POO – C++ - Laborator 4

Cuprins

1. Clase

Clasa este o extensie a conceptului de structură din C. Prin crearea unei clase se defineşte, de fapt, un nou tip de dată care reprezintă concretizarea unui anumit concept. Ca şi structura, o clasă poate avea mai mulţi membri, care pot fi:

- variabile, care conţin datele clasei şi care se numesc **proprietăţi** sau **câmpuri**
- funcţii, care se numesc **metode**

Declararea unei clase se realizează folosind cuvântul cheie **class** şi are loc într-un fişier header. Declararea clasei presupune, de asemenea, şi declararea variabilelor şi a metodelor proprii ei. Definirea metodelor se face de cele mai multe ori separat, într-un fişier sursa, .cpp . Crearea unei clase permite integrarea atât a datelor care trebuie prelucrate, cât şi a funcţiilor şi operaţiilor ce realizează prelucrarea.

În exemplul de mai jos s-a declarat o clasă care încorporează atât dimensiunile unui dreptunghi, cât şi metodele care permit modificarea datelor şi prelucrarea lor:

dreptunghi.h

```
class Dreptunghi
{
private:
	int lungime, latime;

public:
```

```
        void SetLungime(int lung);
        void SetLatime(int lat);
        int Aria();
};
```

Atenţie! După declararea clasei (după acolada care închide clasa) se pune ; (punct şi virgulă)!

Observăm că, deocamdată, metodele din cadrul clasei au fost doar declarate. Ele urmează să fie definite într-un fişier sursă unde prima dată trebuie inclus headerul în care s-au făcut declaraţiile. Apoi, pentru metodele definite, trebuie specificată individual clasa din care fac parte, folosind operatorul de rezoluţie **: :**

dreptunghi.cpp

```
#include "dreptunghi.h"

void Dreptunghi::SetLungime(int lung)
{
        lungime = lung;
}

void Dreptunghi::SetLatime(int lat)
{
        latime = lat;
}

int Dreptunghi::Aria()
{
        return lungime * latime;
}
```

2. Obiecte

Clasele astfel declarate constituie un tip de dată. Declararea unei variabile de tipul unei clase se numeşte **instanţierea clasei**, iar variabila astfel declarată se numeşte **instanţă** sau **obiect**.

main.cpp

```
int main()
{
        Dreptunghi d; // crearea statica a unui obiect

        Dreptunghi *p; // pointer la clasa
```

```
        p = new Dreptunghi; // crearea unui obiect prin alocare dinamica
        delete p; // eliberarea memoriei

        Dreptunghi dv[10]; // un vector de obiecte alocat static

        Dreptunghi *pv;
        pv = new Dreptunghi[10]; // vector de obiecte alocat dinamic
        delete[] pv; // eliberarea memoriei

        return 0;
}
```

Accesul la membrii clasei se realizează la fel ca în cazul structurilor: se foloseşte operatorul **.** (punct) pentru obiectele definite static (variabilele obişnuite) şi operatorul **->** (săgeată) în cazul pointerilor:

```
        Dreptunghi d;

        d.SetLungime(2);
        d.SetLatime(3);
        cout << d.Aria();

        Dreptunghi *p = new Dreptunghi;
        p->SetLatime(4);
```

3. Specificatori de acces

În exemplele precedente apar cuvintele cheie **private** şi **public**. Acestea se numesc **specificatori de acces** şi, prin utilizarea lor, se decide gradul de accesibilitate al variabilelor şi metodelor clasei. Specificatorii se aplică tuturor membrilor aflaţi sub ei, până la apariţia unui nou specificator sau până la incheierea declarării clasei.

- **private** permite accesul la membrii clasei doar membrilor aceleiaşi clase.
- **public** permite accesul la membrii clasei de oriunde (membrii clasei, funcţii ce nu aparţin clasei, funcţia main())

În mod implicit, toţi membrii unei clase au modul de acces *private*.

dreptunghi.h

```
class Dreptunghi
{
        int lungime, latime;

public:
        void SetLungime(int lung);
```

```
        void SetLatime(int lat);
        int Aria();
private:
        int Perimetru();
};
```

dreptunghi.cpp

```
#include "dreptunghi.h"

void Dreptunghi::SetLungime(int lung)
{
        lungime = lung;

        /*
        membrul lungime este implicit private
        si este accesibil aici, deoarece metoda SetLungime apartine clasei Dreptunghi
        */
}
```

main.cpp

```
int main()
{
        Dreptunghi d;

        cout << d.lungime; //eroare: campul lungime este implicit private, se incearca
                                //accesarea lui din afara clasei Dreptunghi

        cout << d.Aria(); //metoda este public, poate fi accesata din afara clasei

        int p = d.Perimetru(); // eroare: metoda este private, nu se poate utiliza in
                                        // afara clasei unde a fost declarata

        return 0;
}
```

4. Constructor

Atunci când se construiesc, obiectele au nevoie să li se aloce memorie pentru membrii de tip dată şi eventual aceştia să fie iniţializaţi. Iniţializarea face obiectele operabile şi reduce posibilitatea returnării unor valori nedorite în timpul execuţiei. În lipsa iniţializării, există posibilitatea de a fi luate valori aleatoare prezente în locaţiile de memorie asignate câmpurilor.

Pentru a evita acest tip de situaţii, o clasă poate conţine o metodă specială numită ***constructor***. Constructorul are următoarele caracteristici:

- numele este acelaşi cu al clasei;
- nu are tip , deci nu returnează nimic;

- este apelat automat la începerea duratei de viaţă a oricărui obiect de tipul respectiv.

Declaraţia constructorului are următoarea formă:

```
Nume_clasa(lista parametri);
```

Constructorul este apelat când se crează un obiect al clasei. Pentru obiectul deja creat constructorul nu mai poate fi reapelat, ca orice altă metodă obişnuită.

Există o serie de constructori tipici pentru o clasă:

- **constructorul implicit**: nu are nici un parametru şi iniţializează membrii de tip dată cu valori implicite;
- **constructorul de iniţializare**: are de obicei câte un parametru pentru fiecare membru de tip dată din clasă şi îi initializează cu valorile primite ca parametri;
- **constructorul de copiere**, acesta având un regim special în cadrul clasei.

Observaţii:

1) ***Obiectele globale*** (declarate în exteriorul oricărei funcţii) au mod de alocare static şi sunt plasate în segmentul de date; durata lor de viaţă începe înainte de intrarea în funcţia `main()`.
2) ***Obiectele locale*** funcţiilor au mod de alocare auto şi sunt plasate în segmentul de stivă; durata lor de viaţă începe în momentul în care execuţia programului ajunge în locul în care sunt declarate.
3) ***Obiectele instanţiate dinamic*** sunt plasate în zona de heap; durata lor de viaţă începe în momentul alocării dinamice.

5. Constructor de iniţializare

Vom implementa clasa `Dreptunghi` din laboratorul precedent folosind un constructor de iniţializare:

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

class Dreptunghi {
      int lungime, latime;
public:
     Dreptunghi (int, int);
      int arie () {return (lungime*latime);}
};

Dreptunghi::Dreptunghi(int L, int l) {
         lungime = L;
          latime  = l;
}

int main () {
      Dreptunghi dr(10,20);
      cout << "arie: " << dr.arie() << endl;
      _getch();
      return 0;
}
```

Ieşire:
`arie: 200`

Observăm că rezultatul rulării acestui exemplu este identic cu cel din laboratorul anterior. De data aceasta am eliminat funcţiile membru `setLungime()` şi `setLatime()` şi le-am înlocuit cu un constructor care face acelaşi lucru: iniţializează valorile câmpurilor `lungime` şi `latime` cu parametrii pe care îi primeşte, adică `L` şi `l`.

Remarcăm că aceşti parametri sunt transmişi către constructor în momentul creerii obiectului:

```
Dreptunghi dr(10,20);
```

6. Supraîncărcarea constructorilor

Să ne amintim că ***semnătura unei funcţii*** reprezintă numele funcţiei împreună cu lista de parametri. În C++, două sau mai multe funcţii care au acelaşi nume dar semnături diferite se consideră că sunt supraîncărcate. ***Supraîncărcarea*** se utilizează pentru a da acelaşi nume tuturor funcţiilor care fac acelaşi tip de operaţie.

De exemplu, funcţia `abs()` care returnează valoarea absolută, se va numi la fel, indiferent dacă data de intrare este de tip `int`, `long` sau `double`.

```
int abs(int);
long abs(long);
double abs (double);
```

Atât funcţiile globale cât şi metodele pot fi supraîncărcate. Prin urmare, este admisă şi supraîncărcarea constructorilor.

7. Constructorul implicit

Atunci când într-o clasă nu există nici un constructor, compilatorul generează automat un constructor special pentru acea clasă, numit **constructor implicit**. Rolul acestui constructor este de a aloca memorie pentru datele membre. Dacă obiectele construite cu acest constructor implicit sunt globale, iniţializarea membrilor de tip dată se face cu:

- **0** dacă sunt numerice;
- **'\0'** dacă sunt de tip caracter sau şir de caractere;
- **NULL (0)** dacă sunt pointeri.

Dacă în clasă avem membri de tip referinţă, atunci
compilatorul nu generează constructorul implicit, generând o eroare de compilare care ne atrage atenţia că implementarea unui constructor care să iniţializeze referinţele este obligatorie.

Dacă obiectele sunt locale, constructorul implicit generat de compilator nu face nici o iniţializare pentru câmpuri.

Să analizăm următorul exemplu:

```
#include<conio.h>
```

```
#include<iostream>
using namespace std;

class Numar
{
	int nr;
public:
	void setNr(int n) {
		nr = n;
	}
	int getNr() {
		return nr;
	}
};

Numar nrGl;
int main()
{
	Numar nrLoc;
	cout << "Obiect global "
		"initializat implicit: "
		<< nrGl.getNr() << endl;

	cout << "Obiect local "
		"neinitializat implicit: "
		<< nrLoc.getNr() << endl;

	nrLoc.setNr(100);
	cout <<"Obiect local initializat explicit"
		" cu metoda membru: "
		<< nrLoc.getNr() << endl;

	_getch();
	return 0;
}
```

Ieşire

```
Obiect global initializat implicit: 0
Obiect local neinitializat implicit: -858993460
Obiect local initializat explicit cu metoda membru: 100
```

Observăm că deşi nu există un constructor în clasă, se pot crea obiecte, cu ajutorul constructorului implicit generat de compilator.

În acest caz, putem crea obiecte numai folosind următoarea sintaxă:

```
/*nu avem lista de initializare pentru obiect*/
Numar nrLoc;
```

Atunci când un obiect este iniţializat folosind constructorul implicit, trebuie folosită declaraţia fără paranteze. De exemplu:

```
Numar nr1;   // declaratie corecta
Numar nr2(); // declaratie gresita
```

Constructorul implicit poate fi implementat şi de către programator, acesta putând să facă alocări şi / sau iniţializări pentru membrii de tip dată. Dacă programatorul implementează un constructor pentru o clasă, compilatorul nu mai generează constructorul implicit.

Dacă într-o clasă, programatorul implementează un **constructor fără parametri**, acesta se numeşte tot ***constructor implicit***.

```
class Numar
{
      int nr;
 public:
      Numar() {
      // constructor implicit vid
      }

      void setNr(int n)  {
            nr = n;
      }
};
```

Să analizăm următorul exemplu şi să observăm modul în care se iniţializează câmpurile obiectelor instanţiate.

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

class Numar
{
      int nr;
public:

      Numar() {
     //constructor implicit cu initializare
            nr = 10;
      }
      void setNr(int n) {
            nr = n;
      }
      int getNr() {
            return nr;
      }
};

Numar nrGl;

int main()
{
      Numar nrLoc;
      cout << nrGl.getNr() << endl;
      cout << nrLoc.getNr() << endl;

      nrLoc.setNr(100);
      cout << nrLoc.getNr() << endl;
```

```
	_getch();
	return 0;
}
```

Ieşire

```
10
10
100
```

Un caz de ambiguitate care poate să apară la implementarea constructorului implicit este prezentat în următorul exemplu:

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

class Numar
{
	int nr;
public:
	//constructor implicit vid
	Numar() {}

	//constructor cu parametru predefinit
	Numar(int n=0) {
		nr = n;
	}
	void setNr(int n) {
		nr = n;
	}
};

int main()
{
	Numar nr;
	_getch();
	return 0;
}
```

La crearea obiectului `nr`, ambiguitatea apare din cauză că, în acestă situaţie, compilatorul poate apela ambii constructori care au fost definiţi. Pentru rezolvarea ambiguităţii este recomandată eliminarea constructorului implicit, deoarece constructorul de iniţializare cu parametru predefinit poate fi utilizat ca şi constructorul implicit.

Să studiem următorul exemplu care utilizează supraîncărcarea constructorilor:

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

class Dreptunghi {
	int lungime, latime;
public:
```

```
      //constructor implicit
      Dreptunghi ();

      //constructor cu parametri
      Dreptunghi(int,int);
      int arie() {return lungime * latime;}
};

Dreptunghi::Dreptunghi () {
      lungime = 10; latime = 10;
}

Dreptunghi::Dreptunghi(int L, int l) {
     lungime = L;
     latime = l;
}

int main () {
      Dreptunghi dr1 (10,20);
      Dreptunghi dr2;
      cout << "arie dr1: "
          << dr1.arie() << endl;
      cout << "arie dr2: "
          << dr2.arie() << endl;
      _getch();
      return 0;
}
```

Ieşire

```
arie dr1: 200
arie dr2: 100
```

8. Destructor

Destructorul unei clase este complementarul constructorului şi are următoarele caracteristici:

- numele este acelaşi cu al clasei, precedat de caracterul '~';
- nu are tip, deci nu returnează nimic şi nu are argumente, deci nu poate fi supraîncărcat;
- este apelat automat la terminarea duratei de viaţă a obiectelor nealocate dinamic. Pentru obiectele alocate dinamic apelarea destructorului trebuie să se facă explicit;
- trebuie să dezaloce zonele de memorie alocate dinamic pentru membrii de tip dată.

Declaraţia destructorului are următoarea formă:

```
~Nume_clasa();
```

Observaţii:

Există situaţii când nu este necesară implementarea destructorului:

- atunci când în clasă nu există date membre alocate dinamic, ele având tipuri fundamentale. Dacă nu se implementează destructorul, compilatorul va genera un ***destructor implicit***, pe care îl apelează când se încheie durata de viaţă a obiectului.

- atunci când în clasă există tipuri abstracte de dată (instanţieri de structuri sau clase). Dacă clasa obiectului imbricat are un destructor explicit, compilatorul îl apelează pentru distrugerea acestui obiect. În caz contrar, compilatorul apelează destructorul implicit din clasa obiectului imbricat.

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

class Numar
{
	int *nr;
public:

	Numar() {
		nr = new int;
	}
	~Numar(){
		cout << "~Numar()" << endl;
		delete nr;
		_getch();
	}
	void setNr(int n) {
		*nr = n;
	}
};

class Intreg{
	Numar n;
public:
	Intreg(int nr){
		n.setNr(nr);
	}
};

int main() {
	Intreg i(10);
	return 0;
}
```

Ieşire

```
~Numar()
```

Durata de viaţă a obiectelor se termină:

- pentru obiectele globale cu mod de alocare static, la ieşirea din funcţia `main()`;
- pentru obiecte locale cu mod de alocare auto, la ieşirea din blocul în care au fost declarate.
- pentru obiectele cu alocare dinamică, la apelarea explicită a operatorului `delete`.

9. Exerciţii

1. Să se implementeze clasa `Multime`, care reprezintă o mulţime de întregi.

Clasa va conţine următoarele câmpuri private:

- `int *date` – vectorul de numere. Se va aloca în constructor şi se va dezaloca în destructor;
- `int dim` – dimensiunea vectorului `date`; totodată reprezintă numărul maxim de elemente din mulţime;
- `int n` – numărul curent de elemente din mulţime; în orice moment de timp, elementele mulţimii vor fi primele `n` elemente din vectorul `date`.

Pe parcursul existenţei mulţimii, numărul `n` şi elementele din `date` se pot modifica, dar câmpul `dim` rămâne neschimbat.

Membrii publici:

- constructorul implicit care iniţializează câmpurile private ale mulţimii; dimensiunea maximă a mulţimii va fi o valoare oarecare prestabilită;
- constructorul cu un parametru pentru câmpul `dim`, reprezentând dimensiunea maximă a mulţimii;
- destructorul care va elibera memoria alocată dinamic;
- metoda `void adauga(int)` care adaugă un element în mulţime; în cazul în care elementul deja există, mulţimea rămâne nemodificată; în cazul în care vectorul `date` este plin, se va afişa un mesaj de eroare;
- metoda `void extrage(int)` care extrage un element din mulţime; în cazul în care mulţimea nu conţine elementul, ea rămâne neschimbată;
- metoda `void afisare()` care afişează mulţimea.

Folosiţi următorul program pentru testarea mulţimii:

```
int main() {
	Multime m(10);
	m.adauga(4);
	m.adauga(3);
	m.afisare();
	m.extrage(4);
	m.extrage(4);
	m.afisare();
	m.adauga(9);
	m.adauga(2);
	m.adauga(2);
	m.afisare();
	_getch();
	return 0;
}
```

2. Să se implementeze clasa `Stiva` având următoarele metode:

- constructorul implicit care iniţializează câmpurile private ale stivei;
- metoda `push` care adaugă un element în stivă;
- metoda `pop` care extrage un element din stivă;
- metoda `top` care returnează vârful stivei, fără să îl extragă;

- metoda `print` care afişează stiva.

Folosiţi următorul program pentru testarea stivei:

```
int main() {
	Stiva s;
	s.push(4);
	s.push(3);
	cout << s.top() << endl;
	s.push(9);
	cout << s.pop() << endl;
	s.push(2);
	s.print();
	_getch();
	return 0;
}
```