Programare in C++, POO

2. DEFINIREA DE CLASE IN C++

2.1 Sintaxa definirii unei clase

Pentru definirea unei clase se poate folosi unul din cuvintele cheie "class", "struct" sau "union", cu urmatoarele efecte asupra atributelor de accesibilitate ale membrilor clasei:

- O clasa definita prin "class" are implicit toti membri invizibili din afara clasei (de tip "private").
- O clasa definita prin "struct" (sau "union") are implicit toti membri publici, vizibili din afara clasei.

In practica avem nevoie ca datele clasei sa fie ascunse (private) si ca functiile clasei sa poata fi apelate de oriunde (publice). Pentru a stabili selectiv nivelul de acces se folosesc cuvintele cheie "public", "private" si "protected", care determina mai multe sectiuni in cadrul unei clase. In mod uzual, o clasa are doua sectiuni: sectiunea de date ("private") si sectiunea de metode ("public"). Atributul "protected" se foloseste in clasele din care se vor deriva alte clase, pentru date (sau functii) ce trebuie sa fie accesibile numai claselor derivate.

Exemplul urmator arata o posibilitate de definire a unei clase "Byte", de numere intregi pozitive cuprinse intre 0 si 255.

Clasa "Byte" contine ca date un numar intreg ("val") si ca metode doua functii pentru acces la date: "getVal" asigura citire (utilizarea) si "setVal" permite modificarea valorii "val".

```
   // varianta 1 de definire
class Byte {
  int val;    // valoare intreaga
public:
   void setVal (int v ) {
   assert (v >=0 && v < 256); val=v; }
   int  getVal () { return val; }
};
```

De retinut ca definitia unei clase trebuie terminata cu punct si virgula (';'), ca orice alta definitie de tip sau declaratie.

Justificarea acestei clase este aceea ca permite semnalarea depasirii limitelor pentru numere de un octet, ceea ce nu este posibil atunci cand se foloseste direct tipul predefinit "unsigned char". In plus, afisarea unor variabile de tip "unsigned char" la "cout" ca numere nu se poate face fara o conversie explicita de tip:

```
unsigned char b;
cout << (int)b;  // altfel se afiseaza caracterul cu codul b
```

In prima varianta de definire a clasei "Byte", functiile clasei sunt definite in cadrul clasei, ceea ce se si recomanda pentru functiile mici. Functiile definite in clasa sunt automat de tip "inline".

In a doua varianta de definire functiile membru sunt doar declarate (anticipat) in clasa si sunt definite ulterior, in afara clasei. In orice declaratie anticipata (prototip de functie) numele parametrilor formali pot lipsi, fiind suficient tipul lor.

Se recomanda ca definitia unei clase sa fie plasata intr-un fisier antet separat, care va fi inclus de toate fisierele sursa ce folosesc tipul respectiv. Definitiile functiilor membre se face de obicei intr-un fisier sursa separat de celelalte fisiere ale aplicatiei; eventual se compileaza si se introduc intr-o biblioteca. In acest fel este separata descrierea (specificarea) clasei de implementarea clasei.

```
// fisier BYTE.H
     class Byte {
```

```
public:
  void setVal (int );
  int  getVal () ;
private:            // sectiune de date
   int val;
};
   // fisier BYTE.CPP
 #include "byte.h"
  // stabileste o valoare ptr. un obiect Byte
void inline Byte:: setVal (int v)  {
    assert (v >=0 && v < 256);
    val=v;
}
  // obtine valoare obiect de tip Byte
int inline Byte:: getVal () {
    return val;
}
    // fisier TESTBYTE.CPP - utilizare
 #include "byte.h"
 #include <iostream.h>
 void main () {
        Byte b1,b2;
        int x ;
   do {
     cin >> x;      // nu  cin >> b1
     b1.setVal(x);  // nu  b1=x
     b2=b1 ;        // sau  b2.setVal (b1.getVal());
     cout << b2.getVal() << '\n";  // nu cout << b2;
  } while (cin);   // pana la Ctrl-Z
 }
```

In practica, numarul functiilor membre este mai mare si un fisier antet poate reuni definitiile unor clase inrudite, pentru a reduce numarul fisierelor antet si numarul directivelor "include".

Componentele publice ale unei variabile din clasa Byte se folosesc la fel cu componentele unei variabile de un tip "struct" sau "union", precedate de numele variabilei. Desi exista o singura functie "setVal" pentru toate variabilele de tip "Byte", efectul acestei functii depinde de variabila pentru care se executa : efectul apelului b1.setVal(x) este echivalent cu b1.val = x si este evident diferit de efectul apelului b2.setVal(x) (care este b2.val=x).

Pentru variabile de un tip clasa se pot folosi numai doi operatori, fara a mai fi definiti. Acestia sunt operatorul de atribuire ('=') si operatorul de obtinere a adresei variabilei ('&'). La atribuirea intre variabile de un tip clasa se copiaza numai datele clasei (cu exceptia claselor ce contin metode virtuale).

2.2 **Constructori de obiecte**

Orice clasa are una sau mai multe functii constructor, avand numele clasei. De obicei se defineste explicit efectul functiei constructor, dar absenta unui constructor nu este o eroare sintactica pentru ca este generat automat de compilator un constructor cu efect nul.

Constructorul permite initializarea automata a datelor dintr-un obiect simultan cu crearea (cu definirea) obiectului, evitandu-se astfel erori de utilizare a unor obiecte neinitializate.

De remarcat ca functia constructor nu are tip, pentru ca ea nu este apelata explicit intr-un program; compilatorul genereaza apeluri ale functiei constructor la definirea unor variabile de un tip clasa, la crearea dinamica de obiecte (cu operatorul "new") si in alte situatii (transmitere de parametri, rezultate intermediare s.a.)

De multe ori functiile constructor folosesc parametri cu valori implicite, care simplifica declararea unor obiecte.

Exemplu de clasa cu un singur constructor:

```
class Byte {
  int val;    // valoare intreaga
public:
   Byte (int v =0 ) {
   assert (v >=0 && v < 256); val=v; }
   int  getVal () { return val; }
};
...
Byte b1, b2=2, b3 (3) ;   // 3 forme de apelare constructor
Byte * pb= new Byte (4);  // constructor apelat de new
...
```

Declaratia de mai sus este echivalenta cu declaratiile:

```
Byte b1= Byte(0), b2= Byte(2), b3 =Byte(3);
```

Existenta unui constructor explicit permite si conversia de la tipul argumentului la tipul clasei, deci atribuiri de forma:

```
b1= 7;          // echivalent cu  b1= Byte(7);
b2 = b1 + 9;    // b2= b1+ Byte(9)
```

La evaluarea expresiilor cu operanzi de tipuri diferite compilatorul incearca conversia unuia dintre operanzi pentru aducere la tipul celuilalt operand. Conversiile automate de tip se aplica si pentru variabile de un tip clasa, cu conditia sa existe un constructor corespunzator.

Un constructor cu parametru de tip "int" pentru clasa "Byte" asigura conversia de la tipul "int" la tipul "Byte". Aceeasi conversie se poate face si cu operatorul "cast", care face apel tot la constructor:

```
b1= (Byte) 7;
```

Conversia inversa, de la tipul "Byte" la tipul "int" este oarecum realizata de functia membru "getVal", dar se poate defini si un operator cu numele "int" pentru aceasta conversie.

Uneori sunt utile cateva functii constructor, cu acelasi nume, dar cu parametri diferiti ca tip si semnificatie. De exemplu, pentru o clasa "String", ce corespunde unui sir de caractere cu operatiile asociate, putem avea un constructor cu parametru pointer la caractere si un alt constructor cu parametru intreg ce reprezinta lungimea sirului:

```
// clasa pentru siruri de caractere
    class String {
      char * str;
    public:
      String (char * s) { int len=strlen(s);
             str=new char[len+1]; strcpy (str,s); }
      String (int lung) { str=new char[lung]; *str='\0'; }
      ~String () { delete str;}     // destructor
      ...
   };
```

Pentru clasa "String" mai este util sa definim si un al treilea constructor, numit constructor prin copiere ("copy constructor"), necesar pentru toate clasele care contin pointeri catre date alocate dinamic.

Un constructor prin copiere este generat automat de compilator pentru a fi folosit in situatiile cand este necesara construirea unui obiect temporar: la initializarea unui obiect pe baza

altui obiect, la transmiterea unui obiect ca parametru, la evaluarea unor expresii (pentru rezultate intermediare), la functii de un tip clasa.

Problema este ca acel constructor implicit copiaza toate datele unui obiect in obiectul temporar, iar pentru clase cu pointeri rezulta mai multe obiecte care contin adresa unei singure zone de date.

Exemplu de situatie anormala creata prin folosirea constructorului de copiere implicit.

```
// initializarea foloseste constr. de copiere
String s1="unu", s2=s1;
s2[0] = '2' ;  cout << s1;   // afiseaza "2nu"
{ String s3 =s1; }     // s3 distrus la iesirea din bloc
cout << s1 << endl;    // nu afiseaza corect
...
```

Constructorul de copiere necesar pentru clasa String arata astfel:

```
// constructor de copiere
   String::String (String & s) {
      int len = strlen (s.str);      // lungime sir original
      char * nou = new char[len+1];  // aloca memorie
      strcpy (str, s.str);           // si copiaza datele
   }
```

Aceeasi problema apare si la atribuirea intre obiectele unei clase cu pointeri, ceea ce face necesara redefinirea operatorului de atribuire pentru astfel de clase.

2.3 **Destructori de obiecte**

Obiectele unei clase, ca orice alte variabile, au o durata de viata si o clasa de alocare, rezultate din locul unde au fost definite. Deci putem avea obiecte locale unei functii, care exista in memorie numai cat este activa functia, si obiecte externe functiilor, mentinute pe toata durata executiei unui program. In plus, pot exista si obiecte alocate la cerere, fara nume, si adresate printr-un pointer; aceste obiecte dureaza pana la distrugerea lor explicita, prin eliberarea memoriei alocate.

Functia constructor este apelata la inceputul executiei pentru obiectele statice, la activarea unei functii pentru obiectele automatice (locale) si la executarea operatorului 'new' pentru obiectele alocate la cerere.

Exista si o functie pereche, numita "destructor", apelata automat la disparitia unui obiect: la terminarea programului pentru obiectele statice, la iesirea dintr-o functie pentru obiectele locale (si parametri functiei), la executarea operatorului "delete" pentru obiecte dinamice.

Functia destructor nu are tip, iar numele ei deriva din numele clasei, precedat de caracterul tilda ('~').

Functia destructor este definita explicit mult mai rar ca un constructor, deoarece efectul nul al destructorului implicit este suficient pentru obiectele care nu contin date alocate dinamic.

Pentru clase ce contin date alocate dinamic, constructorul trebuie sa realizeze si alocarea de memorie necesara, iar destructorul sa elibereze memoria alocata de constructor.

Exemplul urmator defineste o clasa "Vector" care contine un vector alocat dinamic, in functie de dimensiunea specificata la declararea fiecarui obiect.

```
// clasa care contine un pointer
    class Vector {
      int * vec;      // adresa vector alocat dinamic
      int dim;       // dimensiune vector
    public:
```

```
    Vector (int m=100)
      { vec=new int [dim]; dim=m; }
    ~Vector () { delete [] vec; }
    void setVal (int i, int x) { assert (i < dim); vec[i]=x; }
    int getVal (int i) { assert (i < dim); return vec[i]; }
  };
```

Aceasta solutie pentru un vector este mai flexibila, deoarece permite specificarea dimensiunii maxime separat pentru fiecare vector din fiecare aplicatie. Exemple de obiecte Vector:

```
Vector v1 ;       // v1 are dimens. 100
Vector v2 (30);   // v2 are dimens. 30
```

Destructorul clasei elibereaza toata memoria alocata pentru vector la construirea sa.

Pentru o lista inlantuita destructorul trebuie sa distruga toate nodurile ramase, deci foloseste repetat operatorul "delete".

Functia destructor este mai complexa pentru colectii de pointeri la date alocate dinamic (vectori sau liste inlantuite de pointeri).

In aceste cazuri trebuie distruse mai intai datele (obiectele) adresate de pointeri si apoi obiectul ce reprezinta colectia de pointeri. Daca se distruge doar lista de pointeri, atunci raman date in memoria dinamica (heap) care nu mai pot fi adresate pentru ca s-au pierdut adresele lor. Exemplul urmator contine definitia unei clase de liste inlantuite de pointeri la siruri de caractere.

```
#include <iostream.h>
#include <string.h>
   // lista de pointeri la siruri
class List {
  struct nod {
    char * str;
    nod * urm;
  } * list;
public:
   List() {list = NULL;}    // constructor lista
   ~List() ;                // destructor lista
   void add ( char * );
   void view() ;
};
   // destructor lista
List::~List () {
 while (list != NULL) {   // parcurge toata lista
   nod * p =list;         // adresa primului nod
   delete p->str;         // elib. mem. ocupata de sir
   list=list->urm;        // elimina primul nod
   delete p;              // elib. mem. ocupata de nod
  }
}
 ... // alte definitii de metode
```