Laborator IV

CLASE, CONSTRUCTORI, DESTRUCTORI

1. Scopul lucrării

Lucrarea de laborator are ca scop ilustrarea conceptului de clasă (class) ca tip de date abstract, precum și a unor elemente legate intrinsec de construirea tipurilor abstracte în POO:

- constructori,
- destructor,
- stil de programare la nivelul acestor elemente.

2. Clase

Conceptul de clasă apare ca generalizare a structurilor: *struct* în C și C++ (*record* în Pascal). Structurile permit încapsularea datelor de tipuri diferite, rezultatul fiind un obiect care poate descrie o realitate cu mai multe atribute. Un exemplu clasic este:

```
struct Person
{
  char Name[80J; //Numele ...
  float Age;     //Varsta ...
  int Height;    //Inaltimea in cm ...
  float Weight;  //Greutatea ...
};
```

În POO structura rămâne un caz cu totul particular de clasă: clasă cu membrii publici. Clasa posedă un număr de elemente care permit atât o încapsulare a datelor, cât și "ascunderea" controlată a acestora.

Data hiding

Ascunderea datelor se face în cazul claselor prin declararea membrilor clasei ca aparținând unuia din cele trei nivele: *private*, *protected*, *public*.

private

Membrii declarați *private* pot fi accesați direct numai de membrii clasei. Acest tip de ascundere a datelor este gândit pentru "închiderea" obiectelor *private* ale clasei și accesul numai printr-o interfață proiectată folosind membrii *public*.

Observație: La derivarea claselor, membrii *private* nu pot fi accesați decât prin intermediul membrilor public ai clasei de bază.

protected

Membrii declarați *protected* pot fi accesați direct numai de membrii obiectului. La același nivel de derivare sunt similari membrilor *private*.

Observație: În clasele derivate membrii *protected* rămân accesibili direct, spre deosebire de cei *private*, și membrilor clasei derivate.

public

Membrii declarați *public* pot fi accesați direct prin intermediul obiectelor declarate ca fiind instanțe ale clasei. Membrii *public* se constituie în interfața clasei.

friend

Obiectele declarate "prietene", *friend*, pot accesa direct membrii unei clase. Acest mecanism ocolește mecanismul implementat de *data hiding*. Din această cauză nu trebuie să se abuzeze de folosirea obiectelor *friend*.

3. Constructori și destructori

Orice clasă folosește cel puțin două tipuri de funcții membru speciale: *constructori* și *destructor*.

Constructorii și destructorul sunt folosiți implicit de compilator pentru *crearea* și/sau *inițializarea* obiectelor declarate de tipul clasă, și respectiv pentru *distrugerea* obiectelor respective. Constructorii sunt apelați, pentru fiecare obiect de tipul clasă, la începutul *domeniului de existentă*, iar destructorul la sfârșitul acestuia. Pentru o clasă pot fi definiți mai mulți constructori, însă numai un destructor. Există contexte pentru care apelul constructorului se face explicit.

Constructori

Constructorii sunt funcții membru speciale care nu returnează nimic, nici măcar *void*. În schimb pot avea parametri. O clasă poate avea mai mulți constructori prin supraîncărcarea funcțiilor.

Constructor implicit

Constructorul implicit nu are parametri. Dacă nu este specificat de programator el va fi generat automat de compilator folosind un mecanism implicit.

Este bine ca fiecare clasă, mai ales cele ale căror obiecte ocupă memorie în *heap*, să fie prevăzute cu constructor implicit proiectat de programator (vezi cursurile și codul).

Constructor de copiere

Constructorul de copiere este folosit în momentul în care un obiect este declarat ca o copie a altui obiect. (În categoria constructorilor de copiere se pot înscrie și constructorii de conversie de tip. Prin specificul particular al acestora ei se deosebesc totuși de constructorii de copiere clasici.)

Constructorul de copiere al unei clase este apelat în cazul în care o funcție returnează un obiect de tipul clasei respective.

Dacă nu este specificat de programator, constructorul de copiere va fi generat automat de compilator folosind un mecanism implicit.

Este bine ca fiecare clasă, mai ales cele ale căror obiecte ocupă memorie în *heap*, sa fie prevăzute cu constructor de copiere proiectat de programator (vezi cursurile și codul).

Constructori cu parametri

Pe lângă constructorii implicit și cel de copiere o clasă poate conține o serie de constructori cu parametri, care sunt apelați conform cu regulile de supraîncărcare a funcțiilor.

Destructor

Destructorul unei clase este apelat în momentul în care domeniul de existentă al obiectului de tipul clasă este depășit. Destructorul este responsabil de *eliberarea resurselor* ocupate de obiectul care urmează să fie distrus (e.g. memorie alocată dinamic).

Ca și constructorii implicit și de copiere, este bine ca destructorul să fie bine proiectat pentru fiecare clasă.

4. Program 1

Acest cod ilustrează modul de operare cu membri *private*, *protected* și *public*, modul în care sunt apelați constructorii și destructorul clasei și modul de alocare dinamică de memorie pentru un vector de obiecte declarate de tipul clasă.

Descrieți pe o foaie de hârtie, acasă, după ce ați studiat cursul, modul în care sunt executate instrucțiunile acestui program.

Rulați programul pas cu pas și urmăriți funcționarea acestuia. Experimentați accesul eronat la membrii *private* și *protected*. Observați comportarea funcțiilor și operatorilor pentru alocarea dinamică de memorie.

Reorganizați programul în fișiere *header* și *surse* în modul în care considerați că este mai util pentru dezvoltarea unui program. (Se observă o oarecare dificultate în operarea un singur fișier.)

```
/*********************************************
clase1.cpp
*********************************************/

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

```
//Compilare conditionata ...
//#define _APEL_METODA_

using namespace std; //se asigura vizibilitatea spatiului de nume std

//Declararea tipului de date Clasa1
class Clasa1
{
//Declararea membrilor clasei Clasa1
private:
        int             IntPrivate;
        double          DoublePrivate;
        const char     *Nume; //Sir constant de caractere care reprezinta numele obiectului...
                              //Sau pointer la char constant ...
        //Declararea functiilor membru private ale clasei Clasa1
        void Mesaj(char *);

protected:
        int          IntProtected;
        double       DoubleProtected;
        //Declararea functiilor membru protected ale clasei Clasa1
        void PrintNumeCon(void); //Afiseaza numele la consola ...

public:
        int          IntPublic;
        double       DoublePublic;

        //Declararea functiilor membru publice ale clasei Clasa1 : metodele clasei
        Clasa1(void); //Constructor implicit ...
        Clasa1(Clasa1 &); //Constructor de copiere ...
        Clasa1(int, double, const char *, int, double, int, double); //Constructor ...
        Clasa1(const char *); //Constructor ... .
        //Modificatorul const asigura faptul ca valoarea de la adresa transferata nu va fi
//modificata.
        //Testul se face la compilare.

        ~Clasa1(); //Destructor ...

        //Atribuie membrilor valorile precizate ca parametri ...
        void Set(int, double, const char *, int, double, int, double);
        void SetNume(const char *);
        void SetPrivate(int, double);
        void SetProtected(int, double);
        void SetPublic(int, double);

       //Functii membru "inline"
      //Returneaza referinta -> un pseudonim pentru obiectul returnat -> este lvalue si rvalue
       int & GetPrivateInt(void) {return IntPrivate;}
       double & GetPrivateDouble(void) {return DoublePrivate;}
       int & GetProtectedInt(void) {return IntProtected;}
       double & GetProtectedDouble(void) {return DoubleProtected;}

       void PrintCon(void); //Afiseaza valorile variabilelor membru la consola ...
       void ScanCon(void); //Citeste valorile variabilelor membru de la consola ...

};
```

```
//Definitia functiilor membru ale clasei Clasa1.
//Se foloseste operatorul de rezolutie pentru a specifica
//clasa careia ii apartine functia.
Clasa1::Clasa1(void)
{
    Nume = nullptr;
    IntPrivate = IntProtected = IntPublic = 0;
    DoublePrivate = DoubleProtected = DoublePublic = 0.0;
    Mesaj("S-a apelat constructorul implicit al clasei \"Clasa1\"" );
}
```

```
Clasa1::Clasa1(Clasa1 &RefObiectClasa1)
{
    Nume = RefObiectClasa1.Nume;
    IntPrivate = RefObiectClasa1.IntPrivate;
    DoublePrivate = RefObiectClasa1.DoublePrivate;
    IntProtected = RefObiectClasa1.IntProtected;
    DoubleProtected = RefObiectClasa1.DoubleProtected;
    IntPublic = RefObiectClasa1.IntPublic;
    DoublePublic = RefObiectClasa1.DoublePublic;
    Mesaj("S-a apelat constructorul de copiere al clasei \"Clasa1\"");
}

Clasa1::Clasa1(int IntPrivate, double DoublePrivate, const char *NumeObiect,
                        int IntProtected, double DoublePotected,
                        int IntPublic, double DoublePublic)
{
#ifdef _APEL_METODA_
    Nume = nullptr; //De ce se face aceasta atribuire in acest moment ?
    Set(IntPrivate, DoublePrivate, NumeObiect, IntProtected, DoublePotected, IntPublic, DoublePublic);
#else
    Clasa1::IntPrivate = IntPrivate; //Se utilizeaza operatorul de rezolutie in cadrul clasei...
    Clasa1::DoublePrivate = DoublePrivate;
    Nume = NumeObiect;
    Clasa1::IntProtected = IntProtected;
    Clasa1::DoubleProtected = DoublePotected;
    Clasa1::IntPublic = IntPublic;
    Clasa1::DoublePublic = DoublePublic;
#endif
    Mesaj("S-a apelat constructorul cu parametri al clasei \"Clasa1\"");
}

Clasa1::Clasa1(const char *NumeObiect)
{
    Nume = NumeObiect;
    IntPrivate = IntProtected = IntPublic = 0;
    DoublePrivate = DoubleProtected = DoublePublic = 0.0;
    Mesaj("S-a apelat constructorul cu parametri al clasei \"Clasa1\"");
}

Clasa1::~Clasa1()
{
    Mesaj("S-a apelat destructorul clasei \"Clasa1\"");
}

void Clasa1::Mesaj(char *Mesaj)
{
    PrintNumeCon();
    cout << " : " << Mesaj << endl;
    cout << "Apasati o tasta pentru a continua..."<<endl;
    _getch();
}

void Clasa1::Set(int IntPrivateParam, double DoublePrivateParam, const char * NumeParam,
                        int IntProtectedParam, double DoubleProtectedParam,
                        int IntPublicParam, double DoublePublicParam)
{
#ifdef _APEL_METODA_
    SetNume(NumeParam);
    SetPrivate(IntPrivateParam, DoublePrivateParam);
    SetProtected(IntProtectedParam, DoubleProtectedParam);
    SetPublic(IntPublicParam, DoublePublicParam);
#else
    IntPrivate = IntPrivateParam;
    DoublePrivate = DoublePrivateParam;
    if(Nume==nullptr) //Daca obiectul nu are nume ...
        Nume = NumeParam;
    IntProtected = IntProtectedParam;
    DoubleProtected = DoubleProtectedParam;
    IntPublic = IntPublicParam;
    DoublePublic = DoublePublicParam;
#endif
}
```

```
void Clasa1::SetNume(const char *NumeParam)
{
    if(!Nume)
        Nume = NumeParam;
}
void Clasa1::SetPrivate(int Int, double Double)
{
    IntPrivate = Int;
    DoublePrivate = Double;
}
void Clasa1::SetProtected(int Int, double Double)
{
    IntProtected = Int;
    DoubleProtected = Double;
}
void Clasa1::SetPublic(int Int, double Double)
{
    IntPublic = Int;
    DoublePublic = Double;
}
void Clasa1::PrintNumeCon(void)
{
    cout << "obiectul ";
    if(Nume != nullptr) //Daca Nume este diferit de nullptr
        cout << Nume;
    else //Daca Nume == nullptr
        cout << "Anonim ";
}
void Clasa1::PrintCon(void)
{
    cout << endl;
    PrintNumeCon();
    cout << " : \n" << "Valoarea \"int\" private:\t\t\t" << IntPrivate << endl;
    //Acces indirect la membru private ...
    cout << "Valoarea \"double\" private:\t\t" << DoublePrivate << endl;
    //Acces indirect la membru private ...
    cout << "Valoarea \"int\" protected:\t\t" << IntProtected << endl;
    //Acces indirect la membru protected ...
    cout << "Valoarea \"double\" protected:\t\t" << DoubleProtected << endl;
    //Acces indirect la membru protected ...
    cout << "Valoarea \"int\" public:\t\t\t" << IntPublic << endl;
    //Acces direct la membru public ...
    cout << "Valoarea \"double\" public:\t\t" << DoublePublic << endl;
    //Acces direct la membru public ...
    cout << endl;
}
void Clasa1::ScanCon(void)
{
    cout << endl << "Pentru ";
    PrintNumeCon();
    cout << endl << "Introduceti:" << endl;
    cout << "Valoarea \"int\" private:" << endl;
    cin >> IntPrivate; //Acces indirect la membru private ...
    cout << "Valoarea \"double\" private:" << endl;
    cin >> DoublePrivate; //Acces indirect la membru private ...
    cout << "Valoarea \"int\" protected:" << endl;
    cin >> IntProtected; //Acces indirect la membru protected ...
    cout << "Valoarea \"double\" protected:" << endl;
    cin >> DoubleProtected; //Acces indirect la membru protected ...
    cout << "Valoarea \"int\" public:" << endl;
    cin >> IntPublic; //Acces direct la membru public ...
    cout << "Valoarea \"double\" public:" << endl;
    cin >> DoublePublic; //Acces direct la membru public ...
    cout << endl;
}
```

```
//Obiect global de tipul Clasa1:
Clasa1 Obiect1; //Se apeleaza constructorul implicit


int main(void)
{
    //Citirea de la consola a valorilor pentru membrii unui obiect ... .
    cout << "Introduceti pentru Obiect1:\nValoarea \"int\" private" << endl;
    cin >> Obiect1.GetPrivateInt(); //Acces indirect la membru private ...

    cout << "Valoarea \"double\" private" << endl;
    cin >> Obiect1.GetPrivateDouble(); //Acces indirect la membru private ...

    cout << "Valoarea \"int\" protected" << endl;
    cin >> Obiect1.GetProtectedInt(); //Acces indirect la membru protected ...

    cout << "Valoarea \"double\" protected" << endl;
    cin >> Obiect1.GetProtectedDouble(); //Acces indirect la membru protected ...

    cout << "Valoarea \"int\" public" << endl;
    cin >> Obiect1.IntPublic; //Acces direct la membru public ...

    cout << "Valoarea \"double\" public" << endl;
    cin >> Obiect1.DoublePublic; //Acces direct la membru public ...

    //Scrierea la consola a valorilor membrilor unui obiect ... .
    Obiect1.PrintCon();

    Clasa1 Obiect2(Obiect1); //Se apeleaza constructorul de copiere ...

    cout << endl << "Valorile unei copii:" << endl;
    Obiect2.PrintCon();

    int                 Int1, Int2, Int3;
    double           Double1, Double2, Double3;

    cout << "Introduceti trei valori \"int\"" << endl;
    cin >> Int1 >> Int2 >> Int3;
    cout << "Introduceti trei valori \"double\"" << endl;
    cin >> Double1 >> Double2 >> Double3;

    Clasa1 Obiect3(Int1, Double1, "Obiect 3", Int2, Double2, Int3, Double3);
    Obiect3.PrintCon();

    //Instructiunea compusa : bloc
    {
        Clasa1 Obiect4("\"Obiect local unui bloc\"");
        Obiect4.ScanCon();
        Obiect4.PrintCon();
    }

    //Alocare dinamica de memorie pentru un obiect din clasa Clasa1
    Clasa1 *PtrClasa1;
    PtrClasa1 = new Clasa1("\"Obiect alocat dinamic\"");
    if(PtrClasa1)
    {
        PtrClasa1->ScanCon();
        PtrClasa1->PrintCon();
    }
    //Eliberarea memoriei
    if(PtrClasa1)
    {
        delete PtrClasa1;
        PtrClasa1 = nullptr; //Obicei bun ...
    }

    int Dim;
    cout << "Introduceti dimensiunea vectorului: ";
    cin >> Dim;
    //Alocare dinamica de memorie pentru un vector de obiecte din clasa Clasa1
    PtrClasa1 = new Clasa1[Dim]; //Se apeleaza constructorul implicit de Dim ori ...
```

```
    if(PtrClasa1)
    {
        Clasa1 *PtrClasa1Temp = PtrClasa1;
        for(int Contor = 0; Contor < Dim; Contor++)
        {
            cout << endl << "Obiectul " << Contor << " alocat dinamic:" << endl;
            (PtrClasa1Temp++)->PrintCon();
            cout << "Apasati o tasta pentru a continua ...";
            _getch();
        }
        cout << endl;
    }
    //Eliberarea memoriei
    if(PtrClasa1)
    {
        delete []PtrClasa1;
        //delete PtrClasa1; //nu este apelat destructorul decat pentru primul obiect ...
        PtrClasa1 = nullptr; //Obicei bun ...
    }

    //Alocare memoriei folosind malloc() ... .
    //Nu se apeleaza nici constructor nici destructor !!! .
    PtrClasa1 = (Clasa1 *)malloc(Dim * sizeof(class Clasa1));
    if(!PtrClasa1)
    {
        cout << "Eroare la alocarea dinamica de memorie ..." << endl;
        return 1;
    }
    for(int Contor = 0; Contor < Dim; Contor++)
        PtrClasa1[Contor].ScanCon();
    for(int Contor = 0; Contor < Dim; Contor++)
        PtrClasa1[Contor].PrintCon();

    if(PtrClasa1)
    {
        free(PtrClasa1);
    }

    //Alocare memoriei folosind calloc() ... .
    //Nu se apeleaza nici constructor nici destructor !!! .
    PtrClasa1 = (Clasa1 *)calloc(Dim, sizeof(class Clasa1));
    if(PtrClasa1)
    {
        free(PtrClasa1);
    }

    return 0;
}
```

5. Program 2

Acest cod ilustrează modul de operare cu membri *private*, *protected* și *public*, modul în care sunt apelați constructorii și destructorul clasei și modul de alocare dinamică de memorie în interiorul obiectelor declarate de tipul clasă (vector de *int*).

Descrieți pe o foaie de hârtie, acasă, după ce ați studiat cursul, modul în care sunt executate instrucțiunile acestui program.

Rulați programul pas cu pas și urmăriți funcționarea acestuia. Experimentați accesul eronat la membrii *private* și *protected*. Observați comportarea funcțiilor și operatorilor pentru alocarea dinamică de memorie. Încercați să suprimați constructorii implicit și cel de copiere și rulați programul în varianta în care compilatorul a generat implicit acești constructori.

Reorganizați programul în fișiere *header* și *surse* în modul în care considerați că este mai util pentru dezvoltarea unui program. (Se observă o oarecare dificultate în operarea unui singur fișier.)

```
/*********************************************
Clase2.cpp
*********************************************/

#include <iostream>
#include <conio.h>

using namespace std; //se asigura vizibilitatea spatiului de nume std

//Starea in care se poate afla un obiect ... .
enum State
{
    Distrus = 0,
    Construit = 1,
    SetUpCompleted = 3
    //Alte stari pe care le poate avea obiectul ... .
};

//Starea de eroare pe care o poate
enum Error
{
    success = 0,
    error = 1
    //Alte stari de eroare ... .
};

class Clasa2
{
private:
        int     StareObiect; //Stare obiectului ... .
        int    *PtrIntPrivate; //Vector de intregi ... .
        int     Dim; //Dimensiunea vectorului de intregi ... .

public:
        Clasa2(void); //Constructor implicit ... .
        Clasa2(Clasa2 const &); //Constructor explicit ... .
        Clasa2(int); //Constructor cu parametru dimensiunea vectorului de intregi ... .
        ~Clasa2(); //Destructor ... .

        void ClearAndDestroy(void); //Functie pentru distrugerea explicita a unui obiect ... .
        int SetUp(int); //Functie pentru reconstruirea explicita a unui obiect ... .
        int Copy(Clasa2 const &); //Copierea obiectului parametru ... .

        void ScanCon(void); //Citirea valorilor de la cosola ... .
        void PrintCon(void); //Afisarea valorilor la consola ... .
};

Clasa2::Clasa2(void)
{
    StareObiect = Construit;
    PtrIntPrivate = nullptr;
    Dim = 0;
}

Clasa2::Clasa2(Clasa2 const &C)
{
    //De ce asa ?
    StareObiect = Distrus;
    PtrIntPrivate = nullptr;
    Copy(C);
}

Clasa2::Clasa2(int Dim)
{
    StareObiect = Distrus;
    SetUp(Dim);
}

Clasa2::~Clasa2()
{
    ClearAndDestroy();
```

```
}

//Are grija sa nu distruga efectiv de doua ori consecutiv acelasi obiect.
//Aceasta previne printre altele eliberarea unei zone de memorie deja eliberata.
void Clasa2::ClearAndDestroy(void)
{
    if(!StareObiect) //Daca obiectul este deja distrus ....
        return;
    StareObiect = Distrus;
    if(!PtrIntPrivate)
        return;
    delete []PtrIntPrivate; //Este eliberata zona de memorie corespunzatoare vectorului ....
    PtrIntPrivate = nullptr; //Obicei bun ...
}

int Clasa2::Copy(Clasa2 const &C)
{
    if(StareObiect)
        ClearAndDestroy(); //Este distrus explicit obiectul curent ... .
    if(!C.StareObiect)
        return success;

    StareObiect = C.StareObiect;
    Dim = C.Dim;
    if(!Dim)
    {
        PtrIntPrivate = nullptr;
        return success;
    }
    PtrIntPrivate = new int[Dim]; //Alocare dinamica a memoriei necesare pentru Dim int ... .
    if(!PtrIntPrivate)
        return error;

    //Copierea valorilor din obiectul sursa ... .
    for(int Contor = 0; Contor < Dim; Contor++)
        PtrIntPrivate[Contor] = C.PtrIntPrivate[Contor];

    /*
    //Sau cu aces direct prin pointer ... .
        int *PtrIntDest = PtrIntPrivate;
        int *PtrIntSursa = C.PtrIntPrivate;
        for(int Contor = 0; Contor < Dim; Contor++)
            *PtrIntDest++ = *PtrIntSursa++;
    */

    StareObiect = SetUpCompleted;
    return success;
}

int Clasa2::SetUp(int Dim)
{
    if(StareObiect)
        ClearAndDestroy(); //Este distrus explicit obiectul curent ... .

   //Se reconstruieste obiectul ... .
   Clasa2::Dim = Dim;
   PtrIntPrivate = new int[Dim];
   if(!PtrIntPrivate) //Nu s-a putut aloca memorie pentru Dim int ... .
   {
       StareObiect = Distrus;
       return error;
   }
   //S-a reusit alocare memoriei pentru Dim int ... .
   StareObiect = SetUpCompleted;
   return success;
}

void Clasa2::ScanCon(void)
{
    if(!StareObiect || !PtrIntPrivate)
    {
```

```
        cout << "Care este dimensiunea vectorului de intregi ? ";
        cin >> Dim;
        if(SetUp(Dim))
        {
            cout << "Completati mesajul in mod corespunzator !!!" << endl;
            return;
        }
    }
    cout << "Introduceti valorile intregi ale elementelor vectorului de "
         << Dim << " intregi:" << endl << "Indice\t\tvaloare" << endl;
    for(int Contor = 0; Contor < Dim; Contor++)
    {
        cout << Contor << " :\t\t";
        cin >> PtrIntPrivate[Contor];
    }
    return;
}

void Clasa2::PrintCon(void)
{
    if(!StareObiect)
    {
        cout << "Obiect neconstruit ..." << endl;
        return;
    }
    if(!PtrIntPrivate)
    {
        cout << "Vector vid ..." << endl;
        return;
    }
    cout << "Elementele vectorului de "
         << Dim << " intregi:" << endl << "Indice\t\tvaloare" << endl;
    for(int Contor = 0; Contor < Dim; Contor++)
    {
        cout << Contor << " :\t\t" << PtrIntPrivate[Contor] << endl;
    }
    cout << "Apasati o tasta pentru a continua..." << endl;
}

void main(void)
{
    Clasa2 Obiect1;

    Obiect1.PrintCon();
    Obiect1.ScanCon();
    Obiect1.PrintCon();

    Clasa2 Obiect2(Obiect1);
    Obiect2.PrintCon();
    Obiect1.SetUp(10);

    Obiect1.PrintCon();
    Obiect1.ScanCon();
    Obiect1.ClearAndDestroy();
    Obiect1.PrintCon();
    Obiect1.Copy(Obiect2);
    Obiect1.PrintCon();
    Obiect2.PrintCon();
}
```

6. Temă

1. Definiți clasa *Punct* care să aibă ca membri *privați* coordonatele unui punct x și y, reali dublă precizie și ca membri *publici*: constructorul implicit, un constructor cu parametri, constructorul de copiere, destructorul, funcții membru de actualizare a coordonatelor x și respectiv y. Scrieți un program minimal de test pentru clasa astfel definită care să declare o variabila de tip *Punct p1* și o variabila de tip *Punct p2*, cu valori inițiale x=0.9, y=-1.2. Ilustrați utilizarea constructorului de copiere pentru construcția unei variabile *p3*, cu aceleași coordonate ca și *p2*. Afișați apoi valorile coordonatelor punctelor *p1*, *p2* și *p3*.

Indicație: folosiți ca model exemplul de la curs pentru clasa *Complex*.

2. Analizați programele prezentate în lucrarea de laborator folosind creionul și hârtia. În laborator rulați programele pas cu pas și confruntați rezultatele analizei pe hârtie cu rezultatele rulării programelor respective.

3. Reconcepeți exemplul ce ilustra o aplicație a structurilor de date, din lucrarea de laborator III, utilizând clase.

Conceptul de spaţiu de nume (namespace)

Un *spaţiu de nume* reprezintă un mecanism de grupare a elementelor ce se doresc a fi incluse într-un program sau într-o bibliotecă, pentru a se evita conflictul de nume. Astfel, folosind cuvântul cheie **namespace**, se defineşte o un grup declarativ ce oferă o modalitate de a separa un set de nume de altul. Prin urmare, numele din cadrul unis spaţiu nu va intra în conflict cu acelaşi nume declarat în alt spaţiu.

Declararea unui spaţiu de nume - exemple

```
namespace ConstanteLungi
{
const double PI = 3.141592653589793;
const double E = 2.718281828459045;
}

namespace ConstanteScurte
{
const double PI = 3.1415;
const double E = 2.7182;
}
```

Referirea la constantele PI şi E din cele două spaţii de nume se face cu ajutorul operatorului de rezoluţie :: .

```
cout << ConstanteLungi::PI <<endl; // afiseaza valoarea lui PI din spatiul ConstanteLungi
cout << ConstanteScurte::PI<<endl; // afiseaza valoarea lui PI din spatiul ConstanteScurte
cout <<  ConstanteLungi::E<<endl; //afiseaza valoarea lui E din spatiul ConstanteLungi
cout << ConstanteScurte::E<<endl; // afiseaza valoarea lui PI din spatiul ConstanteScurte
```

Directiva **using** – asigură vizibilitatea spaţiului de nume.

```
using namespace ConstanteLungi;

// In acest caz referirea la constantele PI şi E din cele două spaţii de nume se poate face astfel:

 cout << PI <<endl;  // afiseaza valoarea lui PI din spatiul ConstanteLungi
 cout<< ConstanteScurte::PI<<endl; // afiseaza valoarea lui PI din spatiul ConstanteScurte
 cout <<  E<<endl; //afiseaza valoarea lui E din spatiul ConstanteLungi
cout << ConstanteScurte::E<<endl; // afiseaza valoarea lui E din spatiul ConstanteScurte
```

Obs. Toate clasele, funcţiile, variabilele care fac parte din biblioteca standard C++ se găsesc în spaţiul de nume **std**. Ex: cin, cout, string etc.

De exemplu, pentru a face vizibil numele cout se foloseşte:
using std::cout;
Pentru a face vizibile toate componentele bibliotecii standard C++ se foloseşte:
using namespace std;