PROGRAMARE OBIECT-ORIENTATA

LABORATOR 1

1.1. REMINDER: VARIABLE vs. FUNCTII IN PROGRAMARE

Incepem discutia in acest prim laborator despre notiunile de baza cu care vom lucra si cu care studentul a este deja familiarizat de la disciplinele "Programarea calculatoarelor" si "Structuri de date si algoritmi". Este vorba de notiunea de variabila si apoi de functie. Variabila se refera la o cantitate de date in timp de functia are rolul de descrie modul cum prelucram variabilele.

O **variabila** are urmatoarele componente:

- Tip de data (int, float, double, bool, char, string, etc)
- Nume variabila (exista anumite reguli de denumire invate la PC)
- Adresa (locatia din memorie care stocheaza acea variabila)
- Valoare (respecta tipul variabilei deci are un anume format)

Nota ! Regulile de denumire variabile sunt urmatoarele:

- Trebuie sa contina intre 1 si 255 de caractere
- Primul caracter trebuie sa fie o litera din alphabet sau un underscore (_)
- Dupa primul caracter pot exista atat litere cat si cifre si underscore (_)
- Sunt case sensitive
- Nu sunt permise spatii sau caractere special
- Nu sunt permise cuvinte cheie (rezervate) din C/C++

O **functie** are urmatoarele componente:

- Tip de data returnat (daca nu returneaza nimic este void , cu anumite exceptii)
- Nume functie (exista si aici anumite reguli de denumire , vezi PC)
- Parametri (fiecare parametru are tip de data)
- Corpul functiei (contine o serie de instructiuni)

Important : la functii gruparea primelor 3 elemente (tipul de data returnat, numele si parametri) poarta denumirea de **semnatura** sau **prototip**. Denumirea de prototip (in engleza: prototype) se va regasi si in erorile oferite de compilator daca programul scris nu are sintaxa corecta)

Tot de la disciplina PC reamintim si diferenta intre operatiunea de declaratie si cea de definire pentru variabile si functii. Astfel avem urmatoarele sintaxe:

Pentru variabile:

- **declaratia** inseamna sa ii precizam tipul de data si numele dupa care scriem ; adica:

```
string nume;
```

- **definirea** inseamna sa avem toate elementele cunoscute , inclusiv valoarea:

```
string nume = "un text";
```

- de precizat este ca **adresa** este alocata implicit de compilator

Pentru functii:

- **declaratia** inseamna sa ii precizam semnatura sau prototipul adica tipul de data , numele si parametri , urmat de ;

```
string functie20(double x, int y);
```

- **definirea** inseamna sa avem toate elementele cunoscute , inclusiv corpul functiei :

```
string functie20(double x, int y)
{
    Instructiune1;
    Instructiune2;
}
```

Important: in toate programele pe care le scriem , recomandat este ca instructiunile din interiorul functiilor sa fie indentate la dreapta folosind tasta TAB; astfel observam si daca parantezele acolade se deschid si se inchid corespunzator. Intotdeauna numarul de paranteze acolade trebuie sa fie in numar par pentru a ne asigura ca fiecare paranteza care se deschide are perechea corespunzatoare pentru inchidere.

1.2. CONCEPTUL DE "CLASA" vs "STRUCTURA DE DATE"

La disciplina SDA, s-a discutat conceptul de structura de date care continea tipuri hibride de date dar care aveau proprietate: toate datele din interiorul structurii erau publice (accesibile) inafara structurii.

In programarea orientata pe obiecte ne dorim sa lucram cu un alt concept numit "clasa" care de fapt isi doreste exact contrariul, datele declarate in interiorul clasei sa fie publice doar in interiorul clasei(in corpul clasei) dar private(inaccesibile inafara clasei).

Sintaxa de definire a unei clase este urmatoarea:

```
class nume_clasa
{

     Declaratii si/sau definitii de date membre

};
```

Atentionari speciale:

- ✓ **Cuvantul cheie *class* se scrie mereu doar cu litere mici (inclusiv litera c)**
- ✓ **Numele clasei este case-sensitive**
- ✓ **Dupa numele clasei nu se pun paranteze rotunde, acest lucru se intampla doar la functii**
- ✓ **Dupa paranteza de inchidere de clasa apare intotdeauna caracterul ; (punct si virgula)**

1.3. ELEMENTELE UNEI CLASE – DATELE MEMBRE : ATRIBUTE SI METODE. OBIECTELE CLASEI.

Prin intermediul unei *Clase* ne dorim sa construim *tipuri noi de date* , inexistente pana in momentul acela in limbajul C++. Cuvantul cheie prin care creem o clasa este "*class*" , acesta nefiind permis a fi folosit ca nume de variabila pentru ca este un cuvant rezervat. Tipul nou definit de data va constitui tipul obiectelor create in program. ***Obiectele*** **sunt instante ale unei clase sau exemplare ale unei clase. Putem crea oricate obiecte (exemplare) de tipul clasei definite intr-un program.**

Exemple de obiecte: pentru clasa persoana , putem avea obiectul Popescu, pentru clasa Calculator , obiectul se poate numi HP, iar pentru clasa Dreptunghi , obiectul se poate numi D1.

O clasa are doua elemente principale care se numesc **DATE MEMBRE** : **Atribute** si **Metode**
Datele membre sunt acele variabile sau functii care au cel putin declaratia scrisa in interiorul clasei. Definitia acestora se poate scrie atat in clasa cat si in exteriorul ei dar folosind un operator special (operator de rezolutie sau scope resolution operator : :)

Atributele pot fi identificate cu variabilele si au aceleasi proprietati cu cele de la PC (inclusiv cele prezentate anterior in aceasta lucrare).
Metodele pot fi identificate cu functiile si asemanator au aceleasi proprietati cu functiile de la PC.
Metodele sunt de 2 tipuri :

- **Metode standard pe care orice clasa le poate avea:**
  - ✓ **Setteri**
  - ✓ **Getteri**
  - ✓ **Constructori**
  - ✓ **Destructori**
- **Metode specifice doar pentru o anume clasa (vor fi prezentate exemple mai jos)**

- **Atributele** reprezinta insusirile sau proprietatile pe care le poate avea tipul nou de data pe care il definim. Atributele sunt de fapt variabile cunoscute din limbajul C++ care trebuie sa prezinte un tip si un nume(int greutate, double real, string nume…etc) . Exemple: clasa complex are ca atribute : partea reala si partea imaginara, o alta clasa persoana poate avea o serie de atribute

precum: nume , prenume, varsta, greutate

- **Metodele** reprezinta actiunile posibile care pot fi realizate pe baza atributelor. Metodele sunt identificate in programarea procedurata ca fiind functii care au bineinteles un tip de data, un nume, un corp al functiei (corpul descrie cum lucreaza functia) si eventual o valoare returnata (un functie de tipul de date al functiei).

Pentru orice clasa putem scrie o serie de **metode standard**: Metode de tip "***setter***" prin care atribuim indirect valori atributelor si metode de tip "***getter***" prin care extragem valori ale atributelor. Ordinea fireasca este de a folosi intai un setter si apoi un getter.

Metodele de tip setter vor avea tipul void pentru ca nu returneaza nimic dar vor avea un argument, metodele de tipul getter vor avea tipul de date pe care il returneaza, exemplu: getterul pentru partea reala va returna un double, getterul pentru nume va returna un string, getterul pentru varsta va returna un int, etc…

Compilatorul va deduce ca e vorba de un getter sau un setter in primul rand dupa numele acestuia si apoi dupa prezenta sau absenta argumentului de functie. Daca metoda are argument inseamna ca va apela setterul , daca metoda nu are argument va apela getterul . Conceptul care sta la baza acestui comportament se numeste Polimorfism.

O serie aparte tot de metode stardard este reprezentata de *constructori*. Orice clasa are un constructor implicit care este apelat intotdeauna la crearea unui obiect . Constructorul implicit poate fi realizat de compilator pentru noi desi nu va insera cod suplimentar vizibil in program. Insa constructor implicit putem scrie si noi ca programatori . In acest ultim caz compilatorul nu va mai crea un constructor automat. Prin constructorul implicit ne propunem sa atribuim valori default atributelor in momentul cand cream un obiect dar nu mai dorim sa accesam si metoda setter pentru acel obiect.

De fapt sunt mai multe metode de a face ca atributele sa primeasca valori:

- **Atribuire directa** : nu ne dorim acest lucru in programarea orientata pe obiecte deoarece dorim sa protejam atributele pentru a nu primi valori nedorite. (conceptul se numeste **incapsularea datelor**) . Atribuirea directa se folosea in programarea procedurala.
- **Atribuire indirecta prin constructor implicit**. Metoda aceasta o folosim in momentul cand nu dorim sa mai creem o metoda setter si dorim ca toate obiectele de acel tip al clasei sa primeasca valori default.
- **Atribuire indirecta prin setter**: este de preferat a fi folosita ca o alternativa la constructorul implicit deoarece in metoda setter putem face verificari suplimentare(filtrari) cu privire la valorile date atributelor. Exemplu: daca dorim sa dam valori unui atribut de tip numitor de fractie, atunci in metoda setter trebuie sa testam daca nu cumva valoarea trimisa atributului numitor este zero (o fractie nu poate avea numitorul zero- caz de nedeterminare).
- Atribuire directa prin alte tipuri de constructori

Mai sunt apoi o ultima parte de metode dar acestea sunt specifice fiecarei clase in parte. De exemplu clasei Persoana ii putem asocia metodele: Mananca(), Dieta(), ProfilPersoana(). Clasei Pistol ii putem asocial mtodele: IncarcaGloante(), Trage(), PunePiedica()..etc.

1.4. CONTROLUL ACCESULUI LA ATRIBUTE SI METODE: MODIFICATORI DE ACCES

Atat atributelor si metodelor li se pot asocia modificatori de acces , acesta pot fi:

- **public** (datele sunt accesibile de oriunde)
- **protected** (datele sunt accesibile doar in interiorul clasei si in clasele derivate, dar inaccesibile in alte clase sau in main())
- **private** (datele sunt accesibile doar in interiorul clasei si inaccesibile in orice alta zona din program)

In general, atributele le vom seta de tip private pentru ca dorim sa nu fie modificabile in mod direct,

Iar metodele le vom seta pe public pentru ca dorim sa fie accesibile in main . Prin intermediul metodelor vom modifica atributele.

Daca ar fi sa facem o paralela cu ce se intampla la structuri de date , acolo datele din interiorul unei structuri puteau fi accesate implicit din exteriorul structurii, aceasta inseamna ca datele membre ale structurii sunt date implicit publice.

La clase insa situatia este la polul opus. **Implicit toate datele membre ale unei clase sunt date private** ; **deci nu pot fi accesate din exteriorul clasei decat daca le asociem modificatori de acces**. In lucrarea curenta ne dorim sa vizualizam efectul modificatorului de acces "*public*" , in felul urmator: atributele ne dorim sa ramana private pentru a le proteja impotriva modificarii directe , iar metodele de tip setter, getter dar si constructorul implicit ne dorim sa le facem publice. Declaratia pentru setter , getter si constructor o vom face in interiorul clasei, folosind modificatorul de acces public, iar definirea acestora o vom face din laboratorul 2, inafara clasei, pentru ca public ne permite acest lucru.

Ne propunem asadar sa construim o prima clasa care modeleaza un numar complex.

Ca si atribute lucram cu o parte reala ***_re*** si o parte imaginara ***_im*** , care nu vor avea modificator de acces alocat deci vor fi private , neaccesibile dinafara clasei. Preferam sa prefixam caracterul _ inainte de numele atributelor pentru a nu le confunda cu numele metodelor . Asadar metodele se vor numi Re() si Im(). Vom avea cate un setter si un getter pentru fiecare din cele doua atribute deci 4 metode. O a cincea metoda va fi constructorul implicit.

1.5 INCAPSULAREA DATELOR CA METODA DE PROTECTIE A ATRIBUTELOR

Am vorbit anterior de modificatori de acces care pot controla acces la datele membre. Daca vorbim strict de atribute, atunci , desi putem folosi si atribute publice, preferam sa folosim private deoarece acestea trebuie sa primeasca valori doar prin intermediul metodelor din clasa. In acest fel suntem siguri ca nu introducem valori eronate; (Exemplu: greutatea unei persoane nu trbeuie sa fie numar negativ). Pentru a seta greutatea , in metoda setter testam intai daca greutatea este numar pozitiv. Daca este negativ, atunci afisam un mesaj de eroare, daca este pozitiv facem atribuirea valorii.

1.6 MOSTENIREA CA METODA DE REUTILIZARE EFICIENTA A DATELOR

In programarea procedural (care nu foloseste clase si obiecte) putem refolosi codul deja scris utilizand functii pe care le putem apela cu diversi parametri astfel incat sa nu mai rescriem acelasi cod in mai multe zone din program.
In programarea orientata pe obiecte, putem refolosi codul scris in clase daca folosim conceptul de mostenire. Astfel la mostenire avem doua concepte, o clasa de ***BAZA***, care are atribute si metode si o clasa **DERIVATA** (sau mai multe clase derivate) care pot folosi atributele si metodele din claza de baza dar pot avea si datele membre proprii.
Astfel se pune problema care date din clasa de baza sunt accesibile in clasele derivate si care sunt disponibile si inafara claselor derivate (in functia main).
Daca datele sunt declarate publice, atunci sunt accesibile de oriunde. Reamintim ca este interzis a declara atribute publice.
Daca datele sunt declarate protected atunci sunt accessible in clasa de baza si in clasele derivate din aceasta.
Daca datele sunt private inseamna ca nu le putem accesa decat in clasa de baza.

1.7 SPATIILE DE NUME SI OPERATORUL DE REZOLUTIE

Daca ne imaginam o situatie in care doua variabile sau doua functii au acelasi nume, acest lucru ne impiedica sa le diferentiem si astfel vom primi eroare la compilare.

Solutia la aceasta problema consta in folosirea de spatii de nume (namespaces) cu nume diferite in care sa plasam variabilele cu acelasi nume , in acest mod ele pot fi diferentiate.

Clasele pot fi considerate spatii de nume , deci putem trage concluzia ca putem avea metode cu acelasi nume dar in clase diferite fara sa primim eroare la compilare.

Pentru a transmite compilatorului din ce spatiu de nume dorim sa accesam o functie va trebui sa folosim operatorul de rezolutie (scope resolution operator) care este “ **: :** ” (4 puncte)

Urmatorul cod exemplifica pe caz general folosirea spatiilor de nume:

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

// primul spatiu de nume
namespace spatiu1
{
   void functie()
   {
      cout << "Functia din spatiul 1" << endl;
   }
}

// al doilea spatiu de nume
namespace spatiu2
{
   void functie()
   {
      cout << "Functia din spatiul 2" << endl;
   }
}

int main ()
{
   // Apelam functia din spatiul 1.
   spatiu1::functie();

   // Apelam functia din spatiul 2.
   spatiu2::functie();

   return 0;
}
```

Rezultatul executiei acestui program va fi:

```
Functia din spatiul 1
Functia din spatiul 2

--------------------------------
Process exited after 0.1943 seconds with return value 0
Press any key to continue . . .
```

Ceea ce dovedeste ca daca precizam spatiul de nume din care sa accesam functia , compilatorul va afisa mesajul din acel spatiu de nume. Pe caz general compilatorul va executa doar codul din functia al carui spatiu de nume este specificat cu operatorul de rezolutie.

1.8 OPERATORUL DE ACCES LA DATELE MEMBRE

Operatorul discutat la paragraful anterior (operatorul de rezolutie) ne ajuta doar sa specificam apartenenta unei functii la un spatiu de nume si implicit sa apelam o functie dintr-un anumit spatiu de nume.

Dorim acum sa explicam un caz particular in care functia pe care dorim sa o apelam (sa o lansam in executie) este membra a unei clase . Reamintim ca o functie este membra a unei clase daca cel putin declaratia este prezenta in corpul clasei. Pentru acest caz apelul nu va fi posibil doar prin simpla scriere in main() a numelui de functie si a parametrilor , dar nici prin folosirea operatorului de rezolutie si a numelui de clasa deoarece regula de baza (corecta) pentru a apela o functie este utilizarea unui obiect astfel:

```
OBIECT . FUNCTIE();// OBIECT APELEAZA FUNCTIE()
```

Folosim deci operatorul punct si nu operatorul de rezolutie. O cerinta obligatorie este ca obiectele are apeleaza functiile de tip date membre sa fie anterior create folosind sintaxa:

```
NUME_CLASA NUME_OBIECT;// OBIECTUL NUME_OBIECT ESTE DE TIPUL NUME_CLASA
```

Exemplul efectiv de creare obiect poate fi considerat daca cream clasa (noul tip de data)CALCULATOR in urmatoarea sintaxa:

```
CALCULATOR HP; //HP este de tipul CALCULATOR (CALCULATOR este numele clasei)
```

1.9. IMPLEMENTAREA CLASELOR COMPLEX SI PERSOANA

Ne dorim sa implementam initial clasa complex in care modelam un numar din spatiul numerelor complexe avand o parte reala si o parte imaginara. Cream cate un Setter si cate un Getter pentru fiecare atribut .

Pentru numele de Setter si Getter nu exista vreo restrictie (doar restrictiile de la functii) dar este recomandat sa folosim template-ul “Set_Nume_Atribut” si “Get_Nume_Atribut” pentru a evidentia despre ce atribut este vorba.

In Exemplul urmator metodele sunt declarate in corpul clasei si de asemenea sunt si definite. Nu folosim momentan operatorul de rezolutie.

Exista in aceasta clasa si un constructor implicit care are rolul de a aloca valori implicite pentru obiectele pentru care nu dorim sa dam valori prin Setter. Alocarea de valori prin Setter va suprascrie valorile preluate din constructor in mod implicit.

In functia main vom crea mai multe obiecte care vor accesa functiile Setter si Getter dar vom crea si un obiect care nu acceseaza Setteri ci doar Getteri, pentru a observa ca valorile atributelor pentru acel obiect sunt preluate din constructorul implicit.

Codul programului propus este urmatorul :

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

class Complex //creez un nou tip de date numit Complex
{
	double _re; // declar cele doua atribute private _re si _im;
	double _im; // le pun _ in fata ca sa le deosebesc de metodele cu acelasi
nume
	//un atribut sau metoda care nu are modificator de acces, se considera
implicit private
public:
	void Re(double r)//definesc metoda de tip setter pentru partea reala
	{
		_re=r; //atribui partii reale valoarea data ca parametru in metoda Re
	}

	void Im(double i)//definesc metoda de tip setter pentru partea imaginara
	{
	//metodele Setter sunt de tip void pentru ca nu returneaza nici un tip de
data
		_im=i; ////atribui partii reale valoarea data ca parametru in metoda
Im

	}

	double Re() //definesc metoda de tip getter pentru partea reala ,
	{
```

```
		return _re;

	}

	double Im() //definesc metoda de tip getter pentru partea imaginara
	{

		return _im;

	}

	Complex()//definesc constructorul clasei Complex,
	{
		_re =0.0; //atribui valori default pentru cele doua atribute ale
clasei
		_im =0.0; //in caz ca prin getter nu atribui valori particulare
acestor atribute

		//la crearea obiectului c1 atributele se initializeaza cu valorile din
constructor
		//daca atribui valori atunci se initializeaza cu acele valori.
		//daca nu atribui valori nici prin setter nici prin constructori
atunci
		//valorile pentru atribute sunt preluate aleator din RAM
		}
};
int main()
{

	double real,imag;

	Complex c0;//creez obiectul cu numele c0 si tipul Complex

	cout<<"partea reala este:";
	cin>>real;

	cout<<"partea imaginara este:";

	cin>>imag;
	c0.Re(real);
	c0.Im(imag);
	cout <<"Numarul complex introdus de la tastatura este: "<<c0.Re()<< " + " <<
c0.Im()<<"i\n";


	Complex c1; //creez obiectul cu numele c1 si tipul Complex
	c1.Re(2.0); //atribui elemente particulare prin setter, daca sunt
	//comentate atunci se folosesc
	//valorile default din constructor
	c1.Im(3.5);

	cout <<"Numarul complex obtinut prin metodele de tip setter este:
"<<c1.Re()<< " + " << c1.Im()<<"i\n";

	Complex c2;//creez obiectul c2 tot de tipul Complex in care atributele isi
vor lua valorile
	//default din constructorul implicit
```

```
	cout <<"Numarul complex cu valori implicite din constructor este: 
"<<c2.Re()<< " + " << c2.Im()<<"i\n";
	return 0;

}//functia main NU se inchide cu ;
```

Al doilea exemplu de clasa va fi clasa Persoana , care va avea ca si atribute: Nume, Prenume , Varsta, Greutate. Vom scrie setter si getter pentru fiecare din aceste 4 atribute, apoi constructor implicit pentru a initializa atributele cu valori default.

Ca metode standard, vom implementa :

- Metoda TesteazaVarsta() care va lua un numar de la tastatura il va testa daca este intre anumite limite si doar daca este o varsta valida va fi atribuita persoanei.

- Metoda mananca() unde testam daca persoana nu cumva este deja prea grasa , caz in care nu ii mai dam de mancare ci doar vom afisa un mesaj de eroare. Daca persoana este normal atunci I se poate aplica metoda mananca in sensul ca greutatea va creste cu 10kg.

- Metoda dieta() unde testam ca nu cumva persoana este deja prea slaba , si implicit evitam cazul in care are o greutate negative (fizic imposibil) daca va avea mai putin de o anumita limita atunci afisam mesaj de eroare , daca este peste limita ii putem aplica metoda aceasta care consta in scaderea greutatii cu 10kg

- Metoda Profil() care , aplicata succesiv dupa mai multe metode mananca ()/dieta() va preciza in ce stare a greutatii se afla persoana, slaba, normal, grasa etc.

Sa examinam codul acestei clase:

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

class Persoana
{
	string _nume,_prenume; //atributele clasei Persoana, private
	int _varsta,_greutate;

public:
	void Nume(string n); //setter pentru nume

	void Prenume(string p); //setter pentru prenume
	string Prenume(); //getter pentru prenume
	string Nume(); //getter pentru nume

	void Varsta(int v); //setter pentru prenume
	int Varsta(); //getter pentru varsta

	int TesteazaVarsta(); //metode standard publice
	int Mananca();
	int Dieta();

	int Profil();

	Persoana();//declar constructorul clasei Persoana
};

void Persoana::Nume(string n) //setter pentru nume
{
	_nume=n;
}

void Persoana::Prenume(string p) //setter pentru prenume
{

	_prenume=p;
}

string Persoana::Nume() //getter pentru nume
{
	return _nume;
}

string Persoana::Prenume() //getter pentru prenume
{
	return _prenume;
}

int Persoana::Varsta() //getter pentru varsta
{
	return _varsta;
}

void Persoana::Varsta(int v)
{
	_varsta=v;
```

```
}

int Persoana::TesteazaVarsta()
{//atribui varsta unui obiect dar in anumite limite
	int v;

	do
	{
		cout<<"Introduceti varsta persoanei : ";
		cin>>v;
	}
	while(v<0 || v>200); //testez daca varsta este valida

	_varsta=v;

	return _varsta;
}

int Persoana::Mananca()//metoda standard
{

	if(_greutate<=80)//testez daca persoana este normala ca greutate
	{
		_greutate=_greutate+10;
	}
	else
		{
			cout<<"Persoana nu mai poate manca a devenit obeza, trebuie sa
slabeasca\n";
		}

	return _greutate;
}



int Persoana::Dieta()
{

	if(_greutate<=30)//testez daca persoana nu cumva e prea slaba
	{
		cout<<"Persoana nu poate tine dieta , este prea slaba, trebuie sa
manance\n";
	}
	else{
			_greutate=_greutate-10;
		}

	return _greutate;

}




int Persoana::Profil()

{
```

```
	switch(_greutate)
	{
		case 30:
			cout<<"Persoana este foarte slaba\n";
			break;

		case 40:
			cout<<"Persoana este slaba\n";
			break;

		case 50:
			cout<<"Persoana este slabuta\n";
			break;

		case 60:
			cout<<"Persoana este Normala\n";
			break;

		case 70:
			cout<<"Persoana este Plinuta\n";
			break;

		case 80:
			cout<<"Persoana este Grasa\n";
			break;

		default:
			cout<<"Eroare\n";
			break;

	}

}




Persoana::Persoana() // definesc constructorul clasei Persoana;

{

	_nume="Popescu";

	_prenume="Ion";

	_varsta=10;

	_greutate=50;

}

int main()

{
	Persoana standard;//valorile vor fi luate din constructorul implicit cout
<<"Numele complet al studentului implicit este :"<<standard.Prenume()<<"
"<<standard.Nume()<<"\n";
```

```
	Persoana student;//valorile vor fi luate din setteri
student.Nume("Dobrescu");//obiectul student apeleaza metoda de tip Setter iar
atributul nume va lua valoarea "Dobrescu"
	student.Prenume("Vlad");
	cout <<"Studentul "<<student.Prenume()<<" "<<student.Nume()<<" are varsta
"<<student.Varsta()<<" ani\n";
	student.Varsta(40);//valoare data prin setter
	cout<<"varsta studentului este "<<student.Varsta()<<" ani\n";
	//atribui varsta unui obiect dar in anumite limite
	cout<<"varsta studentului este "<<student.TesteazaVarsta()<<" ani \n";

	Persoana p4; //creez obiectul p4 de tip Persoana p4.Profil(); //
	cout<<"greutatea persoanei este "<<p4.Mananca()<<"\n";
	p4.Profil();
	cout<<"greutatea persoanei este "<<p4.Mananca()<<"\n";
	p4.Profil();//apelez succesiv metoda profil intre diversele apeluri ale
metodelor mananca() si dieta()
	cout<<"greutatea persoanei este "<<p4.Mananca()<<"\n";
	p4.Profil();
	cout<<"greutatea persoanei este "<<p4.Mananca()<<"\n"; cout<<"greutatea
persoanei este "<<p4.Mananca()<<"\n"; cout<<"greutatea persoanei este
"<<p4.Dieta()<<"\n"; cout<<"greutatea persoanei este "<<p4.Dieta()<<"\n";
	p4.Profil();
	cout<<"greutatea persoanei este "<<p4.Dieta()<<"\n"; cout<<"greutatea
persoanei este "<<p4.Dieta()<<"\n"; p4.Profil();
	cout<<"greutatea persoanei este "<<p4.Dieta()<<"\n"; cout<<"greutatea
persoanei este "<<p4.Dieta()<<"\n"; cout<<"greutatea persoanei este
"<<p4.Dieta()<<"\n";
	p4.Profil();

}
```

Tema:

Modelati **clasa Masina** si **clasa Vietate**: construiti setteri, getteri, constructor implicit pentru fiecare si metode standard.

Fiecare clasa se va realiza in fisier .cpp separat.