarea returnată de el trebuie să fie o l-value. Din acest motiv trebuie să specificăm că tipul valorii returnate este o referință.

Exemplu:

```
char& String::operator[](int elem)
{
   checkBounds(elem);
   return str[elem];
}
```

unde `checkBounds` verifică încadrarea indicelui în domeniul admisibil. Astfel, valoarea returnată de operatorul `[]` este o referință la elementul indicat şi va putea apărea ca destinația unei asignări.

Exemplu:

```
String st("mov");
st[0]='M'; // se asigneaza 'M' elementului de pe pozitia 0 a membrului
           // st.str
```

Operatorul de iterare `()` permite utilizatorului să parcurgă elementele unei clase. La fiecare invocare returnează "următorul" element, până la epuizarea acestora. Următorul fragment de program ilustrează cum ar putea fi folosit un iterator pentru clasă `String`.

Exemplu:

```
String InBuf;
while( cin>>InBuf) // se citeste un obiect String
{
   char ch;
   while( ch=InBuf() ) // se itereaza elementele din InBuf
...
}
```

Această secvență necesită definirea operatorului nemembru `>>` şi a operatorului membru `String::operator ()()`.

Exemplu:

Pentru citirea obiectelor de tip `String`, operatorul de citire poate fi supraîncărcat astfel:

```
istream& operator >> (istream& is, String& s )
{
   char Buf[255];  // Buf[StringSize]
   is >> Buf;
   s=Buf;          // String::operator=(char *)
   return is;
}

String& String::operator=(const char *s)
{
   // operatorul permite asignarea unui sir de caractere unui obiect
   // de tip String
   len=strlen(s);
   delete str;
   str=new char[len+1];
   strcpy(str,s);
   return *this;
}
```

Prin definirea unui operator de iterare `()` se permite parcurgerea elementelor unui tip. După ce toate elementele au fost parcurse, iteratorul returnează `0`. Acest lucru permite construcții de tipul:

Exemplu:

```
while( ch=InBuf() )
// ...
```

Pentru implementarea unui iterator pentru clasa `String`, este necesară adăugarea unei noi date membru, `index`. Ea va "indica" următorul element de returnat. Fiecare constructor al clasei `String` trebuie să inițializeze `index` cu `0`.

Exemplu:

```
char String::operator()()
{
   if (index<len)
   return str[index++];
   return (index=0);
}
```

Implicit, alocarea dinamică a unui obiect este rezolvată prin operatorii globali predefiniti `new` şi `delete`. O instanță membră a lui `new` trebuie să returneze tipul `void*` şi să transmită ca argument tipul `size_t` (definit în `<stddef.h>`). Acest argument este automat initializat de compilator cu dimensiunea în octeți a tipului clasei. Când `new` se aplică unui nume de clasă, compilatorul verifică dacă clasa are definită propria instanță a operatorului `new`. Dacă da, aceasta este invocată, dacă nu, se invocă instanța globală.

Exemplu:

```
void* NumeClasa::operator new( size_t size)
```

Dacă `new` e propriu unei clase, el este invocat numai pentru alocări individuale de obiecte, nu şi pentru alocări de tablouri de obiecte de tipul clasei.

Exemplu:

```
StringList *p=new StringList;
   // invoca instanta membra a lui new
StringList *pia=new StringList[10];
   // invoca instanta predefinita(globala) a lui new.
```

Observație:

Definiția explicită a unui obiect, cum ar fi `StringList s`; nu invocă nici una din instanțele lui `new`.

Programatorul poate invoca instanța nemebră a operatorului `new`, prin intermediul operatorului domeniu:

Exemplu:

```
StringList *ps = ::new StringList;
   // invoca operatorul predefinit new
```

Analog,

```
::delete ps;
   // invoca operatorul predefinit delete
```

Operatorul `delete` trebuie să aibă un prim argument de tip `void*`. Poate fi specificat un al doilea argument de tip `size_t`. Daca acest al doilea argument este prezent, el va fi inițializat cu dimenisunea în octeți a obiectului adresat de primul argument. Returnează `void`.

Observație:

Operatorii `new` şi `delete` functionează ca şi nişte funcții membre statice ale clasei lor. Ei nu au pointer `this` şi sunt "interpretați" `static` pentru că sunt apelați înainte de constructorul obiectului (`new`) sau dupa ce obiectul a fost distrus cu destructor (`delete`).

Cum am vazut în capitolele anterioare, atribuirea unui obiect altui obiect de tipul aceleiaşi clase se face implicit, membru cu membru, pentru toți membri nestatici. Compilatorul genereaza o instanță pentru clasa respectivă, de forma:

```
X& X::operator = (const X&);
```

pentru rezolvarea cazurilor de atribuire dintre două obiecte de acelasi tip.

Exemplu:

```
String articol("un");
String comun("exemplu");
```

Atribuirea `comun = articol ;` este rezolvată astfel:

```
String& String::operator = (const String& s)
{
   len=s.len;
   str=s.str;
   index=s.index;
}
```

La acest tip de asignare apar trei probleme:

1) ambele obiecte adresează aceeasi zonă de memorie, deci pot apărea probleme dacă nu au acelaşi domeniu de existenţă;
2) zona alocată pentru `"exemplu"` e definitiv pierdută;
3) am definit un index care permite iterarea unui şir de caractere şi care trebuie să fie `0` la sfârsitul iterării; deci după o copiere `index` trebuie să fie `0`, nu valoarea indexului altui obiect.

Pentru rezolvarea acestor probleme, poate fi adoptată următoarea soluţie:

Exemplu:

```
String& String::operator=(const String& s)
{
   index=0;
   len=s.len;
   delete str;
   str=new char[len+1];
   strcpy(str, s.str);
   return *this;
}
```

Operatorul ->

Selectorul de membri este un operator unar al operandului sau stâng, operand care trebuie să fie sau un obiect, sau o referinţă la un obiect. Valoarea returnată trebuie să fie un pointer la un obiect sau un obiect pentru care s-a definit operatorul `->`. Valorii returnate trebuie să i se poată aplica operatorul predefinit `.` .

Exemplu:

```
class eX
{
   public:
      String* operator->();
...
   private:
      String *ps;
...
};
```

Implementarea operatorului ar putea fi:

```
String *eX::operator->()
{
   if(ps==0)
      // initializare ps
      // prelucrare ps
   return ps;
}
```

Operatorul de selecţie membru poate fi invocat fie pentru un obiect `eX`, fie printr-o referinţă la un obiect `eX`.

Exemplu:

```
void ff(eX x, eX& rx, eX *px)
{
   int i;
   i=x->getLen();  // ok   x.ps->getLen()
```

```
   i=rx->getLen(); // ok   rx.ps->getLen()
   i=px->getLen(); // eroare, nu exista eX::getLen()
}
```

Operatorul de selecție membru nu poate fi invocat printr-un pointer la `eX`, deoarece compilatorul nu poate face distincție între instanța predefinită şi cea supraîncărcată.

8.2. Conversii definite de utilizator

Conversiile standard pentru tipurile de date predefinite previn înmulțirea excesivă a operatorilor şi a funcțiilor supraîncărcate. De exemplu, fără conversiile aritmetice, următoarele 6 adunări ar solicita 6 operatori de adunare:

Exemplu:

```
char ch;
short sh;
int ival;

ch+ival; // solutia problemei consta in promovarea
ch+sh;   // operanzilor la tipul int; astfel va fi
ival+sh; // necesara o singura operatie: cea de
ival+ch; // adunare a 2 numere intregi.
sh+ch;
sh+ival;
```

Aceste conversii sunt făcute implicit de compilator, deci transparente utilizatorului.

În această secțiune vom analiza cum poate proiectantul unei clase să definească un set de conversii utilizator pentru acea clasă. Aceste conversii sunt implicit invocate de către compilator ori de câte ori este nevoie.

Exemplu:

```
#include <iostream.h>
class SmallInt
{
      // clasa de lucru cu intregi mici
      friend istream& operator>>(istream& is, SmallInt& s);
      friend ostream& operator<<(ostream& os, SmallInt& s)
         { return (os<<s.value); }
   public:
      SmallInt( int i=0 ) : value (rangeCheck(i) ) {}
      int operator = (int i) { return (value=rangeCheck(i)); }
      operator int() { return value; }
   private:
      int rangeCheck(int);
      int value;
};

istream& operator>>( istream& is, SmallInt& s)
{
   int i;
   is>>i;
   s=i;    // SmallInt::operator=(int)
   return is;
}

SmallInt::rangeCheck( int i )
{
   if( i<0 || i>255 )
...
   return i;
}
```

În acest mod, `SmallInt` poate fi folosit oriunde poate fi folosit şi tipul `int`.

Constructorul ca operator de conversie

Există situații când putem folosi chiar şi un constructor ca operator de conversie. Constructorii care se pretează la aşa ceva sunt cei cu un singur argument: ei definesc practic setul de conversii de la tipul argumentului la tipul clasei. Conversiile standard, dacă sunt necesare, se aplică înainte de invocarea constructorului.

Exemplu:

Constructorul `SmallInt(int)` serveşte ca operator de conversie a tipului `int` într-un `SmallInt`.

```
extern f(SmallInt);
int i;
f(i);
```

Apelul este rezolvat astfel:

- `i` este convertit la `SmallInt` printr-un apel `SmallInt(int)`

```
{
   SmallInt temp=SmallInt(i);
   f(temp);
}
```

Dacă este necesar, se aplică o conversie standard înaintea invocării `SmallInt(int)`.

Exemplu:

```
double d;
f(d);
```

devine

```
{
   SmallInt temp = SmallInt ( (int) d); // conversie standard
   f(temp);
}
```

Invocarea unui operator de conversie se face numai dacă nici o altă conversie nu este posibilă.

Exemplu:

Dacă funcția `f` ar fi supraîncărcată astfel:

```
f(SmallInt);
f(double);
int ival;
f(ival);   // se potriveste prin conversie standard cu f(double);
           // nu se mai invoca SmallInt(int).
```

Să revenim la operatorii de conversie. Ei sunt funcţii membre speciale şi specifică conversiile implicite ale unui obiect de un anumit tip la un alt tip.

Exemplu:

```
SmallInt::operator unsigned int()
{
   return ( (unsigned) value );
}
```

Operatorii pot fi apelați explicit prin *cast*.

Exemplu:

O clasă poate defini mai mulți operatori de conversie:

```
#include "SmallInt.h"
class Token
{
   public:
      Token (char *n, int v ): val(v), name(n) {}
      operator SmallInt() { return val; }
      operator char* () { return name; }
      operator int() { return val; }
...
```

```
   private:
      SmallInt val;
      char *name;
};
```

Un operator de conversie are forma generală:

```
operator tip ();
```

unde `tip` poate fi orice tip predefinit sau definit de utilizator.

Observații:

1) un operator de conversie trebuie să fie funcție membră;
2) nu trebuie să specifice un tip de valoare returnată;
3) nu poate fi specificată lista de argumente.

Exemplu:

```
operator int (SmallInt&);       // eroare, nu e membru.
class SmallInt
{
   public:
      int operator int();  // eroare, returneaza tip.
      operator int(int);   // eroare, lista de argumente.
...
};
```

Operatorii de conversie pot fi apelați explicit prin *cast*.

Exemplu:

```
#include "Token.h"
Token T("functie", 78);
SmallInt si = SmallInt(T);      // cast
char *string = (char*)T;        // cast
```

Să presupunem că tipul specificat nu se potriveste exact nici uneia din conversiile definite. Va fi invocat totusi un operator de conversie? Răspunsul este :

DA - dacă:
tipul cerut poate fi obtinut printr-o conversie standard

Exemplu:

```
void f(double);
Token T("constanta", 44 );// se invoca T.operator int()
                          // int devine double prin conversie standard
f(T);
```

NU - dacă:
tipul cerut poate fi obținut printr-o a doua conversie definită de utilizator aplicată primei conversii utilizator. (de accea `Token` defineste atât operatorul `SmallInt` cât şi operatorul `int()`).

Exemplu:

Dacă `Token` nu ar fi definit operatorul `int()`, următorul apel ar fi fost ilegal.

```
void f(int);
Token T("pointer", 37 );
f(T);   // eroare daca nu ar fi foast definit operatorul int()
        // Conversia la >int> ar fi trebuit parcursa in doua faze de
        // conversii utilizator:
        // Token::operator SmallInt();   pentru T -> SmallInt
        // SmallInt::operator int();     pentru SmallInt -> int
```

REGULĂ: Poate fi aplicat un singur nivel de conversii utilizator.

Atenție! Operatorii de conversie pot genera ambiguități.

Exemplu:

```
f(int);
f(SmallInt);
Token T("constanta string", 3);
f(T);   // ambiguu; Token defineste conversii atat pentru <int> cat si
        // pentru <SmallInt>.
```

Ambiguitatea poate fi eliminată prin conversie explicită:

```
f( int(T) );
```

Ambiguități pot să apară şi când două clase definesc conversii între ele.

Exemplu:

```
SmallInt::SmallInt(Token&);     // constructor cu un argument
Token::operator SmallInt();

void f(SmallInt);
Token T("acesta este un test", 197);
f(T);   // eroare, exista doua moduri egal posibile de a converti
        // Token intr-un SmallInt
```

Soluție: conversia explicită `f((SmallInt)T)`, şi nu `f(SmallInt(T))` care este tot ambiguă.

S-a spus în capitolele anterioare că execuția unui constructor cuprinde două faze:

- inițializare (conform listei specificate)
- asignare (conform corpului funcției)

De multe ori inițializarea şi asignarea sunt confundate, generând implementări ineficiente.

Exemplu:

```
class X
{
   public:
      X();
      X(int);
      X(const X&);
      X& operator=(const X&);
...
};
class Y
{
   public:
      Y();
   private:
      X x;
};
```

O implementare a constructorului clasei `Y` în forma:

```
Y::Y() { x=0; }
```

determină apelul a 2 constructori ai clasei `X` plus invocarea operatorului de asignare a lui `X`:

1) `X::X()` - constructor implicit

- invocat înaintea constructorului clasei `Y` pentru inițializarea obiectului membru `x`. Aşa cum s-a prezentat în capitolul trecut, compilatorul parcurge urmatoarele etape:
  - există obiect membru? da.
  - există apel de constructor în lista de inițializare? nu.
  - există constructor implicit? da. => se apelează

2) atribuirea `x=0`

- nu poate fi efectuată direct deoarece clasa `X` nu are definit un operator de atribuire cu argument de tip `int`. Astfel, atribuirea se va face în doi paşi:

P1. constructorul `X::X(int)` este apelat pentru conversia întregului `0` într-un obiect de tip `X`

P2. acest obiect nou creat este atribuit lui `x` prin invocarea operatorului `X::operator=(const X&)`.

Implementarea eficientă ar fi:

```
Y::Y() : x(0) {}
```

Acum este invocat doar constructorul `X::X(int)` la apelul fiecărui constructor al lui `Y`.