Supraincarcarea operatorilor

Supraincarcarea functiilor

C++ permite sa avem mai multe functii cu acelasi nume. Cand rulam programul, compilatorul C++ determina care functie trebuie apelata in functie de numarul si tipul parametrilor pe care intructiunea de apelare ii transmite functiei. Exemplu:

```
#include "stdafx.h"

using namespace std;

int suma(int *matrice, int nr_element)
{
	int rezultat=0;
	int nr;
	for (nr=0; nr<nr_element; nr++)
		rezultat+=matrice[nr];
	return rezultat;
}

double suma(double *matrice, int nr_element)
{
	double rezultat=0;
	int nr;
	for (nr=0; nr<nr_element; nr++)
		rezultat+=matrice[nr];
	return rezultat;
}

void main()
{
	int a[5]={1,2,3,4,5};
	double b[4]={1.11,2.22,3.33,4.44};
	cout<<"suma valorilor int este "<<suma(a,5)<<endl;
	cout<<"suma valorilor double este "<<suma(b,4)<<endl;
}
```

Functiile pot avea acelasi tip, dar atunci trebuie sa difere numarul de parametrii, iar daca au acelasi numar de parametrii trebuie neaparat sa fie de alt tip.

Ce ascundem si ce facem public?

Ca regula generala, cu cat un program stie mai putin despre o clase cu atat mai bine. De aceea ar trebui sa utilizati date si metode private cat mai des posibil. Atunci cand utilizati date si metode private, programele care utilizeaza obiectul trebuie sa foloseasca metodele publice ale obiectului pentru a avea acces la datele obiectului.

Supraincarcarea unui operator

Prin supradefinirea operatorilor se pot realiza operatii specifice unui nou tip de date la fel de usor ca in cazul tipurilor de date standard.

Cand supraincarcati un operator trebuie sa continuati sa utilizati operatorul in formatul sau standard, adica daca supraincarcati operatorul (+) acesta trebuie sa utilizeze operatorul sub forma *operand+operand.* Puteti sa supraincarcati doar operatorii existenti. C++ nu va permite definirea unor operatori proprii. Operatorul supraincarcat se aplica numai instantelor clasei specificate. De exemplu, daca avem o clasa Sir si am incarcat operatorul plus astfel incat operatorul sa concateneze doua siruri de caractere *sir_nou=sir1+sir2;* daca utilizati operatorul plus supraincarcat, cu doua valori intregi, supraincarcarea nu se va aplica. Operatori care NU pot fii supraincarcati sunt **. .* :: ?:**
NU puteti utiliza o functie friend pentru supraincarcarea urmatorilor operatori = [ ] ( ) ->

Adesea functiile operator returneaza un obiect al clasei asupra caruia opereaza. Atunci cand supraincarcati un operator unar lista de argumente trebuie sa fie goala, iar cand supraincarcati un operator binar lista de argumente trebuie sa contina un singur parametru.

Sintaxa:

```
tip_return nume_clasa operator #(lista_argumente){
                    //operatii
            }
```

Supradefinirea se poate face folosind:
· functii membre nestatice
· functii prietene

Probleme:

1. Supraincarcarea operatorului +

Utilizam clasa Complex pentru a supraincarca operatorii + si –

```
class Complex
{
int re, im;
public:
Complex(int re=0, int im=0);
~Complex(void);
void afisare();
};

Complex::Complex(int re, int im)
{
this->re=re;
this->im=im;
}
Complex::~Complex(void)
{
}
void Complex::afisare()
{
cout << re << "+" << im << "i" << "\n";
}
```

Adaugam in clasa Complex urmatorul prototip de declaratie a functiei clasei Complex:

```
friend Complex operator +(const Complex&, const Complex&);
```

Si definirea functiei:

```
Complex operator +(const Complex& c1, const Complex& c2)
{
cout<<"se apeleaza operatorul + supraincarcat "<<endl;
Complex tmp;
tmp.re = c1.re + c2.re;
tmp.im = c1.im + c2.im;
return tmp;
}
```

Daca avem doua variabile de tip Complex x si y apelul x+y se va transforma in operator+(x,y)

Functia main corespunzatoare:

```
void main()
{
Complex c1(1,1), c2(2,2);
Complex r1, r2;
cout<<"c1=";c1.afisare();
cout<<"c2=";c2.afisare();
r1 = c1 + c2;
cout<<"r1=";
r1.afisare();
r2 = c1 + c2 + r1; //se evalueaza (c1 + c2) + r1 tinandu-se seama de
                   //asociativitatea operatorilor
cout<<"r2=";
r2.afisare();
}
```

Supraincarcarea operatorilor folosind functii membre:

2. Supraincarcarea operatorului -

Adaugam la programul anterior prototipul functiei

```
Complex operator- (const Complex&);
```

Definirea functiei

```
Complex Complex::operator- (const Complex& c)
{
cout<<"se apeleaza operatorul - supraincarcat "<<endl;
Complex tmp;
tmp.im = this->im - c.im;
tmp.re = this->re - c.re;
return tmp;
}
```

Daca avem doua variabile de tip Complex x si y apelul x-y se va transforma in x.operator-(y)

Restrictie: in acest caz tipul primului operand este mereu tipul clasei.

Proprietatile operatorilor care nu pot fi modificate:
· pluraritatea
· precedenta si asociativitatea

3. Supradefinirea operatorului =

In lipsa supradefiniri operatorului = atribuirea se face membru cu membru in mod similar constructorului de copiere.
Pentru exemplificare vom folosi clasa Stiva:

```
class Stiva
{
 int dim;
 int pozCurenta;
 int* tab;
public:
 Stiva(int dim=0);
 Stiva(const Stiva &);
 ~Stiva(void);
 void adaug(int el);
 void afisare(void);
};

Stiva::Stiva(int dim)
{
cout<<"se apeleaza constructorul explicit "<<endl;
this->dim = dim;
pozCurenta = 0;
tab = new int[dim];
}

Stiva::~Stiva(void)
{
delete tab;
}

void Stiva::afisare(void)
{
cout << "stiva contine: " ;
for (int i=0; i <pozCurenta; i++){
cout << tab[i] << " ";
}
cout << endl;
}

Stiva::Stiva(const Stiva &s)
{
cout << "se apeleaza constructorul de copiere "<<endl;
this->dim = s.dim;
this->pozCurenta = s.pozCurenta;
for (int i=0; i <pozCurenta; i++){
this->tab[i] = s.tab[i];
}
```

```
}

void Stiva::adaug(int el)
{
cout << "adaug un element in stiva "<<endl;
tab[pozCurenta++] =el;
}
```

Adaugam in clasa Stiva prototipul:

```
Stiva& operator= (Stiva &);
```

Si definim functia:

```
Stiva& Stiva::operator= (Stiva &s)
{
cout << "se apeleaza operatorul supraincarcat = " << endl;
if (this != &s)
{
this->dim = s.dim;
this->pozCurenta = s.pozCurenta;
for (int i=0; i <pozCurenta; i++){
this->tab[i] = s.tab[i];
}
}
return *this;
}
```

Exemplu:

```
int main()
{
Stiva s1(10), s2(20), s3(30);
s1 = s2;
s3 = s2 =s1;
getchar();
return 0;
}
```

4. Operatorul []

Este un operator binar si vom exemplifica supraincarcarea pentru clasa Stiva pentru a avea acces la membri tabloului.

Se adauga urmatorul prototip de declaratie a functiei clasei Stiva:

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

si definirea functiei:

```
int & Stiva::operator[](int i)
{
cout << "se apeleaza operatorul supraincarcat [] " << endl;
return tab[i];
}


int main()
{
Stiva s1(10), s2(20), s3(30);
s1.adaug(3);
s1.adaug(4);
s1.adaug(8);
s1.afisare();
cout << "Stiva [0]: " <<s1[0] << endl;
getchar();
return 0;
}
```

5. Studiati urmatorul exemplu, rulati programul, explicati ce se intampla:

```
#include <stdafx.h>
#include <iostream>
#include <stdlib.h>

using namespace std;

class Loc
{
     int longitudine;
     int latitudine;
public:
     Loc(){}
     Loc(int lg, int lt)
     {
          longitudine=lg;
          latitudine=lt;
     }
     ~Loc()
     {
          //cout<<"Am distrus obiectul "<<endl;
     }
     void print()
     {
```

```
            cout<<"longitudine "<<longitudine<<" ";
            cout<<"latitudine "<<latitudine<<endl;
      }
      Loc operator=(Loc op2);
      friend Loc operator++(Loc &op1); //Supraincarcarea prin
// functie friend
      friend Loc operator--(Loc &op1); // -||-

      void *operator new(size_t dimensiune);
      void operator delete(void *p);
};

Loc Loc::operator=(Loc op2)
{
      longitudine=op2.longitudine;
      latitudine=op2.latitudine;
      return *this;
}

Loc operator++(Loc &op1)
{
      op1.longitudine++;
      op1.latitudine++;
      return op1;
}

Loc operator--(Loc &op1)
{
      op1.longitudine--;
      op1.latitudine--;
      return op1;
}

void *Loc::operator new(size_t dimensiune)
{
      cout<<"Functia new proprie. "<<endl;
      return malloc(dimensiune);
}

void Loc::operator delete(void *p)
{
      cout<<"Functia delete proprie. "<<endl;
      free(p);
}

void main()
{
      Loc ob1(10,20), ob2;
      ob1.print();
      ++ob1;
      ob1.print();
```

```
	ob2=++ob1;
	ob2.print();

	--ob2;
	ob2.print();

	Loc *p1, *p2;
	p1=new Loc(40,50);
	if (!p1)
	{
		cout<<"Eroare la alocare  "<<endl;
		exit(1);
	}
	p2=new Loc(-40,-50);
	if (!p2)
	{
		cout<<"Eroare la alocare !!"<<endl;
		exit(1);
	}
	p1->print();
	p2->print();
	delete p1;
	delete p2;
}
```

Dupa cum ati observat putem supraincarca si operatorii new si delete daca dorim sa alocam o anumita metoda speciala de memorie.

Tipul *size_t* trebuie sa fie capabil sa pastreze partea cea mai mare din memorie pe care functia supraincarcata new o poate aloca.

Parametrul *dimensiune* trebuie sa contina numarul de octeti pe care *new* ii cere pentru a pastra obiectul tocmai alocat.

Functia *new* trebuie sa returneze un pointer la memoria alocata sau sa returneze NULL daca esueaza.

Pentru a elibera memoria alocata cu *new* trebuie sa reincarcam operatorul *delete*. Operatorul *delete* trebuie sa primeasca un pointer la memoria pe care operatorul *new* a alocat-o anterior pentru obiect.

Puteti incarca operatorii *new* si *delete* fie global, fie relativ la o clasa sau la mai multe clase.

Tema

1. Definiti o clasa AgendaTelefon care contine o lista de perechi de genul nume, numar telefon. Programul trebuie sa poata efectua urmatoarele operatii:
· reuniunea a doua agende de telefon
· diferenta a doua agende de telefon
· accesarea directa a unui numar de telefon prin intermediul numelui (supradef op [])
· afisare, adaugarea/stergerea de perechi nume, numar telefon.

2. Proiectati si implementati clasa Rational care sa permita lucru cu numere rationale (adunare, scadere, inmultire, impartire):
-Constructorul clasei va avea doua argumente : numarartorul si numitorul numarului rational (constructorul poate avea si un singur argument, caz in care al doilea se ia implicit 1, sau nici un argument, caz in care se iau valorile implicite 0 si 1) ;
-Se va asigura un constructor de copiere ;
Se vor prevedea functii membre pentru : accesul la numaratorul, respectiv numitorul numarului rational ;
-Daca sunt egale 2 sau mai multe numere rationale, se va afisa pe ecran acest lucru .

3. Modificati problema 1 (clasa Rational) astfel incat sa supraincarcati toti operatorii necesari.

Probleme suplimentare

4. Proiectati si implementati o clasa Vector de numere reale si o clasa Matrice de numere reale. Implementati cel putin urmatoarele:
a) construirea claselor, constructori si destructori corespunzatori
b) produsul scalar a doi vectori
c) adunarea, scaderea, inmultirea a doi vectori
d) adunarea, scaderea, inmultirea a doua matrici, inversa unei matrici
e) inmultirea unui vector si a unei matrici cu o constanta reala
f) produsul unui vector cu o matrice
Alegeti cu grija functiile membre friend.