Recapitularea unor principii ale dezvoltării pe bază de obiecte

1. Cum ați defini paradigma POO?

- *POO poate fi definită ca metoda/stil/paradigmă de programare prin care: programele sunt privite ca mulțimi de obiecte care cooperează, obiectele sunt instanțe ale unor clase și clasele fac parte dintr-o ierarhie de tipuri și sunt corelate prin relații de moștenire.*

2. Ce este o clasă?

- *Clasa reprezintă un tip de date definit de utilizator, care se comportă la fel ca un tip predefinit de date. Pe lângă variabilele folosite pentru descrierea datelor, se descriu şi metodele (funcţiile) folosite pentru manipularea lor.*

3. Ce este un obiect?

- *Instanța unei clase reprezintă un obiect. Instanţa este o variabilă declarată ca fiind de tipul clasei definite.*

4. Ce reprezinta membrii unei clase?

- *Variabilele declarate în cadrul unei clase se numesc variabile membru, iar funcțiile declarate în cadrul unei clase se numesc metode sau funcții membru. Metodele pot accesa toate variabilele declarate în cadrul clasei, private sau publice. Membrii unei clase reprezintă totalitatea metodelor şi a variabilelor membre ale clasei.*

5. Care este diferența între clase și structuri?

- *Diferenţa principală între specificatorii "class", "struct" şi "union" este următoarea: pentru o clasă declarată folosind specificatorul "class", datele membre sunt considerate implicit de tip private, până la prima folosire a unuia din specificatorii de acces public sau protected. Pentru o clasă declarată folosind specificatorul "struct" sau "union", datele membre sunt implicit de tip public, până la prima folosire a unuia din specificatorii private sau protected.*

6. Ce este încapsularea?

- *Incapsularea = contopirea datelor cu codul (metode de prelucrare si acces la date) în clase, ducând la o localizare mai bună a erorilor şi la modularizarea problemei de rezolvat.*

7. Care sunt funcţiile de intrare/ieşire în C++?

Limbajul C++ furnizează obiectele cin şi cout, în plus faţă de funcţiile scanf şi printf din limbajul C. Pe lângă alte avantaje, obiectele cin şi cout nu necesită specificarea formatelor.

8. Ce înseamnă supra-încarcarea funcţiilor în Limbajul C++?

- *Limbajul C++ permite utilizarea mai multor funcţii care au acelaşi nume, caracteristică numită supraîncărcarea funcţiilor. Identificarea lor se face prin numărul de parametri şi tipul lor.*

9. Ce sunt constructorii?

- *Constructorul este o metodă specială a unei clase, care este membru al clasei respective şi are acelaşi nume ca şi clasa. Constructorii sunt apelaţi atunci când se instanţiază obiecte din clasa respectivă, ei asigurând iniţializarea corectă a tuturor variabilelor membru ale unui obiect şi garantând că iniţializarea unui obiect se efectuează o singură dată.*

10. Cum se declară constructorii?

Constructorii se declară, definesc şi utilizează ca orice metodă uzuală, având următoarele proprietăţi distinctive:

- *poartă numele clasei căreia îi aparţin;*
- *nu pot returna valori; în plus (prin convenţie), nici la definirea, nici la declararea lor*
- *nu poate fi specificat "void" ca tip returnat;*
- *sunt apelaţi implicit ori de câte ori se instanţiază un obiect din clasa respectivă;*
- *în caz că o clasa nu are nici un constructor declarat de către programator, compilatorul*
- *va declara implicit unul. Acesta va fi public, fără niciun parametru, şi va avea o listă vidă de instrucţiuni;*
- *constructorii pot avea parametrii.*

11. O clasa poate avea mai mulţi constructori? Dacă da, atunci cum ştie compilatorul să facă diferenţierea între aceștia?

- *clasă poate avea oricâţi constructori, ei diferenţiindu-se doar prin tipul şi numărul parametrilor. Compilatorul apelează constructorul potrivit în funcţie de numărul şi tipul parametrilor pe care-i conţine instanţierea obiectului.*

12. Ce sunt destructorii?

Destructorul este complementar constructorului. Este o metodă care are același nume ca şi clasa căreia îi aparţine, dar este precedat de "~". Dacă constructorii sunt folosiți în special pentru a aloca memorie şi pentru a efectua anumite operaţii (de exemplu: incrementarea unui contor al numărului de obiecte), destructorii se utilizează pentru eliberarea memoriei alocate de constructori şi pentru efectuarea unor operaţii inverse (de exemplu: decrementarea contorului).

13. Ce reprezinta declarațiile *public* și *private*?

Membrii aparţinând secţiunii "public" pot fi accesaţi din orice punct al domeniului de existenţă al respectivei clase, iar cei care aparţin secţiunii "private" (atât date cât şi funcţii) nu pot fi accesaţi decât de către metodele clasei respective. Utilizatorul clasei nu va avea acces la ei decât prin intermediul metodelor declarate în secţiunea public (metodelor publice).

14. Ce este operatorul de rezoluție?

Definirea metodelor care aparţin unei clase se face prefixând numele metodei cu numele clasei, urmat de "::". Simbolul "::" se numeşte "scope acces operator" (operator de rezoluţie sau operator de acces) şi este utilizat în operaţii de modificare a domeniului de vizibilitate.

15. Ce sunt funcţiile prietene (*friend*)?

O funcţie friend este o funcţie care nu e membră a unei clase, dar are acces la membrii de tip private şi protected ai clasei respective. Orice funcţie poate fi friend unei clase, indiferent dacă este o funcţie obişnuită sau este membră a unei alte clase.

16. Ce sunt clasele prietene (*friend*)?

Dacă se doreşte ca toţi membrii unei clase "M" să aibă acces la partea privată a unei clase "B", în loc să se atribuie toate metodele lui "M" ca fiind friend ai lui "B", se poate declara clasa "M" ca şi clasă friend lui "B".

17. Ce este moştenirea?

Moştenirea = posibilitatea de a extinde o clasa prin adăugarea de noi funcționalități.
Considerând o clasă oarecare A, se poate defini o altă clasă B, care să preia toate caracteristicile clasei A, la care se pot adăuga altele noi, proprii clasei B. Clasa A

se numeşte clasă de bază, iar clasa B se numeşte clasă derivată. Acesta este numit "mecanism de moştenire".

18. Cum se realizează moştenirea?

În declaraţia clasei derivate nu mai apar informațiile care sunt moştenite, ele fiind automat luate în considerare de către compilator. Nu mai trebuie rescrise funcţiile membru ale clasei de bază, ele putând fi folosite în maniera în care au fost definite. Mai mult, metodele din clasa de bază pot fi redefinite (polimorfism), având o cu totul altă funcţionalitate.

19. Ce este polimorfismul?

Polimorfismul = într-o ierarhie de clase obținută prin moștenire, o metodă poate avea implementări diferite la nivele diferite în acea ierarhie.

20. Ce reprezinta cuvântul „static"?

Cuvântul cheie "static" poate fi utilizat în prefixarea membrilor unei clase. Odată declarat "static", membrul în cauză are proprietăți diferite, datorită faptului că membrii statici nu aparțin unui anumit obiect, ci sunt comuni tuturor instanțierilor unei clase. Pentru o variabilă membru statică se rezervă o singură zonă de memorie, care este comună tuturor instanţierilor unei clase (obiectelor).

Reguli si standarde POO C++

- Numele clasei trebuie să fie un substantiv la singular, scris cu majusculă (ex. Elev, Student, Cerc)
- Numele membrilor și metodelor trebuie să fie scrise în format camelCase (prima literă mică, iar dacă sunt mai multe cuvinte, acestea cu literă mare)
- Constructorul trebuie să aibă parametri și să primească valorile de instanțiere din altă sursă (nu citește constructorul de la tastatură)
- Getterele trebuie făcute pentru fiecare atribut în parte și trebuie să returneze valori (nu să afișeze)
- Setterele trebuie făcute pentru fiecare atribut în parte și să aibă parametri, nu să citească
- Folosiți oricâte metode pe lângă getters și setters
- Puteți folosi în metode parametri care au același nume cu membrii clasei, caz în care veți folosi cuvântul cheie this pentru a deosebi membri de parametri)

ex (constructor pentru o clasă Student cu 2 membri - nume și anStudiu):

```
Student (string nume, int anStudiu){
this->nume = nume; //this->nume face referire la membrul "nume"
this->anStudiu = anStudiu;
}
```

Mai departe, acest curs prezintă noţiunile de clasă şi obiect, precum şi aspecte referitoare la:

- definirea unei clase
- variabile şi funcţii membre
- declararea obiectelor
- constructori si destructori

I. Încapsularea prin intermediul claselor

Clasele şi obiectele folosite în POO îi permit programatorului să realizeze programe mai compacte decât cele scrise în limbajele neobiectuale. De asemenea, părţi din program pot fi mai uşor reutilizate şi noul program poate fi mai uşor depanat.

1. Încapsularea ca principiu al POO

În C++ încapsularea este îndeplinită prin două aspecte:

1. folosirea claselor pentru unirea structurile de date şi a funcţiilor destinate manipulării lor;
2. folosirea secţiunilor private şi publice, care fac posibilă separarea mecanismului intern de interfaţa clasei;

O clasă reprezintă un tip de date definit de utilizator, care se comportă întocmai ca un tip predefinit de date. Pe lângă variabilele folosite pentru descrierea datelor, se descriu şi metodele (funcţiile) folosite pentru manipularea lor.

Instanţa unei clase reprezintă un obiect - este o variabilă declarată ca fiind de tipul clasei definite. Variabilele declarate în cadrul unei clase se numesc variabile membru, iar funcţiile declarate în cadrul unei clase se numesc metode sau funcții membru. Metodele pot accesa toate variabilele declarate în cadrul clasei, private sau publice.

Membrii unei clase reprezintă totalitatea metodelor şi a variabilelor membre ale clasei.

Sintaxa declarării unei clase este următoarea:

```
specificator_clasa Nume_clasa
{
[ [ private : ] lista_membri_1]
[ [ public : ] lista_membri_2]
};
```

unde:

Specificatorul de clasă specificator_clasa poate fi:

- class
- struct
- union

Numele clasei (Nume_clasa) poate fi orice nume, în afara cuvintelor rezervate limbajului C++. Se recomandă folosirea de nume cât mai sugestive pentru clasele folosite, precum şi ca denumirea claselor să înceapă cu literă mare. (ex: class Elevi)

Folosind specificatorii de clasă “struct” sau “union” se descriu structuri de date care au aceleaşi proprietăţi ca şi în limbajul C (neobiectual), cu câteva modificări :

- se pot ataşa funcţii membru;
- pot fi compuse din trei secţiuni - privată, publică şi protejată (folosind specificatorii de acces private, public şi protected);

Diferenţa principală între specificatorii “class”, “struct” şi “union” este următoarea: pentru o clasă declarată folosind specificatorul “class”, datele membre sunt considerate implicit de tip private, până la prima folosire a unuia din specificatorii de acces public sau protected. Pentru o clasă declarată folosind specificatorul “struct” sau “union”, datele membre sunt implicit de tip public, până la prima folosire a unuia din specificatorii private sau protected. Specificatorul protected se foloseşte doar dacă este folosită moştenirea.

Descrierea propriu-zisă a clasei constă din cele doua liste de membri, prefixate de cuvintele cheie “private” şi/sau “public”. Membrii aparţinând secţiunii “public” pot fi accesaţi din orice punct al domeniului de existenţă al respectivei clase, iar cei care aparţin secţiunii “private” (atât date cât şi funcţii) nu pot fi accesaţi decât de către metodele clasei respective. Utilizatorul clasei nu va avea acces la ei decât prin intermediul metodelor declarate în secţiunea public (metodelor publice).

Definirea metodelor care aparţin unei clase se face prefixând numele metodei cu numele clasei, urmat de “::”. Simbolul “::” se numeşte “scope acces operator” (operator de rezoluţie sau operator de acces) şi este utilizat în operaţii de modificare a domeniului de vizibilitate.

Exemplu:

```
class Stiva
{
int varf;
int st[30];
public:
void init (void);
…
};
void Stiva :: init (void)
{
…
}
```

În stânga lui “::” nu poate fi decât un nume de clasă sau nimic, în cel de-al doilea caz prefixarea variabilei folosindu-se pentru accesarea unei variabile globale. În lipsa numelui clasei în faţa funcţiei membru nu s-ar putea face distincţia între metode care poartă nume identice şi aparţin de clase diferite.

Exemplu:

```
class Stiva
{
public:
```

```
void init (void);
…
};

class Elevi
{
public:
void init (void);
…
};
void Stiva::init (void) // metoda clasei Stiva
{
…
}
void Elevi::init (void) // metoda clasei Elevi
{
…
}
```

Accesarea membrilor unui obiect se face folosind operatorul "." Dacă obiectul este accesat indirect, prin intermediul unui pointer, se foloseste operatorul "->" După cum s-a mai spus, variabilele membru private nu pot fi accesate decât de metode care aparţin clasei respective.

II. Constructori şi destructori

1. Constructori

Constructorul este o metodă specială a unei clase, care este membru al clasei respective şi are acelaşi nume ca şi clasa. Constructorii sunt apelaţi atunci când se instanţiază obiecte din clasa respectivă, ei asigurând iniţializarea corectă a tuturor variabilelor membru ale unui obiect şi garantând că iniţializarea unui obiect se efectuează o singură dată.

Constructorii se declară, se definesc şi se utilizează ca orice metodă uzuală, având următoarele proprietăţi distinctive:

- poartă numele clasei căreia îi aparţin;
- nu pot returna valori; în plus (prin convenţie), nici la definirea, nici la declararea lor nu poate fi specificat "void" ca tip returnat;
- adresa constructorilor nu este accesibilă utilizatorului; expresii de genul "&X :: X()" nu sunt disponibile;
- sunt apelaţi implicit ori de câte ori se instanţiază un obiect din clasa respectivă;
- în caz că o clasa nu are nici un constructor declarat de către programator, compilatorul
- va declara implicit unul. Acesta va fi public, fără nici un parametru, şi va avea o listă vidă de instrucţiuni;

- în cadrul constructorilor se pot utiliza operatorii "new" si "delete",
- constructorii pot avea parametrii.

O clasă poate avea oricâţi constructori, ei diferenţiindu-se doar prin tipul şi numărul parametrilor. Compilatorul apelează constructorul potrivit în funcţie de numărul şi tipul parametrilor pe care-i conţine instanţierea obiectului.

Tipuri de constructori

O clasă poate conţine două tipuri de constructori:

- constructor implicit (“default constructor”);
- constructor de copiere (“copy constructor”);

Constructorii impliciţi se pot defini în două moduri:

- definind un constructor fără nici un parametru;
- prin generarea sa implicită de către compilator. Un astfel de constructor este creat ori de câte ori programatorul declară o clasă care nu are nici un constructor. În acest caz, corpul constructorului nu conţine nici o instrucţiune.

O clasă poate conţine, de asemenea, constructori de copiere. Constructorul de copiere generat implicit copiază membru cu membru toate variabilele argumentului în cele ale obiectului care apelază metoda. Compilatorul generează implicit un constructor de copiere în fiecare clasă în care programatorul nu a declarat unul în mod explicit.

Exemplu:

```
class X {
X (X&); // constructor de copiere
X (void); // constructor implicit
};
```

Apelarea constructorului se copiere se poate face în următoarele moduri:

```
X obiect2 = obiect1;
```

sau sub forma echivalentă:

```
X obiect2 (obiect1);
```

Supraincarcarea constructorilor

Precum alte functii si constructorul poate fi supraincarcat cu oricate functii care au acelasi nume, dar numar si tip diferit de parametrii. Compilatorul va compila apelul la acea functie care se potriveste atat la numarul de parametrii, cat si la tipul acestora. In cazul in care aceasta potrivire nu se poate efectua, se va genera o eroare de compilare.

Exemplu (de testat)

```
#include <iostream.h>

class CRectangle {
        int width, height;
```

```
    public:
        CRectangle (int,int);
        int area (void) { return (width*height); }
 };

CRectangle::CRectangle (int a, int b) {
    width = a;
    height = b;
}

int main () {
    CRectangle rect (3,4);
    CRectangle rectb (5,6);
    cout << "rect area: " << rect.area()<< endl;
    cout << "rectb area: " <<rectb.area() << endl;
}
```

Completați / modificați exemplul de mai sus pentru a ilustra conceptul de supraîncărcare a constructorului.

2. Destructorii

Destructorul are o functionalitate opusa constructorului. Acesta este chemat in mod automat atunci cand un obiect este eliberat din memorie, fie pentru ca scopul existentei obiectului s-a terminat (de exemplu cand este declarata o instanta a unei clase in interiorul unei functiei, iar aceasta se termina de executat) fie pentru ca obiectul a fost alocat dinamic (cu new) si este eliberat utilizand operatorul delete.

Destructorul are intotdeauna acelasi nume ca si clasa si este precedat de tilda (~) ca un prefix. Totodata, la fel ca in cazul constructorului, destructorul nu returneaza nici o valoare. Utilitatea destructorului apare atunci cand un obiect atribuie memorie dinamica pe parcursul existentei acestuia si, in momentul cand este distrus, trebuie sa elibereze memoria alocata lui.

```
// examplu constructor si deconstructor
#include <iostream.h>
class CRectangle {
    int *width, *height;
public:
    CRectangle (int,int);
    ~CRectangle ();
    int area (void) {return (*width * *height);}
};
```

```
CRectangle::CRectangle (int a, int b) {
width = new int;
height = new int;
*width = a;
*height = b;
}
CRectangle::~CRectangle () {
delete width;
delete height;
}
int main () {
CRectangle rect (3,4), rectb (5,6);
cout << "rect area: " << rect.area()<< endl;
cout << "rectb area: " <<rectb.area() << endl;
return 0;
}
```

Destructorul este complementar constructorului. Este o metodă care are acelaşi nume ca şi clasa căreia îi aparţine, dar este precedat de "~". Dacă constructorii sunt folosiţi în special pentru a aloca memorie şi pentru a efectua anumite operaţii (de exemplu: incrementarea unui contor al numărului de obiecte), destructorii se utilizează pentru eliberarea memoriei alocate de constructori şi pentru efectuarea unor operaţii inverse (de exemplu: decrementarea contorului).

Exemplu:

```
class exemplu
{
public :
exemplu (); // constructor
~exemplu (); // destructor
};
```

Utilizatorul dispune de două moduri pentru a apela un destructor:

1. prin specificarea explicită a numelui său – metoda directă;

Exemplu:

```
class B
{
public:
~B();
};
void main (void)
{
B b;
b.B::~B(); // apel direct : e obligatoriu prefixul "B::"
}
```

2. folosind operatorul “**delete**” (metodă indirectă – a se vedea linia 10 din programul următor).
Exemplu (este menţionată ordinea executării):

```
#define NULL 0
struct s
{
int nr;
struct s *next;
};
class B
{
int i;
struct s *ps;
public:
B (int);
~B (void);
};
B :: B (int ii = 0) // 3 si 7
{
ps = new s; ps->next = NULL; i = ps->nr = ii; // 4 si 8
} // 5 si 9
B :: ~B (void) // 11 si 14
{
delete ps; // 12 si 15
} // 13 si 16
void main (void) // 1
{
B *pb;
B b = 9; // 2
pb = new B(3); // 6
delete pb; // 10
}
```

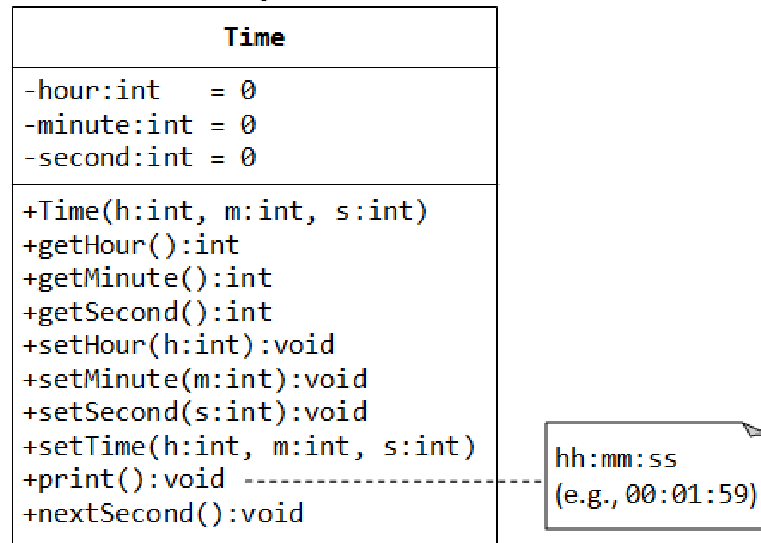
Exerciții:

1. Definiți şi implementați clasa Cerc, având ca dată membră Raza, şi ca funcții membre Aria şi Circumferința. Exploatați clasa cerc prin citirea de la tastatură a unei raze iar programul sa afişeze aria și circumferința cercului de raza data.
2. Definiți și implementați clasa Carte, având ca date membre: Nume, Autor, Nrpag, Pret, și ca funcții membre: citeste_carte si afiseaza_carte. Exploatați clasa Carte prin citirea de la

tastatura a unei cărți (numele, autorul, nr. de pagini si pretul) după care afișați cartea introdusa pe ecran. Clasa Carte are interfața de mai jos:

```
class carte{
        char nume[40];
        char autor[40];
        int nrpag;
        double pret;
public:
        void citeste_carte(char *numecarte, char *autorcarte,
              int *np, double *p);
        void afiseaza_carte();
};
```

3. Să se implementeze clasa Time după următoarea structură:



Bibliografie

Corina Rotar. PROGRAMARE ORIENTATĂ OBIECT. Note de curs. UNIVERSITATEA „1 DECEMBRIE 1918” ALBA IULIA. Seria Didactică.

Dorin Berian, Adrian Cocoş. PROGRAMARE ORIENTATĂ PE OBIECTE. Îndrumător de laborator. Universitatea „Politehnica” din Timişoara Facultatea de Automatică şi Calculatoare.

Rezolvări:

```
#include <iostream>
using namespace std;
class Cerc {
    int raza;
  public:
    void cin_raza(int);

    float arie_cerc()
    {
        return 3.1415*raza*raza;
    }

    float circum_cerc()
    {
        return 2*3.1415*raza;
    }
};

void Cerc::cin_raza(int x)
{
  raza = x;
}

int main()
{
    int r; Cerc ob;
    cout<<"Introduceti raza cercului: "; cin >> r; ob.cin_raza(r);
    cout<<"Aria cercului este: " << ob.arie_cerc() << endl;
    cout<<"Circumferinta cercului este: " << ob.circum_cerc() <<
endl;
    return 0;
}
```

Ex 2:

```
#include <iostream>
using namespace std;
class Cerc{
```

```
    float raza;
public:
    void citire_raza(float raz);
    void afisare_aria();
    void afisare_circuferinta();
};
void Cerc::citire_raza(float raz){
    raza = raz;
}
void Cerc::afisare_aria(){
    float aria;
    float pi;
    pi=3.14;
    aria = pi*raza*raza;
    cout<<"Aria cercului este: "<<aria;
}
void Cerc::afisare_circuferinta(){
    float p;
    float pi;
    pi=3.14;
    p = 2*pi*raza;
    cout<<"Circuferinta cercului este: "<<p;
}
int main()
{
    Cerc c;
    c.citire_raza(2.33);
    c.afisare_aria();
    cout<<endl;
    c.afisare_circuferinta();
    return 0;
}
```

```
#include <iostream>
#include <stdio.h>
#include <string>
using namespace std;
class Carte{
    string nume;
    string autor;
```

```
    int nrpg;
    double pret;
    public:
    void citire_carte(string numecartem, string autorcarte, int np,
double p);
    void afisare_carte();
};
void Carte::citire_carte(string numecarte, string autorcarte, int
np,double p){
    nume = numecarte;
    autor = autorcarte;
    nrpg = np;
    pret = p;



}
void Carte::afisare_carte(){
    cout<<"Numele cartii: "<<nume<<endl;
    cout<<"Autorul cartii: "<<autor<<endl;
    cout<<"Numar de pagini: "<<nrpg<<endl;
    cout<<"Pretul cartii: "<<pret;
}
int main(){
    Carte c;
    c.citire_carte("gh","hjhj",100,135);
    c.afisare_carte();
    return 0;
}
```