

PROGRAMARE OBIECT-ORIENTATA

LABORATOR 3 – CLASE SI OBIECTE (CONTINUARE), OPERATOR DE REZOLUTIE, POLIMORFISM, CONSTRUCTOR CU PARAMETRI, DESTRUCTOR.

In laboratorul anterior am vazut ca putem define complet o metoda (o functie membra a unei clase) direct in interiorul clasei. In aceasta lucrare vom introduce operatorul de rezolutie prin care vom putea define complet o metoda in afara clasei (lasand doar declaratia sa, prototipul in interiorul clasei).

De asemenea, in laboratorul precedent am vazut ca pentru a initializa cu valori atributele unei clase putem folosi

- pe de o parte constructorul implicit
- iar pe de alta parte metodele speciale de tip “Setter”.
- In aceasta lucrare vom explica un alt tip de constructor prin care putem aloca valori atributelor si anume “constructorul cu parametri”.

Intr-o alta ordine de idei, indiferent prin ce modalitate alegem sa alocam valori atributelor, modalitatea prin care returnam / ne folosim de valorile atributelor este prin “Getter”, deoarece aceasta metoda standard este singura care este publica. Va reamintim ca atributele unei clase sunt marcate implicit ca private (chiar in absenta modificadorului de acces private:, dupa acolada de deschidere a clasei elementele ce urmeaza sunt private pana la urmatorul modificador de access).

Exemplul 1: Modelam un nou tip de date denumit “Student”.

Pentru simplitatea clasei, acest nou tip de date poate avea urmatoarele atribute:

- ✓ Numele studentului – un sir de caractere, stocat intr-o variabila de tip string
- ✓ Varsta studentului – un numar natural, stocat intr-o variabila de tip int

Din punct de vedere al metodelor vom crea:

- ✓ Cate un setter si un getter pentru numele studentului si varsta
- ✓ Un constructor implicit in care initializam toate atributele clasei simultan (numele studentului si varsta). Reamintim, acestia poarta denumirea clasei (cu aceiasi capitalizare la toate caracterele ca si numele clasei).
 - La crearea obiectelor, compilatorul analizeaza argumentele; daca obiectul se creaza fara argumente atunci datele sunt luate din constructorul implicit. Orice obiect creat fara argumente, va apela constructorul implicit pentru initializarea argumentelor
 - Nu are tip de date de return
- ✓ Mai multi constructori cu parametri. Si acestia poarta denumirea clasei.
 - Daca obiectul are parametrii precizati la creare (sub forma de constante) atunci acestia trebuie sa corespunda ca numar, ca tip si ca ordine cu cei din constructorul cu argumente deja creat; altfel vom primi eroare.
 - Nu are tip de date de return
- ✓ Un destructor, care de obicei elibereaza resursele alocate obiectului, inclusiv zone de memorie alocate special. Aceasta metoda speciala in clasa poarta denumirea clasei precedata de caracterul TILDA “~”.
 - Aceasta este apelat la iesirea din functia main (), deoarece obiectele create in acest exemplu sunt variabile locale in functia main(), si ele se pierd/distruge la iesirea din functie.

Cateva concepte folosite.

Declaratie vs. definitie a functiei. Prototip

Reamintim acest concept de la materiile anterioare de programare.

Declaratia unei functii (denumita si prototipul functiei) include tipul de date returnate (exceptie notabila constructorii si destructorii care nu au tip de date returnate), numele functiei si trebuie sa contin tipurile de date ale argumentelor sale. Este succedat de operatorul “,”. Numele argumentelor este optional in declaratie.

Definitia unei functii include tipul de date returnate (exceptie notabila constructorii si destructorii care nu au tip de date returnate), numele functiei, tipurile si numele argumentelor.

Exemple de declaratii	Exemple de definitii
int main(); Student();	int main () {cout <<"Hello"<<endl; return 0;} Student() { cout << <<"in constructor implicit"<<endl; Nume="Ion"; Varsta=20; }
Student (int) ;	Student (int v) { cout << <<"in constructor implicit"<<endl; Nume="Ion"; Varsta= v ; }

Constructorul cu parametri

Constructori Pentru o clasa cu 1 atribut putem avea constructori cu 0 parametri (sau argumente, care este constructorul implicit), in care se va initializa atributul la o valoare potrivita, sau cu 1 parametru, in care acelasi atribut se va initializa cu valoarea pasata ca argument in parametrul constructorului.

Pentru o clasa cu doua atribute putem avea

- constructor cu 0 parametri (constructorul implicit)

Definirea obiectului din functia main ()	Constructorul (implicit) apelat
Student s1;	Student(); //constructor implicit

- constructori cu 1 parametru (cu tipul fiecaruia din cele 2 atribute)
- constructori cu 2 parametri (cu tipurile celor 2 argumente, in toate ordinile posibile)

Definirea obiectului din functia main ()	Prototipul constructorului cu parametri apelat
Student s1(5); //5 e intreg	Student(int); //constructor cu // argument int
Student s2("Ion"); //Ion e cuvant	Student (string); //constructor cu // argument string
Student s3 (5, "Ion"); //5 e primul, intreg // Ion este al doilea, cuvant	Student (int, string); //constructor cu //2 argumente // mai intai int // apoi string
Student s3 ("Ion", 5); // Ion este primul, cuvant // 5 e al doilea, intreg	Student (string, int); //constructor cu //2 argumente // mai intai string // apoi int

In toate aceste situatii constructorul trebuie sa initializeze **toate atributele**, si pe cele pentru care a primit in argumentul metodei o valoare implicita, dar si pe celelalte, pentru care nu a primit. In acest caz, programatorul este responsabil de initializarea atributelor ramase in codul metodei.

Deci modalitatatile (cunoscute) prin care putem seta valoarea unui atribut al unui obiect contin: constructor implicit, constructor cu parametru, seter.

Polimorfism

Conceptul pe care se bazeaza compilatorul in alegerea tipului de constructor in functie de existenta, numarul de parametri si ordinea lor se numeste POLIMORFISM sau SUPRAINCARCAREA FUNCTIILOR. Prin acest concept o functie este recunoscuta ca distincta nu doar pe baza numelui sau, ci a combinatiei de nume, precum si a tipului si ordinii argumentelor.

<code>void SetNume(string N);</code>	Cele 2 functii difera prin numele functiei
<code>void GetNume(string N);</code>	
<code>void SetNume(void);</code>	Cele 2 functii difera prin numarul si tipul argumentelor functiei
<code>void SetNume(string N);</code>	
<code>Student (int V);</code>	Cele 2 functii difera prin numarul si tipul argumentelor functiei
<code>Student (string N);</code>	
<code>Student (int V, string N);</code>	Cele 2 functii difera prin ordinea argumentelor functiei
<code>Student (string N, int V);</code>	

Conceptul de polimorfism este inrudit cu cel din programare de SUPRAINCARCARE a operatorilor.

Operatorul de rezolutie

Metodele (functii membre ale clasei) sunt declarate in clasa ca si prototip dar sunt definite in afara clasei avand in fata numelui clasei din care aparțin folosind operatorul `::` de rezolutie pentru a descrie apartenența la o anume clasa.

Pentru a specifica in exteriorul clasei apartenența unei metode intai se specifica tipul de date de return, apoi denumirea clasei, operatorul de rezolutie `(::)`, numele metodei, urmat de lista de tipuri si nume de atribute ale metodei.

```
<Tip de date return> <Denumirea clasei> :: Numele metodei (Tip_argument1 Nume_argument1, Tip_argument2  
Nume_argument2, ...)
```

Intre declaratia metodei din clasa si definitia functiei din exteriorul clasei exista corespondent intre elementele functiei. Sa exemplificam pe cazul metodei SetNume din clasa student ce urmeaza sa o folosim.

Specificarea numelui (ex: N) argumentului metodei (ex: SetNume) este optională . Este corect ca in prototip sa nu spun numele argumentului (N).	Specificarea numelui (ex: N) argumentului metodei (ex: SetNume) este optională . Este corect ca in prototip sa spun numele argumentului (N).
<pre>class Student { ... void SetNume(string); //prototip aka declaratie ... }; void Student::SetNume(string N) {Nume = N;} //definitie</pre>	<pre>class Student { ... void SetNume(string N); //prototip aka declaratie ... }; void Student::SetNume(string N) {Nume = N;} //definitie</pre>
<ul style="list-style-type: none"> • Returneaza void • Clasei Student • Ii apartine <code>(::)</code> • Se numeste SetNume • Are un prim argument de tipul string • Argumentul se cheama N. Denumirea argumentului este obligatorie doar in definitie. Variabila N este folosita in interiorul metodei. • Nume este denumirea atributului specific clasei Student. 	<ul style="list-style-type: none"> • Returneaza void • Clasei Student • Ii apartine <code>(::)</code> • Se numeste SetNume • Are un prim argument de tipul string • Argumentul se cheama N. Denumirea argumentului este obligatorie doar in definitie. Variabila N este folosita in interiorul metodei. • Nume este denumirea atributului specific clasei Student.

Sa analizam implementarea exemplului 1:

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

class Student //definesc un nou tip de date, o clasa Student
{
    //fara existenta unui alt modificator de access,
    //elementele din clasa sunt private
    string Nume;
    int Varsta;

public:
    void SetNume (string N); //prototip setter pentru Nume
    string GetNume (); //prototip getter pentru Nume

    void SetVarsta (int V); //prototip setter pentru Varsta
    int GetVarsta (); //prototip getter pentru Varsta

    Student(); //constructor implicit
    Student(int); //constructor cu 1 argument, care va impune varsta
    Student(string); //constructor cu 1 argument, care va impune numele
    Student(int, string); //constructor cu 2 argumente,
                           // care va impune varsta si numele
    Student(string, int); //constructor cu 2 argumente,
                           //care va impune varsta si numele

    ~Student(); //destructor

}; // la finalul clasei pun ; dupa {

void Student::SetNume (string N) //clasei "Student" ii apartine :: metoda SetNume
{
    cout << "in setter de Nume" << endl;
    //setez atributul Nume la valoarea impusa prin argumentul functiei: N
    Nume = N;
}

string Student::GetNume () //clasei "Student" ii apartine :: metoda GetNume
{
    cout << "in getter de Nume" << endl;
    return Nume;
}

void Student::SetVarsta (int V) //clasei "Student" ii apartine :: metoda SetVarsta
{
    cout << "in setter de Varsta" << endl;
    //setez atributul Varsta la valoarea impusa prin argumentul functiei: V
    Varsta = V;
}

int Student::GetVarsta ()

Student::Student() //clasei "Student" ii apartine :: constructorul implicit
{
    cout<<"in constructor implicit"<<endl;
    Nume = "Ion";
    Varsta = 20;
}

Student::Student(int V) //clasei "Student" ii apartine :: constructorul cu param. int
```

```

{
    cout<<"in constructor cu param. int"<<endl;
    Nume = "Ion";
    Varsta = V;
}

Student::Student(string N)//clasei "Student" ii apartine :: constr. cu param. string
{
    cout<<"in constructor cu param. string"<<endl;
    Nume = N;
    Varsta = 20;
}

Student::Student(int V, string N) //clasei "Student" ii apartine :: constr. cu 2 param.
{
    cout<<"in constructor cu param. int, string"<<endl;
    Nume = N;
    Varsta = V;
}

Student::Student(string N, int V) //clasei "Student" ii apartine :: constr. cu 2 param.
{
    cout<<"in constructor cu param. string, int"<<endl;
    Nume = N;
    Varsta = V;
}

Student::~Student() //clasei "Student" ii apartine :: destructorul ("~Student")
{
    cout<<"in destructor "<<endl;
}

int main()
{
    cout << "creere S1" << endl;
    Student s1; //va apela constructorul implicit

    cout << "creere S2" << endl;
    Student s2(5); //va apela constructorul cu 1 parametru
                    // de tipul constantei 5: int

    cout << "creere S3" << endl;
    Student s3("Ionel"); //va apela constructorul cu 1 parametru
                        // de tipul constantei "Ionel": string

    cout << "creere S4" << endl;
    Student s4(5,"Ionica"); //va apela constructorul cu 2 parametri
                           // primul de tipul constantei 5: int
                           // al doilea de tipul constantei "Ionica": string

    cout << "creere S5" << endl;
    Student s5("Ionica", 5); //va apela constructorul cu 2 parametri
                           // primul de tipul constantei "Ionica": string
                           // al doilea de tipul constantei 5: int

    cout << "la final " << endl;
    return 0;
    //la final se apeleaza destructor
    //    pentru fiecare din obiectele locale
    //    in functia main()S1, s2, s3, s4, s5
}

```