Lab 3 – P2

Cuprins

1. Clase

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

- variabile, care conțin datele clasei și care se numesc **proprietăți** sau **câmpuri**
- funcții, care se numesc **metode**

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

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

dreptunghi.h

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

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

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

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

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 numește **instanțierea clasei**, iar variabila astfel declarată se numește **instanță** 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 folosește operatorul **.** (punct) pentru obiectele definite static (variabilele obișnuite) și 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** și **public**. Acestea se numesc **specificatori de acces** și, prin utilizarea lor, se decide gradul de accesibilitate al variabilelor și metodelor clasei. Specificatorii se aplică tuturor membrilor aflați sub ei, până la apariția unui nou specificator sau până la încheierea declarării clasei.

- **private** permite accesul la membrii clasei doar membrilor aceleiași clase.
- **public** permite accesul la membrii clasei de oriunde (membrii clasei, funcții ce nu aparțin clasei, funcția 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 incearcă 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 declarată

	return 0;
}
```

4. Constructori si destructori

Constructorii sunt metode speciale din cadrul claselor, care se apelează automat la crearea obiectelor. Ei poartă aceeași denumire ca și clasa, și nu au tip de return.

4.1. 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 acestuia 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.

4.2. Constructorul fără argumente

Acest constructor apare declarat explicit în cadrul clasei. Nu are parametri, și, cel mai adesea, în interiorul său se realizează alocări de memorie și se dau valori implicite câmpurilor clasei.

4.3. Constructorul de iniţializare

Acest constructor are o lista de parametri pe care îi utilizează la inițializarea câmpurilor clasei. Este o variantă supraîncărcată a constructorului fără argumente.

dreptunghi.h

```
class Dreptunghi
{
	int lungime, latime;

public:
	Dreptunghi(); // constructor fara argumente
	Dreptunghi(int lat, int lung); // constructor de initializare

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

dreptunghi.cpp

```
#include "dreptunghi.h"

Dreptunghi::Dreptunghi() //definitia constructorului fara argumente
{
	//campurilor li se dau valori implicite, prestabilite
	lungime = 0;
	latime = 0;
}

Dreptunghi::Dreptunghi(int lung, int lat) //definitia constructorului de
                                          //initializare
{
	//campurile sunt initializate cu valorile parametrilor constructorului
	lungime = lung;
	latime = lat;
}
```

main.cpp

```
int main()
{
	Dreptunghi a; //se apeleaza automat constructorul fara argumente

	Dreptunghi b(2, 3); // se apeleaza automat constructorul de initializare

	return 0;
}
```

Ca alternativă, în cadrul constructorilor, câmpurile pot lua valori și prin intermediul **listelor de inițializare**, în acest caz nemaifiind nevoie de inițializarea lor în corpul constructorului:

```
Dreptunghi::Dreptunghi():lungime(0), latime(0)
{
}

Dreptunghi::Dreptunghi(int lung, int lat):lungime(lung), latime(lat)
{
}
```

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

```
Dreptunghi a; //corect
Dreptunghi a(); //gresit
```

Un caz de ambiguitate poate apărea în unele situații în care constructorii au parametri cu valori implicite. Considerăm următorul exemplu:

```
class Numar
{
	int nr;
public:
	Numar()
	{
		nr = 0;
	}
	Numar(int n = 1)
	{
		nr = n;
	}
};
```

Presupunem că se declară un obiect `Numar a;` . În această situație, compilatorul nu va ști pe care dintre cei doi constructori să-i utilizeze. Soluția este eliminarea unuia dintre constructori, cel mai adesea a celui fără parametri.

Destructorii sunt metode care se apelează automat la sfârșitul duratei de viață a obiectelor (de exemplu, la ieșirea din funcția unde au fost create obiectele, la încheierea execuției aplicației, la apelarea operatorului **delete** în cazul obiectelor alocate dinamic etc.). Au aceeași denumire ca și clasa, precedată de simbolul **~** (tilda). Destructorii nu au parametri și nici tip de return.

Ca și în cazul constructorilor, în absența unui destructor definit în prealabil compilatorul generează automat un destructor implicit.

Destructorii definiți explicit se utilizează cel mai adesea la eliberarea memoriei alocate dinamic în cadrul clasei.

```
class Dreptunghi
{
	int lungime, latime;

public:
	Dreptunghi();
	Dreptunghi(int lat, int lung);
	~Dreptunghi(); //destructor
```

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

5. Membri de tip pointer

În cazul în care într-o clasă apar câmpuri de tip pointer, alocarea, respectiv dealocarea memoriei trebuie făcută explicit. Aceasta lucru se realizează de obicei în constructori (alocarea), respectiv destructori (dealocarea).

Consideram clasa **Persoana**, care are un câmp **nume** de tip **char*** (pointer la char). Așadar, câmpul **nume** va servi la salvarea unui șir de caractere. Înainte de specificarea acestui șir însă, trebuie alocată memorie pornind de la pointerul **nume**, lucru pe care-l vom realiza în constructor. La sfârșitul duratei de viață a obiectului de tip Persoana, memoria alocata nu se eliberează automat, fiind necesară dealocarea deliberată, într-un destructor definit în mod explicit.

persoana.h

```
class Persoana
{
        char *nume; //camp de tip pointer la char
        int varsta;
public:
        Persoana(); //constructor fara argumente
        Persoana(char *n, int v); //constructor de initializare
        ~Persoana(); //destructor
        void Afisare();
};
```

persoana.cpp

```
#include "persoana.h"

Persoana::Persoana()
{
        nume = 0;
        varsta = 0;
}

Persoana::Persoana(char *n, int v)
{
        int lungimeNume = strlen(n);
        nume = new char[lungimeNume + 1]; //alocare dinamica pentru membrul de tip
                                          //pointer
        strcpy_s(nume, strlen(n) + 1, n);
        varsta = v;
}

Persoana::~Persoana()
{
        if(nume != 0)
                delete[] nume; //daca s-a realizat alocare,
                               // memoria trebuie eliberata explicit
}

void Persoana::Afisare()
{
        cout << nume << " are " << varsta << " ani." << endl;
}
```

main.cpp

```
int main()
{
	Persoana p1; //se apeleaza constructorul fara argumente
	Persoana p2("Vasile", 20); //se apeleaza constructorul cu argumente

	Persoana *p3 = new Persoana("Ion", 30); // se apeleaza constructorul cu
	                                        //argumente

	p2.Afisare();
	p3->Afisare();

	delete p3; // se apeleaza destructorul lui p3

	return 0;
	//la finalul executiei aplicatiei se apeleaza destructorii lui p1, p2
}
```

6. Exerciții

Implementați o clasă **Multime** cu următorii membri:

- **nrElem**, de tip intreg
- **elem**, de tip pointer la int
- constructor fără argumente, care inițializează **nrElem** cu 10 si vectorul **elem** cu numerele 0, 1, 2, ..., 9
- constructor de inițializare, care primește ca parametri un pointer la int și un număr întreg, inițializând câmpurile clasei pornind de la acești parametri
- constructor de inițializare, care primește ca parametri două numere întregi. Primul va fi utilizat pentru inițializarea numărului de elemente, iar al doilea, numit **val**, va servi la inițializarea elementelor din **elem** cu valorile **val, val + 1, ..., val + nrElem -1**
- destructor
- metodă de afișare a elementelor mulțimii
- metodă de căutare a unei valori printre elementele mulțimii

a) Testați toate metodele definite anterior

b) Definiți două obiecte de tip Multime. Determinați intersecția si reuniunea celor două mulțimi; calculați și afișați rezultatele utilizând alte două obiecte de tip Multime.