# Lucrarea de laborator nr. 2

**Tema:** Clase (constructori, destructori) **.** Funcţii şi clase prietene.

**Scopul lucrării:** familiarizarea studenţilor cu noţiunea de clase, utilizarea constructorilor, destructorilor, cu noțiunile de funcţii şi clase prietene.

**Consideraţiile teoretice necesare:**

***Clase***

Sintaxa simplificată a declarării unei clase este următoarea:

*class* NumeClasă
{
...
declaraţii variabile membri
...
declaraţii funcţii membri
...
}

Din definirea clasei se poate observa că clasa este asemănătoare cu o structură. Ea are în componenţa sa membri atît de tip variabilă, cît şi de tip funcţie. Pentru datele din interiorul clasei se utilizează, de regulă, termenul de *date membri*, iar pentru funcţii – denumirea de *funcţii* sau *metode*. O clasă permite *incapsularea* în interiorul sau a datelor şi a codului.

Pentru a putea utiliza efectiv un tip de date (în cazul de faţă o clasă), trebuie sa definim o variabilă de acel tip. Într-un mod similar declaraţiei

*int i;*

putem scrie:

*NumeClasă variabilă*

Vom considera că *variabilă* este un *obiect*. Exprimarea uzuală este că *un obiect este instanţierea unei clase*.

O clasă este compusă din două părţi: declaraţia şi implementarea ei. Declaraţia clasei prezintă membrii clasei. Membrii clasei sînt variabile de instanţiere şi funcţii membri indicate prin prototipul lor (tipul returnat, numele funcţiei, lista de parametri). Implementarea funcţiilor membri are loc prin implementarea clasei. Gradul de accesibilitate la elementele componente ale clasei este indicat prin cuvintele: *private* sau *protected* – elementele clasei sînt accesate numai prin intermediul funcţiilor membri sau prietene *friend, public* – toate elementele sînt disponibile în exteriorul clasei. De exemplu:

```
#include<iostream.h>
class myclass   // se declară un nou tip de date myclass
{private:  int a;      // componenta  int a  se declară implicit în //zona private
public:           // funcţiile membri declarate mai joc sînt din //zona public
void set_a (int num); // prin intermediul acestor funcţii se
                          // accesează componenta a
int get_a ();
};
void myclass::set_a(int num) {  a=num;}
              //  această funcţie atribuie  valoare componentei a
int myclass::get_a(){return a; }
```

```
        // această funcţie furnizează valoarea componentei a
void main ()          // funcţia de bază a programului
{  myclass ob1, ob2;   // se declară două obiecte ob1 şi ob2 //de tipul myclass
   ob1.set_a (10);    // pentru obiectul ob1 se atribuie
                      //valoare componentei a egală cu 10
   ob2.set_a (99);   // pentru obiectul ob2 se atribuie
                      //valoare componentei a egală cu 99
    cout << ob1.get_a ()<<" \n"; // pentru obiectul ob1 se //furnizează valoarea
                      componentei a
                      //care apoi se tipăreşte la ecran
  cout << ob2.get_a () << " \n"; // pentru obiectul ob2 se //furnizează valoarea
                       componentei a
                       //care apoi se tipăreşte la ecran
}
```

Rezultatul îndepliniri programului:

```
10
99
```

Funcţiile *set_a* şi *get_a*, pentru setarea şi furnizarea valorii pentru componenta *a*, nu sînt necesare,dacă componenta *a* va fi inclusă în zona *public*,. Componenta *a* va fi explicit accesată şi i se va iniţializa sau atribui valoare. Exemplul de mai sus se va modifica în felul următor:

```
#include<iostream.h>
class myclass   // se declară un nou tip de date myclass
{public:
   int a;        // componenta  int a  se declară explicit în
                  //zona public
        // funcţiile membri declarate mai sus nu sînt necesare
};
void main ()
{  myclass ob1, ob2;     // se declară două obiecte ob1 şi ob2 //de tipul myclass
    ob1.a =10;            // pentru obiectul ob1 se iniţializează //valoarea componentei a în
                          mod
                          //explicit cu valoarea 10
 ob2.a = 99;         // pentru obiectul ob2 se iniţializează  //valoarea componentei a în mod
                          // explicit cu valoarea 99
cout << ob1.a << "\n"; // pentru obiectul ob1
                          // se tipăreşte valoarea componentei a
 cout << ob2.a << "\n"; // pentru obiectul ob2
                          // se tipăreşte componentei a
}
```

Rezultatul îndepliniri programului:

```
10
99
```

***Constructorii*** sînt un tip special de funcţie membru, avînd acelaşi nume ca şi numele clasei, nu returnează rezultat şi sînt apelaţi automat la instanţierea unei clase, fie ea statică sau dinamică. Ei au scopul de a atribui valori iniţiale elementelor membri, dar pot efectua şi unele operaţii, cum ar fi, alocarea dinamică de memorie, deschiderea unui fişier ş.a. De exemplu:

```
class persoana
{  private:
  char nume[40];
  long int telefon;
public:
persoana() {nume='\0'; telefon =0;};
//constructorul iniţializează valori nule elementelor membri
persoana(char*p, long int t) {strcpy(nume,p); telefon=t;}
//constructor iniţializează valori concrete pentru elementele //membri  ale clasei
persoana(char* nume) {return nume; };
//aceste funcţii atribuie valori pentru elementele membri nume
persoana(long int telefon) {return telefon;};        //şi telefon
persoana persoana_ input (char *n, long int t=0)
        {persoana p;
         strcpy(p.nume,n); p.telefon=t;
         return p;  };
};
```

Apelul constructorului se efectuează în momentul declarării unui obiect. Dacă declarăm o variabilă de tipul *persoana*, fie

```
persoana p = persoana ("Vasilina", 743567);
```

sau

```
persoana p ("Vasilina", 743567);
```

constructorul va iniţializa elementele membri *nume* şi *telefon* ale clase *persoana* respectiv cu valorile *"Vasilina"* şi *743567*. Dacă se va declara un obiect de tipul *persoana* fără date iniţiale, constructorul va completa elementele membri *nume* cu stringul vid '\0' şi *telefon* cu valoarea *0*.

***Destructorii*** dezactivează toate funcţiile unui obiect, îl distruge şi sînt apelaţi automat la eliminarea unui obiect, la încheierea timpului de viaţă în cazul static, sau la apelul unui *delete* în cazul dinamic. De regulă, destructorii sînt utilizaţi în cazul, cînd constructorii efectuează alocări dinamice de memorie. Destructorul are acelaşi nume ca şi constructorul, fiind precedat de semnul "~". De exemplu:

```
#include<iostream.h>
#include<string.h>
#include<stdlib.h>
#define Size 255
class strtype
{ private:
char *p;
int len;
public:
strtype()                          // constructorul
{ p=new char;
 if (!p){cout << "Eroare la alocarea memoriei \n";  exit(1);}
*p='\0'; len=0;    };
~strtype()  {cout << "Eliberarea memoriei\n";   delete p; }
                          // destructorul
void set (char*ptr)
```

```
{ if (strlen(ptr)> Size )
  cout<<"Stringul conţine mai mult de 255 de caractere \n"; strcpy(p,ptr);
len=strlen(p);};
void show()
{ cout << p << "- lungimea "<< len << "\n";}
};
void main()
{ strtype s1,s2;
s1.set ("Test");    s2.set("Program C++");
s1.show();  s2.show();
}
```

Rezultatul îndeplinirii programului:

*Test- lungimea 4*
*Program C++- lungimea 11*
*Eliberarea memoriei*
*Eliberarea memoriei*

Destructorii obiectelor membri sînt apelaţi, după ce destructorul obiectului principal a fost executat. Dacă obiectul membru este compus din alte obiecte, atunci se va proceda la executarea destructorilor obiectelor incluse. Destructorii obiectelor membri sînt apelaţi în ordine inversă, în care aceştea apar în declaraţia clasei.

Din punct de vedere cronologic, constructorul este apelat după alocarea memoriei necesare, deci în faza finală a creării obiectului, iar destructorul înaintea eliberării memoriei aferente, deci în faza iniţială a distrugerii sale.

Constructorii şi destructorii se declară şi se definesc similar cu celelalte funcţii membri, dar prezintă o serie de caracteristici specifice:

- numele lor coincide cu numele clasei căreia ii aparţin; destructorii se disting de constructori prin faptul că numele lor este precedat de caracterul
- nu pot returna nici un rezultat
- nu se pot utiliza pointeri către constructori sau destructori
- constructorii pot avea parametri, destructorii insa nu. Un constructor fără parametri poartă denumirea de *constructor implicit*.

în care o clasa nu dispune de constructori sau destructori, compilatorul de C++ generează automat un constructor, respectiv destructor, implicit.

***Membrii unei clase***

Accesarea membrilor unei clase se face în felul următor:

*obiect.VariabiăMembru = valoare;*

pentru *accesul* la o variabilă membru, şi

*obiect.FuncţieMembru();*

pentru *apelarea* unei funcţii membri.

Pentru exemplificare să consideram o implementare a noţiunii de *punct*. Ca variabile membri avem nevoie doar de coordonatele *x* şi *y* care definesc poziţia în spaţiu a unui punct. Am mai declarat o funcţie care calculează aria dreptunghiului avînd punctele *(0, 0)* şi *(x, y)*.

```
class Point
{unsigned x, y;
unsigned long Arie() {return x * y;};
unsigned GetX();
```

```
unsigned GetY();
void SetX(unsigned X);
void SetY(unsigned Y);
};
unsigned Point::GetX() {return x;}
unsigned Point::GetY(){return y;   }
void Point::SetX(unsigned X){  x = X;  }
void Point::SetY(unsigned Y) {  y = Y;  }
```

Am folosit un operator nou, specific C++, *::*, numit *operator de rezoluţie*, numit şi *operator de acces* sau de *domeniu*. El permite accesul la un identificator dintr-un bloc în care acesta nu este vizibil datorită unei alte declaraţii locale. Un exemplu de folosire este următorul:

```
char *sir = "variabilă globală";
void funcţie()
{ char *sir = "variabilă locală";
printf("%s\n", ::sir); // afişează variabila globală
printf("%s\n", sir); // afişează variabila locală
}
```

Pentru definiţiile funcţiilor membri aflate în afara declaraţiei clasei este necesară specificarea numelui clasei urmat de acest operator, indicînd faptul că funcţia are acelaşi domeniu cu declaraţia clasei respective şi este membru a ei, deşi este definită în afara declaraţiei.

***Cuvîntul-cheie this.*** Toate funcţiile membri ale unei clase primesc un parametru ascuns, pointer-ul *this*, care reprezintă adresa obiectului în cauză. Acesta poate fi utilizat în cadrul funcţiilor membri. De exemplu:

```
unsigned long Point::Arie()
{return this->x * this->y;  }
```

***Crearea şi distrugerea obiectelor.*** Să considerăm următorul program C++:

```
void main()
{Point p; }
```

În momentul definirii variabilei *p*, va fi alocat automat spaţiul de memorie necesar, acesta fiind eliberat la terminarea programului. În exemplul de mai sus, variabila *p* este de tip static. În continuare vom modifica acest program pentru a folosi o variabilă dinamică (pointer).

```
void main()
{ Point *p;
p = new Point;
p->x = 5;    p->y = 10;
printf("Aria = %d\n", p->Aria());
delete p;  }
```

Operatorul *new* este folosit pentru alocarea memoriei, iar sintaxa acestuia este:

```
variabila = new tip;
variabila = new tip(valoare_iniţială);
variabila = new tip[n];
```

Prima variantă alocă spaţiu pentru *variabilă* dar nu o iniţializează, a doua variantă ii alocă spaţiu şi o iniţializează cu valoarea specificată, a treia alocă un tablou de dimensiune *n*. Acest operator furnizează ca rezultat un pointer conţinînd adresa zonei de memorie alocate, în caz de succes, sau un pointer cu valoarea *NULL* (practic 0) cînd alocarea nu a reuşit.

Eliminarea unei variabile dinamice şi eliberarea zonei de memorie aferente se realizează cu ajutorul operatorului *delete*. Sintaxa acestuia este:

*delete variabilă;*

Deşi aceşti doi operatori oferă metode flexibile de gestionare a obiectelor, există situaţii în care aceasta nu rezolvă toate problemele. De aceea, pentru crearea şi distrugerea obiectelor în C++ se folosesc nişte funcţii membri speciale, numite *constructori* şi *destructori*, despre care s-a menţionat mai sus.

Să completăm în continuare clasa *Point* cu un constructor şi un destructor:

```
Point::Point() // constructor implicit
{ x = 0;  y = 0;  }
Point::Point(unsigned X, unsigned Y)
{ x = X;  y = Y;  }
Point::~Point() {    }
```

Aţi remarcat cu această ocazie modul de marcare a comentariilor în C++: tot ce se află după caracterul *//* este considerat comentariu.

Definiţii de forma

```
Point p;                          //sau
Point *p = new Point();
```

duc la apelarea constructorului implicit.

O întrebare care poate apare este motivul pentru care am realizat funcţiile GetX(), GetY(), SetX(), SetY(), cînd puteam utiliza direct variabilele membri x şi y. Deoarece una din regulile programării C++ este de a proteja variabilele membri, acestea pot fi accesate numai prin intermediul unor funcţii, care au rolul de metode de prelucrare a datelor incapsulate în interiorul clasei.

***Funcţii şi Clase friend.*** Conceptul friend permite abateri controlate de la ideea proiecţiei datelor prin incapsulare. Mecanismul de friend (sau prietenie) a apărut datorita imposibilităţii ca o metodă să fie membru a mai multor clase.

***Funcţiile prietene*** sînt funcţii care nu sînt metode ale unei clase, dar care au totuşi acces la membrii privaţi ai acesteia. Orice funcţie poate fi *prietenă* a unei clase, indiferent de natura acesteia.

Sintaxa declarării unei funcţii prietene în cadrul declaraţiei unei clase este următoarea:

```
friend NumeFuncţie
```

De exemplu:

```
class Point {
friend unsigned long Calcul(unsigned X, unsigned Y);
public:
friend unsigned long AltăClasă::Calcul(unsigned X,
                                       unsigned Y);
...     };
unsigned long Calcul(unsigned X, unsigned Y)
{return X * Y / 2; }
unsigned long AltăClasă::Calcul(unsigned X, unsigned Y)
    {       ...       }
```

***Funcţii membri ca prietene.*** Orice funcţie membru nu poate fi prietenă aceleiaşi clase, dar, posibil, să fie prietena altei clase. Astfel, funcţiile *friend* constituie o punte de legătură între clase. Există două moduri de a face ca o funcţie membru a unei clase să fie prietena altei clase.

Prima variantă este specificarea funcţiei membru a unei clase, ca fiind prietenă altei clase. Deci, în cea de-a doua clasă, vom declara funcţia membru în prima clasă ca fiind de tip *friend.*

```
class B;
class A
{…..
void Al (B &x) ;
…..
);
class B
 {  friend void A::A1(B &x);  );
```

A doua variantă este declararea unei clase prietenă, astfel, că toate funcţiile sale membri, sînt, de fapt, prietene clasei în care un obiect de tipul primei clase este declarat ***friend***.

```
class B;
class A
{  void Al (B &x) ;   );
class B
{…
friend A;
…};
```

Indiferent de metodă, se impune predeclararea clasei, care va fi prietenă sau va conţine funcţii, care sînt prietene unei alte clase.

***Clasele prietene*** sînt clase care au acces la membrii privaţi ai unei clase. Sintaxa declarării unei clase prietene este:

*friend class NumeClasăPrietenă*

De exemplu:

```
class PrimaClasă {
...
};
class ADouaClasă {
...
friend class PrimaClasă;
};
```

Clasa PrimaClasă are acces la membrii privaţi ai clasei ADouaClasă.

*Relaţia de prietenie nu este tranzitivă*. Daca o clasa A este prietena a clasei B, şi clasa B este prietenă a unei clase C, aceasta nu înseamnă ca A este prietenă clasei C. De asemenea, *proprietatea de prietenie nu se moşteneşte în clasele derivate*. *Clase prietene* sînt utile în situaţia în care avem nevoie de clase, care să comunice între ele deseori, acestea aflîndu-se pe acelaşi nivel ierarhic. Pentru exemplificare, presupunem că vom implementa o stivă de caractere ca o listă simplu înlănţuită. Vom utiliza două clase, una ataşată nodurilor din listă şi una – stivei propriu-zise.

```
#include<conio.h>
#include<stdio.h>
class stiva;
class nod
{   private:
  friend stiva;
```

```
   nod(int d, nod *n);
   int data;
   nod *anterior;
};
class stiva
{   private:
   nod *virf;
  public:
  stiva ()   {      virf=NULL;    }
  ~stiva()   {      delete virf;    }
void push (int c);
int pop ();
};
nod::nod (int d, nod *n) {data=d; anterior=n;}
void stiva::push (int i)    {nod *n=new nod(i, virf); virf=n;  }
int stiva::pop ()
   {   nod *t=virf;
   if (virf)
     {   virf=virf->anterior;
     int c= t->data;
     delete t;
     return c;   }
     return -1;
}
void main()
{  int c;
  stiva cs;
printf("Introdu un sir de caractere, ce se termina in *");
   while ((c=getch ())!='*')
      {    cs.push (c);
      putch(c); }
     putch(c);
          while ((c=cs.pop ())!=-1)
         {     putch (c);   }
      c='\n';   putch(c);
}
```

Rezultatul îndeplinirii programului:

*Introdu un sir de caractere, ce se termina in *ertyuioppoiu*uiop*

*poiuytre*

Prototipul unei funcţii *friend* cu o clasă se află, de regulă, în cadrul clasei respective. Funcţia *friend* nu este membru a acestei clase. Indiferent de poziţia declaraţiei unei asemenea funcţii, în cadrul declaraţiei clasei funcţia va fi publică. De exemplu:

```
   class punct
   { private:
         int x, y;
   public:
```

```
 punct (int xi, int yi) {x=xi; y=yi; };
 friend int compara (punct &a, punct &b);
};
int compara (punct &a, punct &b)
{   //returnează    <0, dacă a este mai aproape de origine
   //                  >0, dacă b este mai aproape de origine
   //                 =0, dacă a şi b sînt egal depărtate.
   return a. x*a. x+a. y*a. y-b. x*b. x-b. y*b. y;   }
 void main()
{ punct p (14,17), q(57,30);
if(compara(p,q)<0) printf("p este mai apropiat de origine\n");
else printf (“q este mai apropiat de origine. \n") ;   }
```

Orice funcţie *friend* a unei clase poate fi transformată într-o funcţie membru a acelei clase, renunţîndu-se însă la gradul de “prietenie”. Exemplul de mai sus se modifică în felul următor:

```
#include<stdio.h>
class punct
{ private:
      int x,y;
public:
punct (int xi, int yi)   { x=xi;  y=yi; }
int compara (punct &b);
};
int punct:: compara (punct &b)
{ return x*x+y*y-b.x*b.x-b.y*b.y;  }
void main ()
 {  punct p(14,17), q(57,30);
if (p.compara (q)<0)printf ("p este mal apropiat de origine\n");
else  printf ("q este mai apropiat de origine.\n");
}
```

Rezultatul îndeplinirii programului:

*p este mal apropiat de origine*

**Tema pentru lucrărea de laborator:**

Să se scrie un program pentru implimentarea unei clase care include un set de date specifice cît și următoarele metode:

- Constructor prestabilit,
- Constructor de copiere,
- Constructor cu parametri,
- Metodă de inițializare a datelor,
- Metodă de afișare a datelor,
- Metode de returnare și afișare separată a datelor(opțional).

Clasa va conține metode descrise atît în interior cît și în exteriorul ei. Cel puțin o metod va fi declarată ca funcție prietenă. După posibilități să se utilizeze pointerul **this** și operatorul rezoluție( **::**).

Sarcină individuală.

Varianta:

1. Apartament – Numar de odăi, suprafața, etajul.
2. Motocicletă – Numele de firmă, numarul de roți, volumul motorului.
3. Calculator – firma, viteza procesorului, capacitatea memoriei.
4. Camion – numele de firma, tonajul, distanșa parcursă.
5. Continent – emisfera, suprafața ,numarul de locuitori.
6. Elev – numele, clasa, balul mediu.
7. Film – denumirea, anul emiterii, costul filmării.
8. Medic – specialitate, staj, salariul.
9. Oraș – continent, țara, număr de locuitori.
10. Patrulater – numele, numărul de laturi, suprafața.
11. Piramidă – forma bazei, inalțimea, suprafața totală.
12. Profesor – numele, specialitatea, vîrsta.
13. Rîu – continent, lungimea, debitul de apă.
14. Sportiv – numele, țara, numărul de medalii.
15. Țară – continentul, numarul de locuitori, suprafața.