# Lab 3 – P2

## Cuprins



## 1. Clase

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

- variabile membre, care conțin datele reprezentative ale clasei și care se numesc **proprietăți** sau **câmpuri**
- funcții membre, care se mai numesc și **metode**. Cu ajutorul acestor funcții se manipulează variabilele membre ale clasei.

Atât variabilele membre cât și funcțiile membre pot fi accesate și utilizate **numai folosind o instanță** a clasei (declararea unei variabile de tipul clasei). Un obiect este o instanță a clasei.

Declararea unei clase se realizează folosind cuvântul cheie `class` și se face, de obicei, 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, cu extensia .cpp .

În exemplul de mai jos s-a declarat o clasă care conține atât dimensiunile unui dreptunghi (lațime, lungime), cât și metodele care permit modificarea datelor (funcțiile Set) și prelucrarea lor (calculul Ariei):

dreptunghi.h

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

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

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

Metodele din cadrul clasei au fost doar declarate. Ele trebuie să fie definite într-un fișier sursă. În acest fișier sursă trebuie inclus header-ul în care s-a făcut declararea clasei. Pentru a specifica faptul că o funcție definită face parte dintr-o clasă, se folosește operatorul de rezoluție **::** . Pentru funcțiile declarate în clasa `Dreptunghi`, definițiile sunt:

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 declarate constituie un tip nou de dată, definit de utilizator. Declararea unei variabile de tipul unei clase se numește **instanțierea clasei**, iar variabila se numește **instanță** sau **obiect**.

main.cpp

```
int main()
{
	Dreptunghi d; // declararea unui obiect alocat static

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

	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. Membrii clasei nu pot fi accesați decât prin intermediul unui obiect de tipul clasei. Cu alte cuvinte, funcția `Arie()` nu exista și nu poate fi accesată dacă nu există un obiect de tipul clasei `Dreptunghi`. Metodele clasei `Dreptunghi` sunt accesate, pentru obiectul `d`, în modul următor:

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

## 3. Specificatori de acces

La declararea clasei `Dreptunghi` apar cuvintele cheie `private` și `public`. Acestea se numesc **specificatori de acces.** Prin utilizarea lor, se decide dreptul de acces către variabilele și metodele clasei. Specificatorii de acces se aplică tuturor membrilor ce se definesc după ei, până la apariția unui nou specificator sau până la încheierea declarării clasei. Mai există și un al treilea specificator de acces, `protected`. Acesta va fi detaliat în laboratoarele următoare, pentru că este strâns legat de conceptul de moștenire, ce va fi introdus ulterior.

Cei doi specificatori de acces folosiți până acum au următoarele proprietăți:

- `private` : membrii privați pot fi accesați numai de membrii aceleiași clase;

- `public`: membrii publici sunt accesibili de oriunde din domeniul de vizibilitate al obiectului.

În cadrul unei clase, specificatorul de acces implicit este `private`.

Considerăm exemplul următor:

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 in funcția
SetLungime, deoarece aceasta apartine clasei Dreptunghi
	*/
}
```

main.cpp
```
int main()
{
	Dreptunghi d;

	cout << d.lungime; //eroare: variabila lungime este implicit private,
			//se incearcă accesarea ei din afara clasei Dreptunghi

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

	int p = d.Perimetru(); // eroare: metoda este privata, nu se poate
			//utiliza in afara clasei
	return 0;
}
```

# 4. Constructori si destructori

**Constructorii** sunt metode speciale din cadrul claselor, care se apelează automat la crearea obiectelor. Aceștia sunt funcții speciale, care au același nume ca și clasa, dar nu au tip de return.

## 4.1. Constructorul implicit

Atunci când într-o clasă nu există niciun constructor, compilatorul generează automat un constructor special pentru acea clasă, numit **constructor implicit (*eng.* default constructor)**. 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 datele membre.

## 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 listă 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. Exerciţii

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

- **nrElem**, de tip intreg
- **elem**, un vector de numere intregi alocat static
- 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.