POO – C++ - Laborator 6

Cuprins

1. Funcţii prietene. Clase prietene

În C++ putem accesa membrii privaţi sau protejaţi ai unui obiect folosind funcţii globale sau metode aparţinând altor clase. Condiţia este ca aceste funcţii, respectiv clase, să fie declarate **prietene** cu clasele ale căror membri privaţi sau protejaţi le accesează. În acest scop se utilizează cuvântul cheie **friend**.

Relaţia de prietenie se poate declara între :

a. O funcţie globală şi o clasă ;
b. O funcţie membră a unei clase şi o altă clasă ;
c. Între 2 clase diferite.

Considerăm următorul exemplu în care am exemplificat relaţia (a) şi (c):

```
class Punct {
      int x, y;
public:
      Punct(int x1, int y1) {
            x = x1;
            y = y1;
      };
      friend void f(void);// functie prietena
      friend class X;// clasa prietena
};
void f() {
      Punct a(2,0);
      a.y = 9;/* acces la un membru privat al
obiectului*/
}
class X {
      Punct p;
public:
      void g() {
```

```
            p.x++;// acces la membru privat
      }
};
```

În următorul exemplu prezentăm relaţia de prietenie dintre metoda unei clase şi o altă clasă:

```
class Y {
public:
      void f();
};

class X {
private:
      int a, b;
      friend void Y::f();
};

void Y::f() {
      X x;
      x.a = 1;
}
```

Notă: Relaţia de prietenie între clase nu este simetrică. Dacă A este prietena lui B, aceasta nu implică direct că B este prietena lui A. Dacă dorim, putem să declarăm 2 clase să fie reciproc prietene – A prietenă cu B şi B prietenă cu A. Relaţia de prietenie nu este nici tranzitivă. Dacă A este prietenă cu B, iar B prietenă cu C, aceasta nu inseamnă în mod automat că A este prietenă cu C.

2. Supraîncărcarea operatorilor

Limbajul C++ permite ca acţiunea operatorilor să fie redefinită pentru noi tipuri de date. De exemplu putem defini clasa matrice, şi operatorii `+` şi `*` , care să efectueze adunarea şi produsul a 2 matrici. Codul care ar face adunarea între 2 matrici ar arăta exact ca şi adunarea între 2 numere:

```
Matrice a,b,suma,produs;

//... iniţializare a si b

suma = a + b;
produs = a * b;
```

Pentru a supraîncărca un operator care să poată fi aplicat obiectelor, trebuie să definim ***funcţiile operator,*** care pot avea domeniu local (declarate în cadrul clasei) sau global (declarate şi definite în afara clasei). Funcţiile operator sunt funcţii a căror nume este format din cuvântul cheie `operator` urmat de simbolul operatorului. De exemplu `tip operator+(...).`

În continuare, vom dezvolta clasa `Complex` din laboratoarele anterioare. Vom adăuga operatorul de adunare (`+`), supraîncărcat pentru numere complexe.

```
//supraincarcare operator +
#include<iostream>
```

```
using namespace std;
class Complex {
      int re,im;
public:
      Complex () {};
      Complex (int,int);
      Complex operator + (Complex);
      void afisare();
};

Complex::Complex (int re, int im) {
      this->re = re;
      this->im = im;
}

Complex Complex::operator+(Complex c2) {
      Complex temp;
      temp.re = this->re + c2.re;
      temp.im = this->im + c2.im;
      return temp;
}

void Complex::afisare() {
      cout << "(" << re << "," << im << ")" << endl;
}

int main () {
      Complex a(3,1);
      Complex b(1,2);
      Complex c;
      c = a + b;
      c.afisare();
      return 0;
}
```

Ieşire:

```
(4,3)
```

Metoda `operator+` din clasa `Complex` realizează supraîncărcarea operatorului de adunare (`+`). Observăm că metoda are ca parametru un singur număr complex, deşi realizează adunarea a 2 numere complexe. Acest lucru se datorează faptului că primul parametru al metodei este implicit şi este pointerul `this`, fiind şi primul operand din operaţia de adunare.

Această metodă poate fi apelată atât implicit folosind simbolul +, cât şi explicit, folosind numele funcţiei:

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

Cele două expresii sunt echivalente.

Funcţiile operator pot fi definite atât ca metode membre (ca în exemplul de mai sus) cât şi sub formă de funcţii globale. În cazul în care sunt definite ca metode, operandul din stânga va fi chiar `this`, iar operandul din dreapta va fi transmis ca parametru. În cazul în care funcţia operator este globală, aceasta va avea 2 parametri reprezentând cei doi operanzi.

Pentru a putea accesa membrii privaţi ai clasei care a instanţiat obiectele parametri avem 2 posibilităţi:

1. Declarăm funcţia globală prietenă a clasei a cărei instanţă sunt obiectele parametru.

2. Să creăm metode prin care să citim valorile câmpurilor private.

De exemplu, pentru clasa `Complex`, putem supraâncărca operatorul + ,ca funcţie globală, şi prietenă a clasei `Complex`:

```
class Complex {
      int re,im;
public:
      Complex () {};
      Complex (int,int);
      void afisare();
      friend Complex operator+(Complex, Complex);
};

//... Definirea celorlalti membri
Complex operator+(Complex c1, Complex c2) {
      Complex temp;
      temp.re = c1.re+c2.re;
      temp.im = c1.im + c2.im;
      return temp;
}

int main () {
      Complex a(3,1);
      Complex b(1,2);
      Complex c;
      c = a + b;
      c.afisare();
      return 0;
}
```

În locul funcţiilor prietene putem să ne garantăm accesul la membrii privaţi, definind două metode pentru citirea câmpurilor private.

În general, parametrul funcţiei operator poate fi de orice tip. De exemplu, putem să supraîncărcăm din nou operatorul +, care va avea parametru un număr întreg. Funcţia va aduna numărul întreg atât la partea reală cât şi la partea imaginară. Şi valoarea returnată de o funcţie operator poate fi de orice tip, char şi `void`. Similar cu operatorul + poate fi implementat orice alt operator binar.

Şi operatorii unari, gen `++`, pot fi supraîncărcaţi.

În tabelul de mai jos este prezentat modul cum pot fi declarate diverse funcţii operator (înlocuiţi `@` cu operatorul în fiecare caz):

Expresie	**Operator**	**Metodă**	**Funcţie globală**
@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)	-

Pe baza tabelului de mai sus, putem implementa operatorul unar `!`, care va realiza afişarea numărului complex:

```
void Complex::operator!() {
      cout << "(" << re << "," << im << ")" << endl;
}
```

Operatorul poate fi utilizat astfel:

```
Complex a(3,1);
!a;
```

Echivalentul operatorului `!` implementat ca funcţie globală are următorul prototip:

```
void operator!(Complex c1);
```

3. Supraîncărcarea operatorilor ++ şi ––

Operatorii unari `++` şi `––` au particularitatea că se pot utiliza în două moduri:

1. Ca prefix, de exemplu `++obiect`, caz în care întâi se face incrementarea cu unu, apoi se utilizează obiectul;
2. Ca sufix, de exemplu `obiect++`, caz în care întâi se utilizează obiectul iar apoi se face incrementarea cu unu.

Implicit, supraîncărcarea operatorilor unari `++` şi `––` se face pentru forma prefixată. Să exemplificăm pentru operatorul `++`, considerând o clasă `Numar`:

```
#include<iostream>
using namespace std;
class Numar {
      int nr;
public:
      Numar()  { nr = 0; }
      Numar &operator++()  {
            ++nr;
            return  *this;
      }
```

```
	void afisare() {
		cout<<nr<<endl;
	}
};

int main() {
	Numar nr;
	++nr;
	nr.afisare();
	return 0;
}
```

Ieşire:

```
1
```

Dar dacă se doreşte implementarea operatorului ++ ca sufix, vom proceda astfel:

```
#include<iostream>
using namespace std;
class Numar {
	int nr;
public:
	Numar()  { nr = 0; }
	Numar &operator++(int a)  {
		nr++;
		return  *this;
	}
	void afisare() {
		cout<<nr<<endl;
	}
};

int main() {
	Numar nr;
	nr++;
	nr.afisare();
	return 0;
}
```

Ieşire:

```
1
```

Se observă că funcţia operator are un parametru de tip întreg. Acest parametru nu este folosit. El serveşte pentru a face diferenţa între funcţia `operator++()` prefixat, şi funcţia `operator++()` postfixat.

Funcţiile `operator++` prefixat, şi postfixat, supraîncărcate ca funcţii globale, mai au un parametru în plus. Prototipurile sunt următoarele:

```
Numar &operator++(Numar &nr);
Numar &operator++(Numar &nr, int a);
```

2. Supraîncărcarea operatorului de atribuire (=)

Un operator special este cel de atribuire (=), care se apelează ori de câte ori se întâlneşte o expresie de tipul `a = b`. Acesta este singurul operator definit de compilator implicit. Implementarea implicită copie valoarea fiecărui câmp al operandului din dreapta în câmpul corespunzător al obiectului operand stânga. Forma generală a operatorului = este:

```
X &operator=(const X&);//X – numele clasei
```

Pentru clasa `Complex` de mai sus, compilatorul generează următoarea **implementare implicită**:

```
Complex &operator=(const Complex &sursa) {
      this->re = sursa.re;
      this->im = sursa.im;
      return *this;
}
```

Operatorul = returnează o referinţă la obiect pentru a permite atribuiri înlănţuite `(a = b = c = d)`.

În continuare prezentăm un exemplu de cod în care se apelează operatorul de atribuire:

```
Complex d(2,3);
Complex e;
e = d;// apelare operator de atribuire
```

Supraîncărcarea de către programator a operatorului = este absolut necesară atunci când clasa care instanţiază obiectul **conţine câmpuri de tip pointer**. În supraîncărcarea operatorului egal se vor copia zonele de memorie referite de pointeri, şi în caz de nevoie, se va face realocarea.

Prezentăm următorul exemplu:

```
#include<iostream>
#include<string.h>
using namespace std;
#pragma warning(disable : 4996)

class Persoana {
      char *nume;
public:
      Persoana(char *nume) {
            this->nume = new char[strlen(nume)+1];
            strcpy(this->nume, nume);
      };
      Persoana(const Persoana &p) {
            nume = new char[strlen(p.nume)+1];
            strcpy(nume, p.nume);
            cout<<"Constructor de copiere: "
              << nume<< endl;
      }
      Persoana &operator=(const Persoana &p) {
            if (nume != NULL) {
```

```
                  delete[] nume;
            }
            nume = new char[strlen(p.nume)+1];
            strcpy(nume, p.nume);
            cout<<"Operatorul= " << nume<< endl;
            return *this;
      }
      ~Persoana() {
            if (nume != NULL) {
                  delete[] nume;
            }
            cout<<"~Persoana()" << endl;
      }
};

void main() {
      Persoana popescu("popescu");
      Persoana pop("pop");
      Persoana pop2=pop;
      pop2 = popescu;
}
```

Ieşire:

```
Constructor de copiere: popescu
Operatorul= popescu
~Persoana()
~Persoana()
~Persoana()
```

3. Exerciţii

1. Completaţi clasa `Complex` din laborator cu următorii operatori:
   1.1. –
   1.2. *
   1.3. ==
   1.4. ~ - modulul numărului complex

   Scrieţi un program care citeşte de la tastatură 2 numere complexe, şi returnează rezultatul celor 4 operatori aplicaţi asupra numerelor.

2. Completaţi clasa `Multime` din laboratorul 3, problema 1, cu următorii membri:
   - Operatorul `"+="` cu parametru `int` – adaugă un element la mulţime, echivalent cu funcţia `Multime::adauga()`.
   - Operatorul `"-="` cu parametru `int` – extrage un element din mulţime, echivalent cu funcţia `Multime::extrage()`.
   - Operatorul `"="` . Atenţie la datele alocate dinamic.
   - Constructorul de copiere.
   - Operatorul `"+="` cu parametru `Multime`. Va adăuga la mulţimea curentă elementele mulţimii primite ca parametru. Practic, după această operaţie mulţimea curentă va deveni reuniunea dintre cele 2 mulţimi operanzi. Va fi utilizat într-o expresie de genul `a+=b`

- Operatorul "+" cu parametru `Multime`. Va realiza reuniunea dintre cele 2 mulţimi operanzi. Spre deosebire de operatorul "+=" , mulţimea reuniune va fi un obiect nou, returnat de funcţia operator. Mulţimile operanzi nu vor fi modificate. Se va utiliza într-o expresie de genul `a=b+c`

Realizaţi un program care testează toţi aceşti operatori.

3. Implementaţi o clasă `String` care să reprezinte un şir de caractere şi operaţiile aferente. Definiţi următorii membri:
   3.1. Operatorul `+`, concatenarea şirurilor.
   3.2. Operatorul =
   3.3. Operatorul ==
   3.4. Metoda
   `int cauta(String subsir)`
   realizează căutarea unui subşir într-un şir. Returnează prima poziţie în şirul curent, în care a fost găsit `subsir`. Sau `-1` dacă subşirul nu a fost găsit. De exemplu `sir.cauta(subsir)` va returna 3, dacă `sir` reprezintă "alabala" iar `subsir` - "ba".
   3.5. Metoda
`void afisare()`
Afişează şirul.
   3.6. Metoda
`int compara(String sir2)`
Realizează compararea a 2 şiruri în ordine lexicografică. Returnează -1 dacă şirul curent (`this`) este mai mic decît `sir2`, 0 dacă şirurile sunt egale, şi 1 dacă şirul curent este mai mare.
   3.7. Constructorul vid – crează un şir vid.
   3.8. Constructorul cu argument un şir de caractere (`char *`).
   3.9. Constructorul de copiere.
   3.10. Destructorul.