Programare Orientata pe Obiecte

L A B O R A T O R U L 7

Caracteristici clase C++

SUPRAINCARCAREA OPERATORILOR

C++ incorporeaza optiunea de a utiliza operatorii standard intre clase in plus fata de tipurile fundamentale. De exemplu:

```
int a, b, c;
a = b + c;
```

sunt perfect valide, din moment ce variabilele adunarii sunt toate tipuri fundamentale. Totusi, nu este clar daca se poate executa urmatoarea operatie (de fapt, este incorect):

```
struct { char product [50]; float price; } a, b, c;
a = b + c;
```

Atribuirea unei clase (class sau structura struct) alteia de acelasi tip este permisa (se va apela constructorul de copiere). Ceea ce va produce eroare va fi operatia de adunare care, in principiu, NU este valida intre tipuri nefundamentale.

Insa multumita abilitatii C++ de a supraincarca operatorii, se poate efectua si acest lucru, pana la urma. Obiecte derivate din tipurile compuse, precum cele dinainte, pot accepta operatori care in mod normal nu sunt acceptati, operatori care poti fi modificati ca si efect (cod C++). Urmatoarea lista reprezinta operatorii care poti fi suprainscrisi:

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

Pentru a suprainscrie un operator este necesara o functie membru a unei clase al carei nume este operator, urmat de semnul operatorului de suprainscris, conform urmatorului prototip:

```
type operator sign (parameters);
```

Iata un exemplu de suprainscriere pentru operatorul +. Se vor aduna vectorii bidimensoinali a(3,1) si b(1,2). Adunarea a doi vectori bidimensionali consta in operatia simpla de adunare a coordonatelor x pentru rezultatul noii coordonate x si de adunare a coordonatelor y pentru rezultatul noii coordonate y. In acest caz rezultatul va fi (3+1,1+2)=(4,3).

```
// vectori: exemplu de suprainscriere operatori - 1
#include <iostream.h>

class CVector {
  public:
    int x,y;
    CVector () {};
    CVector (int,int);
    CVector operator + (CVector);
};

CVector::CVector (int a, int b) {
```

```
  x = a;
  y = b;
}

CVector CVector::operator+ (CVector param) {
  CVector temp;
  temp.x = x + param.x;
  temp.y = y + param.y;
  return (temp);
}

int main () {
  CVector a (3,1);
  CVector b (1,2);
  CVector c;
  c = a + b;
  cout << c.x << "," << c.y;
  return 0;
}
```

Motivul pentru care apare atat de des numele de CVector este acela ca in unele situatii acesta reprezinta referinta la clasa cu acelasi nume (ceea ce intoarce operatorul +, de exemplu), iar in alte situatii reprezinta functii cu acelasi nume (constructorul, de exemplu); a nu se confunda:

```
CVector (int, int);            // numele functiei CVector (constructor)
CVector operator+ (CVector);   // functia operator+ care intoarce tipul CVector
```

Functia operator+ a clasei CVector este aceea care suprainscrie operatorul aritmetic +. Acesta poate fi apelat printr-una din urmatoarele doua moduri:

```
c = a + b;
c = a.operator+ (b);
```

Nota: S-a inclus, de asemenea, si constructorul vid (empty) - fara parametrii - care s-a definit printr-un bloc de instructiuni de tip nop (sau no-op - no operation):

```
CVector () { };
```

Acest lucru este necesar din moment ce deja exista un alt constructor

```
CVector (int, int);
```

asadar nu exista nici un constructor implicit in clasa CVector. Includerea constructorului vid permite ca declaratia

```
CVector c;
```

din main(), sa fie valida.

Nota la nota: Blocul de instructiuni nop NU este recomandat pentru implementarea constructorului, deoarece nu satisface cerinta de minima functionalitate pe care un constructor trebuie sa o aiba, anume initializarea tuturor variabilelor clasei.

In cazul constructorului vid al exemplului, variabilele x si y raman neinitializate.

Ca atare o declaratie recomandabila ar fi ceva simlar cu:

```
CVector () { x=0; y=0; };
```

Exercitiul 1

Sa se creeze un program care:

- defineste clasa CVector ca in exemplul de mai sus si foloseste un constructor vid care initializeaza variabilele clasei.
- suprainscrie operatorul - (minus) conform cu rezultatul: (3,1) - (1,2)=(2,-1)

La fel cum o clasa include in mod implicit un constructor vid si unul de copiere (daca nu este nici un constructor definit explicit), include, de asemenea, si definitia implicita a operatorului de atribuire (=) intre doua clase de acelasi tip. Acest operator copieaza intreg continutul datelor membre non-statice ale obiectului parametru (cel din dreapta semnului de egal) in obiectul destinatie (cel din stanga semnului de egal). Bineinteles ca si acest operator poate fi suprainscris, asociindu-i o alta functionalitate dorita de utilizator, cum ar fi copierea a numai o parte din membrii clasei.

Supraincarcarea operatorilor nu inseamna in mod necesar ca operatia construieste o relatie matematica, desi acest lucru este recomandat. De exemplu, nu este chiar logica utilizarea operatorului + pentru a scadea doua clase, sau folosirea operatorului == pentru a intializa la 0 o clasa, desi este perfect posibil acest lucru.

Prototipul functiei operator+ poate fi evident: acesta ia partea dreapta a operatorului ca parametru pentru functia operator+ a obiectului din stanga.

Alti operatori poate nu sunt atat de evidenti. In tabelul de mai jos se poate observa cum trebuiesc declarati diversi operatori (se va inlocui @ cu operatorul in cauza din coloana II):

Expresie	Operator (@)	Functie membru	Functie globala
@a	+ - * & ! ~ ++ --	A::operator@()	operator@(A)
a@	++ --	A::operator@(int)	operator@(A, int)
a@b	+ - * / % ^ & | < > == != <= >= << >> && || ,	A::operator@(B)	operator@(A, B)
a@b	= += -= *= /= %= ^= &= |= <<= >>= [ ]	A::operator@(B)	-
a(b, c...)	()	A::operator()(B, C...)	-
a->b	->	A::operator->()	-

unde a este un obiect al clasei A, b este un obiect al clasei B si c este un obiect al clasei C.

Din ultimul tabel se poate observa ca exista doua modalitati de a suprainscrie operatorii unei clase: ca functie membru sau ca functie globala. Utilizarea uneia sau alteia este indiferenta, dar trebuie mentionat ca functiile care nu sunt membre ale clasei nu pot accesa membrii privati (private) sau protejati (protected) in mod direct, in afara cazului in care functia globala este prietena cu clasa.

Cuvantul cheie this

Cuvantul cheie this reprezinta, in cadrul unei clase, adresa de memorie a obiectului acelei clase care este in curs de executare. In fapt acesta reprezinta un pointer a carui valoare este intotdeauna adresa obiectului. Cuvantul cheie this poate fi utilizat pentru a verifica daca un parametru transferat unei functii membru a unui obiect este el insusi un obiect. Exemplu:

```
// this
#include <iostream.h>

class CDummy {
  public:
    int isitme (CDummy& param);
};

int CDummy::isitme (CDummy& param)
{
  if (&param == this) return 1;
  else return 0;
}

int main () {
  CDummy a;
  CDummy* b = &a;
  if ( b->isitme(a) )
    cout << "yes, &a is b";
  return 0;
}
```

Operatorul this este, de asemeni, utilizat frecvent in functia membru operator=, care intoarce obiecte prin referinta (evitand utilizarea obiectelor temporare). Suprainscrierea operatorului = ar putea arata in felul urmator:

```
CVector& CVector::operator= (const CVector& param)
{
  x=param.x;
  y=param.y;
  return *this;
}
```

ceea ce ar putea reprezenta chiar codul implicit generat de catre compilator pentru clasa, in cazul in care nu s-ar defini functia membru operator=.

Exercitiul 2

Sa se introduca suprainscrierea operatorului = de mai sus in exercitiul 1.

Sa se compileze si sa se ruleze programul rezultat.

Membrii statici

O clasa poate contine si membrii statici, date sau functii.

Membrii date statice ale unei clase sunt cunoscute sub denumirea de "variabile ale clasei", deoarece continutul lor nu depinde de nici un obiect (de tipul clasei respective, bineinteles). Exista o unica valoare pentru toate instantele (obiectele) unei clase.

De exemplu, o variabila a clasei poate fi utilizata ca sa tina evidenta numarului de obiecte (instante) declarate ale clasei, ca in exemplul urmator:

```
// static members in classes
#include <iostream.h>

class CDummy {
  public:
    static int n;
    CDummy () { n++; };
    ~CDummy () { n--; };
};

int CDummy::n=0;

int main () {
  CDummy a;
  CDummy b[5];
  CDummy * c = new CDummy;
  cout << a.n << endl;
  delete c;
  cout << CDummy::n << endl;
  return 0;
}
```

In fapt membrii statici au aceleasi proprietati ca si variabilele globale, dar se bucura si de proprietatile asociate clasei. Din acest motiv si, evitand declararea de mai multe ori, in concordanta cu standardul C++, se poate doar include prototipul (declaratia) in declaratia clasei, nu insa si definitia (initializarea). Pentru a initializa un membru data static trebuie inclusa o definitie formala in afara clasei, in scop global, ca in exemplul anterior.

Deoarece este vorba de o variabila unica pentru toate obiectele aceleasi clase (instantelor), aceasta (variabila) poate fi referita ca membru al oricarui obiect al acelei clase sau chiar direct de numele clasei (aceasta referire este valida numai pentru membrii statici):

```
cout << a.n;
cout << CDummy::n;
```

Aceste doua apeluri incluse in exemplul precedent refera aceeasi variabila: variabila statica n din interiorul clasei CDummy.

Nota: de reamintit ca variabila clasei asociata unei clase este, in fapt, o variabila globala. Singura diferenta o reprezinta numele din afara clasei. Asa cum se pot introduce date statice intr-o clasa, tot astfel se pot include si functii. Acestea vor reprezenta functii globale care sunt apelate ca si cum ar fi membrii ai obiectelor pentru o clasa data. Functiile statice pot referi numai date statice si NU POT folosi cuvantul cheie this, din moment ce acesta face referire la un pointer de tip obiect si aceste functii, in fapt, nu sunt membrii directi ai nici unui obiect, ci membrii directi ai clasei.

Exercitiul 3

Sa se modifice programul din exemplul anterior, astfel:

- sa se schimbe membrul data n din public in privat
- sa se adauge doua functii membre statice si publice care sa acceseze membrul
- data static n, cu prototipurile:
- static int readN(void);
- static void writeN(int);
- sa se implementeze functionalitatea programului din exemplu anterior:
- incrementarea valorii variabilei statice si private n in cadrul constructorului
- decrementarea valorii variabilei statice si private n in cadrul destructorului
- se se modifice in mod corespunzator afisarea membrului static si privat n din main().

Functii prietene (cuvantul cheie friend)

S-a observat anterior ca exista trei nivele de protectie interna pentru diferiti membrii ai clasei: public, protected si private. In cazul membrilor protected si private acestia nu pot fi accesati din afara aceleasi clase unde au fost declarate.

Totusi, aceasta regula poate fi "incalcata" prin utilizarea cuvantului cheie friend in cadrul clasei, pentru a permite functiilor externe sa isi castige accesul la membrii protected si private ai clasei.

Pentru a permite unei functii externe sa aiba access la mebrii private sau protected ai unei clase trebuie declarat prototipul functiei externe care va castiga accesul prin adaugarea cuvantului cheie friend - prototipul trebuie declarat in cadrul declaratiei clasei care isi partajeaza membrii. In urmatorul exemplu se declara functia prietena duplicate:

```
// functii prietene (friend)
#include <iostream.h>

class CRectangle {
    int width, height;
  public:
    void set_values (int, int);
    int area (void) {return (width * height);}
    friend CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b) {
  width = a;
  height = b;
}

CRectangle duplicate (CRectangle rtparam)
{
  CRectangle rtres;
  rtres.width = rtparam.width*2;
  rtres.height = rtparam.height*2;
  return (rtres);
```

```
}

int main () {
  CRectangle rt, rtb;
  rt.set_values (2,3);
  rtb = duplicate (rt);
  cout << rtb.area();
}
```

Din cadrul functiei duplicate, care este prietena cu CRectangle, nu este posibil accesul asupra membrilor width si height ale diverselor obiecte de tip CRectangle.

De observat ca nici in declaratia lui duplicate() nici in utilizarea lui ulterioara din main() nu s-a considerat ca functia duplicate() ar fi membra a clasei CRectangle. Nici nu este!

Functiile prietene pot servi, de exemplu, la implementarea operatiilor dintre doua clase diferite. In general, utilizarea functiilor prietene este in afara filozofiei de programare orientata-obiect, deci, de cate ori este posibil, mai bine se vor utiliza membrii ai aceleiasi clase pentru implementare. Precum in exemplul anterior, ar fi fost mai usoara integrarea functiei duplicate() in cadrul clasei CRectangle.

Supraincarcarea operatorilor << si >> pentru tipuri de date definite de programator

Este posibila supradefinirea operatorilor de inserare/extragere din stream pentru a permite utilizarea lor si pentru tipuri de date definite de programator, pe langa cele standard.

Declaratiile operatorilor pot fi de forma:

```
istream& operator>> (istream&, tip_utilizator&);
ostream& operator<< (ostream&, tip_utilizator);
```

Primul argument trebuie sa fie o referinta la un obiect istream sau ostream, sau al uneia din clasele derivate acestora. De aceea, functiile operator nu pot fi membre ale clasei definite de utilizator. De regula sunt functii independente prietene ale clasei definite de utilizator. Rezultatul returnat trebuie sa fie adresa obiectului stream primit ca argument, pentru a permite efectuarea unei secvente de operatii.

Exemplu:

```
class NrComplex {
 float re, im;
 public:
  NrComplex(float r=0, float i=0) {
     re=r; im=i;
     }
  friend istream& operator >> (istream&, NrComplex&);
  friend ostream& operator << (ostream&, NrComplex);
 };
istream& operator >> (istream& in, NrComplex& c) {
  in>> c.re >> c.im;
  return in;    }
```

```
ostream& operator << (ostream& out, NrComplex c){
  out << c.re << "+i*" << c.im;
  return out;
}
```

Clase prietene (friend)

Asa cum exista posibilitatea definirii de functii prietene, tot asa se poate defini o clasa ca fiind prietena cu alta clasa, permitand celei de-a doua accesul la membrii protected si private ai primei clase. Exemplu:.

```
// clasa prietena (friend)
#include <iostream.h>

class CSquare;

class CRectangle {
    int width, height;
  public:
    int area (void)
      {return (width * height);}
    void convert (CSquare a);
};

class CSquare {
  private:
    int side;
  public:
    void set_side (int a)
      {side=a;}
    friend class CRectangle;
};

void CRectangle::convert (CSquare a) {
  width = a.side;
  height = a.side;
}

int main () {
  CSquare sqr;
  CRectangle rt;
  sqr.set_side(4);
  rt.convert(sqr);
  cout << rt.area();
  return 0;
}
```

In acest exemplu s-a declarat clasa CRectangle ca prietena a clasei CSquare astfel incat CRectangle poate accesa mebrii protected si private ai clasei CSquare, mai concret, data membru CSquare::side, care defineste latura patratului.

Se poate observa, de asemenea, ceva nou in programul de mai sus: prima instructiune, care reprezinta un prototip vid al clasei CSquare. Aceasta este necesara pe motivul ca in cadrul functiei CResctangle se face referire la CSquare (ca parametru in functia convert()). Definitia lui CSquare este

inclusa mai tarziu, iar daca nu ar fi fost instructiunea prototipului vid, clasa CSquare nu ar fi fost vizibila din cadrul definitiei lui CRectangle - compilatorul ar fi sesizat o eroare, in acest caz.

Prietenia nu se subintelege si reciproc, decat daca este explicit specificata. In CSquare, CRectangle este considerata o clasa prietena, deci CRectangle poate accesa membrii protected sau private ai lui CSquare, dar nu si invers. Pentru reciprocitate trebuie ca si in clasa CRectangle, CSquare sa fie considerata prietena.

Exercitiul 4

Sa se modifice exemplul de mai sus astfel:

- sa se considere CSquare prietena in cadrul CRectangle
- sa se adauge functia publica void CSquare::convert(CRectangle cr) care sa modifice dimesiunea laturii patratului ca fiind suma dintre latimea si lungimea dreptungiului
- sa se adauge functia publica CSquare::area() care sa calculeze aria patratului
- sa se adauge constructorul de initializare CRectangle::CRectangle(int a, int b), care sa initializeze latimea si lungimea dreptunghiului; sa se defineasca un obiect care sa apeleze acest constructor
- sa se apeleze functia de conversie void CSquare::convert(CRectangle cr) unde parametrul cr reprezinta obiectul initializat mai sus
- sa se afiseze aria patratului obtinut anterior