POO – C++ - Laborator 5

Cuprins

1. Membri statici ai claselor

Folosind cuvântul cheie **static** unii membri ai clasei pot fi declaraţi de tip static. Membrii de tip dată declaraţi statici au următoarele proprietăţi:

- există într-un singur exemplar indiferent de numărul de instanţieri ale clasei; ca o consecinţă, dimensiunea datelor membre statice nu participă la dimensiunea nici unei instanţieri a clasei;
- pot fi referiţi prin numele clasei urmat de operatorul de rezoluţie : : şi de numele membrului dată static;

Metodele declarate static pot fi referite similar chiar în cazul în care nu există instanţieri ale clasei.

```
#include<iostream>
#include<string.h>
#include<conio.h>
using namespace std;

class Persoana {
      char *nume;
      static unsigned nrPersoane;
public:
      Persoana(char *nume) {
            if(nume != NULL) {
            this->nume =
                  new char[strlen(nume)+1];
      strcpy(this->nume, nume);

            } else {
                  this->nume= NULL;
            }
            nrPersoane++;
      };
      ~Persoana() {
```

```
            if (nume != NULL) {
                  delete[] nume;
            }
            cout<<"~Persoana()" << endl;
            nrPersoane--;
          cout << "Numar persoane ramase = "
               << nrPersoane << endl;
            _getch();
      }
      static void printNrPers() {
           cout <<"nrPers="
               << nrPersoane << endl;
      };
};
//initializare membru static
unsigned Persoana::nrPersoane = 0;

int main() {
      Persoana ionescu("ionescu");
      Persoana popescu("popescu");
      Persoana *simionescu
                  = new Persoana("simionescu");
      Persoana::printNrPers();
      delete simionescu;
      return 0;
}
```

Ieşire

```
nrPers=3
~Persoana()
Numar persoane ramase = 2
~Persoana()
Numar persoane ramase = 1
~Persoana()
Numar persoane ramase = 0
```

2. Cuvântul cheie *this*

Cuvântul cheie `this` este o variabilă locală predefinită în C++, accesibilă în corpul oricărei **metode non-statice a unui obiect** şi care nu trebuie declarată. Această variabilă este de tip pointer la obiectul curent (obiectul a cărui metodă este executată) şi conţine adresa acestui obiect. Situaţia se prezintă ca şi cum metodei i s-ar transmite implicit adresa instanţierii prin care a fost apelată.

Considerăm o clasă `X` şi o instanţiere `x` a acestei clase. Există două posibilităţi de utilizare a variabilei predefinite `this`. Acestea sunt exemplificate prin `func1()` şi `func2()` în exemplul următor:

- când se face apelul metodei `x.func1()`, lui `this` i se dă valoarea adresei lui `x (&x)` şi poate fi folosit în corpul funcţiei `func1`, după care se poate returna ca şi pointer.
- când se face apelul metodei `x.func2()`, se returnează conţinutul pointerului `this`.

```
#include <conio.h>
#include<iostream>
using namespace std;
```

```
class X
{
public:
	X* func1()
	{
		cout<<"Test1 pentru this."<<endl;
		return this;
	}

	X& func2()
	{
		cout<<"Test2 pentru this."<<endl;
		return *this;
	}
};
int main()
{
	X  x;
	X *x1 = x.func1();
	X& x2 = x.func2();
	x1->func1();
	x2.func2();
	_getch();
	return 0;
}
```

Ieşire

```
Test1 pentru this.
Test2 pentru this.
Test1 pentru this.
Test2 pentru this
```

Orice metodă non-statică a unei clase are ca prim parametru variabila this transmisă implicit.

În cadrul unei metode (mai ales în constructor), cuvântul `this` este util pentru a deosebi câmpurile clasei de parametrii cu acelaşi nume. Să urmărim următorul exemplu:

```
// exemplu: cuvantul cheie this
#include<iostream>
using namespace std;

class Complex {
	int re, im;
public:
	Complex(){re = im = 0;}
	Complex(int re, int im) {
		this->re = re;
		this->im = im;
	}
	void afisare() {
		cout <<"("<<re<<","<<im<<")"<<endl;
	}
};
int main() {
	Complex c1(5,3);
	Complex c2(2,-3);
```

```
        c1.afisare();
        c2.afisare();
        return 0;
}
```

Ieşire:

```
(5,3)
(2,-3)
```

Observăm parametrii constructorului, având acelaşi nume ca şi câmpurile clasei:

```
Complex(int re, int im) {
        this->re /*campul clasei*/
            = re /*parametrul*/;
        this->im = im;
}
```

În acest caz variabilele locale `re` şi `im` ascund câmpurile clasei. Totuşi, câmpurile pot fi accesate fără probleme folosind cuvântul cheie `this`. Utilizarea acestei tehnici oferă o mai bună claritate a codului. Dispare necesitatea de a da nume diferite pentru parametrii constructorilor şi ai metodelor de setare a câmpurilor. Tehnica se foloseşte pe larg în limbajele C++, Java şi C#.

3. Constructorul de copiere

Constructorul de copiere este un constructor special, folosit pentru a crea un nou obiect, copie a unui obiect existent. Acest constructor are un singur argument – o referinţă către obiectul ce va fi copiat. Dacă în clasă nu există constructorul de copiere scris de programator, compilatorul generează implicit un constructor de copiere. Forma generală a constructorului de copiere este următoarea:

```
NumeClasa::NumeClasa(const NumeClasa &
obiectSursa);
```

Constructorul de copiere implicit copie fiecare membru a obiectului parametru în membrul corespunzător al obiectului de iniţializat. De exemplu, pentru clasa `Complex` din exemplele noastre, compilatorul va genera următorul constructor de copiere:

```
Complex(const Complex &obiectSursa) {
        this->re = obiectSursa.re;
        this->im = obiectSursa.im;
}
```

Constructorul de copiere are un rol special în C++. El este apelat automat în următoarele situaţii:

1. La declararea unui obiect, iniţializat dintr-un alt obiect. Exemplu:

```
Complex x(3,4); /* constructorul este folosit pentru a
crea obiectul x */
Complex y(x);   /* constructorul de copiere este folosit
pentru a crea obiectul y*/
Complex z = x;  /* constructorul de copiere este folosit
pentru initializare in */
// declaratie
z = x;           /* Operatorul de atribuire (=), nu se
apeleaza constructori */
```

2. O funcţie primeşte ca parametru un obiect transmis prin valoare.
3. O funcţie returnează un obiect.

Să urmărim programul de mai jos:

```
#include<iostream>
using namespace std;
class Complex {
      int re, im;
public:
      Complex(){}
      Complex(int re, int im) {
            this->re = re;
            this->im = im;
      }

      Complex aduna(Complex c2) {
            Complex rez;
            rez.re = this->re+c2.re;
            rez.im = this->im+c2.im;
            return rez;
      }

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

int main() {
      Complex c1(5,3);
      Complex c2(2,-3);
      Complex c3;
      c3 = c1.aduna(c2);
      c1.afisare();
      c2.afisare();
      c3.afisare();
      return 0;
}
```

La apelul funcţiei `aduna()`, constructorul de copiere este apelat de 2 ori. O dată la transmiterea parametrului `c2`, şi a doua oară la returnarea rezultatului `rez`.

În total, există 3 membri generaţi de compilator în mod implicit:

1. Constructorul implicit, în cazul în care nu este definit nici un alt constructor;
2. Constructorul de copiere;
3. Operatorul de atribuire.

Dacă programatorul are nevoie ca oricare dintre aceşti trei membri să fie definit altfel decât în modul implicit, îl poate defini în forma dorită.

4. Constructorul explicit de copiere

Un caz special este atunci când clasa conţine câmpuri de tip pointer. În acest caz utilizatorul trebuie să definească constructorul de copiere în mod explicit. Constructorul de copiere explicit va aloca memoria necesară pentru câmpuri de de tip pointer, şi va iniţializa memoria alocată.

Revenim la exemplul clasei `Persoana`:

```
#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() {
            if (nume != NULL) {
                  delete[] nume;
            }
            cout<<"~Persoana()" << endl;
      }
};

void main() {
      Persoana ionescu("ionescu");
      Persoana popescu("popescu");
      Persoana popescu2=popescu;
}
```

Ieşire:

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

În constructorul de copiere se alocă memorie pentru câmpul nume, după care se copie şirul de caractere din obiectul argument.

5. Pointeri la clase

Se pot crea şi pointeri pe obiectele instanţiate din clase.

De exemplu:

```
Complex *c1, *c2;
```

`C1` şi `c2` sunt pointeri la tipul `Complex`.

În vorbirea curentă, datorită faptului că obiectele pointate sunt instanţieri ale unei clase se foloseşte şi exprimarea „pointer la clasă”.

Putem folosi operatorul special săgeată (`->`) pentru a accesa membrul unui obiect referit de un pointer. Iată un exemplu cu câteva combinaţii posibile:

```
// exemplu cu pointeri la clase
#include<iostream>
using namespace std;
class Complex {
      int re, im;
public:
      Complex(int re, int im) {
            this->re = re;
            this->im = im;
      }
      void afisare() {
            cout<<"("<<re<<","<<im<<")"<<endl;
      }
};

int main () {
      Complex a(1,2);
      Complex *b, *c;
      Complex **d = new Complex*[2];

      b = &a;
      c = new Complex(3,4);
      d[0] = new Complex(5,6);
      d[1] = new Complex(5,6);
      a.afisare();
      b->afisare();
      c->afisare();
      d[0]->afisare();
      d[1]->afisare();
      delete d[0];
      delete d[1];
      delete[] d;
      delete b;
      return 0;
}
```

Ieşire:

```
(1,2)
(1,2)
```

```
(3,4)
(5,6)
(5,6)
```

Mai jos sunt sumarizate operaţiilor posibile cu pointeri la clase:

Expresie	Semnificaţie
`*x`	obiect referit de x
`&x`	adresa lui x
`x.y`	membrul y al obiectului x
`x->y`	membrul y al obiectului referit de x
`(*x).y`	membrul y al obiectului referit de x (echivalent cu expresia anterioară)
`x[0]`	primul obiect din vectorul referit de x (echivalent cu *x)
`x[1]`	al 2-lea obiect din vectorul referit de x
`x[n]`	al (n+1)-lea obiect din vectorul referit de x

6. Exerciţii

1. Să se implementeze o clasă `Complex`, similar cu exemplele din laborator. La această clasă să se adauge o metodă `egal()`, care va realiza compararea a 2 numere complexe. Metoda va avea următorul prototip:

```
int Complex::egal(Complex c2);
```

Metoda va compara complexele `this` şi `c2` şi va returna:

1, dacă `this == c2`

0, în caz contrar

Să se scrie o metodă `Complex::citire()`, care va citi numărul complex de la tastatură.

Scrieţi un program care citeşte de la tastatură 2 numere complexe şi afişează rezultatul comparării lor.

2. Să se implementeze clasa `MultimeComplexe`, care să păstreze o mulţime de numere complexe, reprezentate de clasa `Complex`.

Clasa va avea următoarele câmpuri private:

`Complex *v` - un vector de elemente `Complex`, care va păstra elementele mulţimii.

`int dim` – numărul de elemente alocate în vectorul `v`, dimensiunea maximă a mulţimii.

`int n` – numărul de elemente a mulţimii.

Implementarea este similară cu cea a mulţimii de întregi din lab. 2, doar că elementele stocate de mulţime vor fi de tip `Complex` în loc de `int`. Pentru compararea a 2 numere complexe în scopul determinării dacă sunt egale sau diferite, se va folosi funcţia `egal()` de la problema 1.

Următoarele metode publice sunt similare cu cele ale mulţimii din lab. 2:

- constructorii, care iniţializează câmpurile private ale mulţimii;

- `void adauga(Complex)` care adaugă un element în mulţime. În cazul în care elementul deja există, mulţimea rămâne nemodificată;
- `void extrage(Complex)` care extrage un element din mulţime. În cazul în care elementul nu este prezent, mulţimea rămâne neschimbată;
- `void afisare()` care afişează mulţimea.

Folosiţi următorul program pentru a testa mulţimea:

```
int main() {
	MultimeComplexe m;
	Complex c1(2,3), c2(3,4), c3(2,-1);
	m.init();
	m.adauga(c1);
	m.adauga(c2);
	m.afisare();
	m.extrage(c1);
	m.extrage(c3);
	m.afisare();
	m.adauga(c3);
	m.adauga(c3);
	m.afisare();
	return 0;
}
```

3. Scrieţi un program care testează clasa `MultimeComplexe` cu numere citite de la consolă. Programul va rula în buclă şi va afişa la fiecare iteraţie un meniu, şi va cere utilizatorului să aleaga una din următoarele operaţii:

1 – adăugare element

2 – extragere element

0 – ieşire din program.

În cazul în care se alege 1 sau 2, urmează citirea numărului complex de la tastatură, realizarea operaţiei, şi afişarea mulţimii rezultate.