Universitatea „Politehnica” din Timişoara
Facultatea de Automatică şi Calculatoare

Dorin Berian Adrian Cocoş

PROGRAMARE ORIENTATĂ PE OBIECTE

Îndrumător de laborator

Cuvânt înainte

Acest îndrumător se adresează studenților din anul II Ingineria Sistemelor şi anul II Informatică (Facultatea de Automatică şi Calculatoare de la Universitatea Politehnica din Timişoara) la disciplina Programare Orientată pe Obiecte.

Materialul cuprinde 9 lucrări de laborator care cuprind atât parte teoretică cât şi parte aplicativă cu scopul deprinderii stilului de programare specific limbajelor de programare orientate pe obiecte, în particular limbajul C++.

Dorin Berian

Cuprins

Laborator 1 POO:

Completări aduse de limbajul C++ față de limbajul C

Obiectivul laboratorului: Formarea unei imagini generale, preliminare, despre programarea orientată pe obiecte (**POO**) şi deprinderea cu noile facilitați oferite de limbajul **C++**.

Beneficiul: Completările aduse limbajului C vin în sprijinul programatorului şi îi oferă acestuia noi „instrumente" de lucru, permițându-i să realizeze programe mult mai compacte, într-un mod avantajos din mai multe puncte de vedere: modularizare, fiabilitate, reutilizarea codului etc. De exemplu, supraîncărcarea funcțiilor permite reutilizarea numelui funcției şi pentru alte funcții, mărind astfel lizibilitatea programului.

Cuprins: Laboratorul trata aspecte referitoare la:
- intrări şi ieşiri;
- supraîncărcarea funcțiilor;
- alocarea dinamică a memoriei (operatorii *new* şi *delete*);
- parametrii cu valori implicite;
- transferul prin referință;

1. Conceptele POO

Principalele concepte (caracteristici) ale **POO** sunt:

- **încapsularea** – 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;
- **moştenirea** - posibilitatea de a extinde o clasa prin adaugarea de noi functionalitati
- **polimorfismul** – într-o ierarhie de clase obtinuta prin mostenire, o metodă poate avea implementari diferite la nivele diferite in acea ierarhie;

2. Intrări şi ieşiri

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

Exemplu:

```
cin >> variabila;
cout << "sir de caractere" << variabila << endl;
```

Utilizarea acestora necesita includerea header-ului bibliotecii de stream-uri, "iostream.h". Un **stream** este un concept abstract care desemnează orice flux de date de la o sursă la o destinație. Concret, stream-urile reprezintă totalitatea modalităților de realizare a unor operații de citire sau scriere. Operatorul >> are semnificația de "pune la ...", iar << are semnificația de "preia de la ...".
Exemplu:

```
#include <iostream.h>

void main(void)
{
```

```
    int a;
    float b;
    char c[20];
    cout <<"Tastati un intreg : "<< endl;
    cin >> a;
    cout << "Tastati un numar real : " << endl;
    cin >> b;
    cout << "Tastati un sir de caractere : " << endl;
    cin >> c;
    cout << "Ati tastat " << a << ", " << b << ", " << c << ".";
    cout << endl;
}
```

3. Supraîncărcarea funcțiilor. Funcții cu parametrii impliciți

Supraîncărcarea funcțiilor

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

Exemplu:

```
    int suma (int a, int b)
    {
        return (a + b);
    }

    float suma (float a, float b)
    {
        return (a + b);
    }
```

Dacă se apelează *suma (3,5)*, se va apela funcția corespunzătoare tipului *int*, iar dacă se apelează *suma (2.3, 9),* se va apela funcția care are parametrii de tipul *float*. La apelul funcției *suma (2.3, 9),* tipul valorii "9" va fi convertit automat de C++ în *float* (nu e nevoie de typecasting).

Funcții cu valori implicite

Într-o funcție se pot declara valori implicite pentru unul sau mai mulți parametri. Atunci când este apelată funcția, se poate omite specificarea valorii pentru acei parametri formali care au declarate valori implicite. Valorile implicite se specifică o singură dată în definiție (de obicei în prototip). Argumentele cu valori implicite trebuie să fie amplasate la sfârşitul listei.

Exemplu:

```
    void adunare (int a=5, double b=10)
    {
        ... ;
    }
    ...
    adunare ();     // <=> adunare (5, 10);
    adunare (1);    // <=> adunare (1, 10);
    adunare (1, 4); // <=> adunare (1, 4);
```

4. Operatorii *new* şi *delete*

Limbajul C++ introduce doi noi operatori pentru alocarea dinamică de memorie, care înlocuiesc familiile de funcţii "**free**" şi "**malloc**" şi derivatele acestora.

Astfel, pentru alocarea dinamică de memorie se foloseşte operatorul **new**, iar pentru eliberarea memoriei se foloseşte operatorul **delete**.

Operatorul "**new**" returnează un pointer la zona de memorie alocată dinamic (dacă alocarea se face cu succes) şi NULL dacă alocarea de memorie nu se poate efectua.

Operatorul "**delete**" eliberează zona de memorie la care pointează argumentul său.

Exemplu:

```
struct sistem
{
      char nume[20];
      float disc;
      int memorie;
      int consum;
};

struct sistem *x;

void main(void)
{
      x = new sistem;

      x->disc = 850;
      x->memorie = 16;
      x->consum = 80;

      delete x;
}
```

Programul prezentat defineşte un pointer la o structură de tip *sistem* şi alocă memorie pentru el, folosind operatorul **new**. Dezalocarea memoriei se face folosind operatorul **delete**.

Dacă se doreşte alocarea de memorie pentru un tablou de elemente, numărul elementelor se va trece după tipul elementului pentru care se face alocarea.

Exemplu:

```
x = new sistem[20];    - alocă memorie pentru 20 de elemente de tip sistem;
delete[] x;            - eliberarea memoriei
```

5. Transferul prin referință

O **referință** este un alt nume al unui obiect (variabila).

Pentru a putea fi folosită, o referință trebuie inițializată in momentul declararii, devenind un alias (un alt nume) al obiectului cu care a fost inițializată.

Folosind transferul prin referință, în funcții nu se va mai transmite întreaga structură, ci doar adresa ei. Membrii structurii pot fi referiți folosind "." sau "->" – pentru pointeri. În cazul utilizării referințelor, se lucrează direct asupra obiectului referit.

Considerând un obiect “x”, prin "&x" se înțelege “referință la obiectul x”.

Exemplu:

```
void ex (int i)
{
     i = 7;
}

int n = 3;
ex (n);
cout << n; - se va afişa 3
```

```
void ex (int &i)
{
     i = 7;
}

int n = 3;
ex (n);
cout << n; - se va afişa 7
```

Referința nu este un pointer către obiectul referit, este un alt nume al obiectului!

6. Alte noutăți aduse de C++ față de C

a) Comentarii de sfârşit de linie

Limbajul C admite delimitatorii C “/* */” pentru comentarii care se pot întinde pe mai multe linii. C++ introduce delimitatorul “//” pentru adăugarea mai comodă de comentarii de sfârşit de linie. Tot textul care urmează după “//” până la sfârşitul liniei este considerat comentariu.

Exemplu:

```
if (i == 20) break; //iesire fortata din ciclu
```

b) Plasarea declarațiilor

Limbajul C impune gruparea declarațiilor locale la începutul unui bloc. C++ elimină acest inconvenient, permițând declarații în interiorul blocului, de exemplu imediat înainte de utilizare. Domeniul unei astfel de declarații este cuprinsă intre poziția declarației şi sfârşitul blocului.

Exemplu: (rulați următoarea secvență de program)

```
void main(void)
{
     int i;
     cin >> i;
     int j = 5*i-1; //declaratie şi initializare de valoare
     cout << j;
}
```

c) Operatorul de rezoluție (operator de acces, operator de domeniu)

Limbajul C++ introduce operatorul de rezoluție (::), care permite accesul la un obiect (sau variabilă) dintr-un bloc în care acesta nu este vizibil, datorită unei alte declarații.

Exemplu: (rulați următoarea secvență de program)

```
char s[20]= "variabila globala";

void afiseaza(void)
{
      char s[20] = "variabila locala";
      cout << ::s;          //afiseaza variabila globală
      cout << s;            //afiseaza variabila locala
}
```

d) **Funcții "inline"**

C++ oferă posibilitatea declarării funcțiilor **inline,** care combină avantajele funcțiilor propriu-zise cu cele ale macrodefinițiilor. Astfel, la fiecare apelare, corpul funcției declarate inline este inserat în codul programului de către compilator.

Față de macrodefiniții (care presupun o substitutie de text într-o fază preliminară compilării), pentru funcțiile *inline* compilatorul inserează codul obiect al funcției la fiecare apel. Avantajul creşterii de viteza se plăteşte prin creşterea dimensiunii codului. Aşadar, funcțiile *inline* trebuie să fie scurte.

Exemplu:

```
      inline int comparare(int a,int b)
      {
            if (a>b) return 1;
            if (a<b) return 0;
            if (a==b) return -1;
      }
```

Partea practică a laboratorului:

Aplicația 1

Să se realizeze un program care preia de la tastatură următoarele informații: nume, prenume, vârsta, adresă, telefonul unei persoane. După preluare, aceste informații trebuie afişate.

Aplicația 2

Să se modifice următorul program astfel încât să devină funcțional:

```
      #include <iostream.h>

      void funcție(int a=123, double b, double c=123.456, char *s="prg")
      {
            cout << "\n a=" << a << " b=" << b << " c=" << c << " s=" << s;
      }
```

```
void main(void)
{
      funcție(456, 4.5, 1.4, "apel 1");
      funcție(456, 4.5, 1.4);
      funcție(456, 4.5);
      funcție(456.5);
}
```

Aplicația 3

Să se realizeze un program care calculează produsul a două numere reale şi a două numere complexe, specificate prin parte reala şi parte imaginară. Funcțiile de calcul al produselor vor avea acelaşi nume şi parametri diferiți.

Întrebări:

1. Ce este încapsularea?
2. Ce este moştenirea?
3. Ce este polimorfismul?
4. Care sunt funcțiile de intrare/ieşire în C++?
5. Unde trebuiesc plasate argumentele cu valori implicite?
6. Ce înseamnă supraîncarcarea funcțiilor în Limbajul C++?
7. Care sunt operatorii de alocare şi dezalocare de memorie în limbajul C++?
8. Ce este o referință ?
9. Referința este un pointer?
10. Ce este operatorul de rezoluție?
11. Unde se pot plasă declarațiile de variabile în cadrul Limbajului C++?

Laborator 2 POO:

Încapsularea prin intermediul claselor

Scopul laboratorului: Prezentarea noțiunilor de **clasă** şi **obiect**.

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

Scurtă prezentare: Acest laborator prezintă noțiunile de **clasă** şi **obiect**, precum şi aspecte referitoare la:

- definirea unei clase;
- variabile şi funcții membre;
- declararea obiectelor;

1. Încapsularea ca principiu al POO

În **C++** încapsularea este îndeplinită prin două aspecte:

1. folosirea claselor pentru unirea structurile de date şi a funcțiilor destinate manipulării lor;
2. folosirea secțiunilor **private** şi **publice,** care fac posibilă separarea mecanismului intern de interfața 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 (funcțiile) folosite pentru manipularea lor.

Instanța 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 funcțiile declarate în cadrul unei clase se numesc **metode** sau **functii 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]
};
```

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 proprietăți ca şi în limbajul C (neobiectual), cu câteva modificări :

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

Diferența 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 membrii, prefixate de cuvintele cheie "**private**" şi/sau "**public**".

Membrii aparținând secțiunii "**public**" pot fi accesați din orice punct al domeniului de existență al respectivei clase, iar cei care aparțin secțiunii "**private**" (atât date cât şi funcții) nu pot fi accesați decât de către metodele clasei respective. Utilizatorul clasei nu va avea acces la ei decât prin intermediul metodelor declarate în secțiunea **public** (metodelor publice).

Definirea metodelor care aparțin unei clase se face prefixând numele metodei cu numele clasei, urmat de "::". Simbolul "::" se numeşte "**scope acces operator**" (**operator de rezoluție** sau **operator de acces**) şi este utilizat în operații 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 clasa sau nimic, în cel de-al doilea caz prefixarea variabilei folosindu-se pentru accesarea unei variabile globale (vezi laboratorul 1).

În lipsa numelui clasei în fața funcției membru nu s-ar putea face distincția între metode care poartă nume identice şi aparțin 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 aparțin clasei respective.

Partea practică a laboratorului:

Aplicația 1

Să se scrie o aplicație care implementează o stivă cu ajutorul unui tablou. Se vor implementa funcțiile de adăugare în stivă, scoatere din stivă, afişare a stivei (toate elementele).

Aplicația 2

Să se realizeze un program care implementează un meniu cu următoarele opțiuni:

Meniu:
- O limonada indulcita
- O limonada neindulcita
- Afisare total incasari
- Ieşire

Clasa Lemon
- private:
  - total numar lamai (se foloseste cate una la fiecare limonada)

total numar cuburi de zahar (cate 2 la fiecare limonada indulcita)
suma incasari (se incrementeaza cu pretul corespunzator)

public:

initializare (se specifica numarul de lingurite de zahar si de lamai disponibile)
bea o limonada indulcita (verificare: mai este zahar, mai este lamaie ?)
bea o limonada neindulcita (verificare: mai este lamaie?)
afisare total incasari

Daca acele condiții nu se verifică, se afişează mesajele corespunzătoare.

Întrebări:

1. Ce este încapsularea?
2. Ce este o clasa?
3. Ce este un obiect?
4. Ce este o funcție membra?
5. Care este diferența între clase şi structuri?
6. Pentru ce este utilizat "scope acces operator"?
7. Variabilele membru private pot fi accesate şi în afara clasei respective?

Laborator 3 POO:

Pointeri la metode. Funcții inline. Membri statici

Scopul laboratorului: familiarizarea cu noțiunile de **pointer** (în special cu noțiunea de pointer la metode), funcții **inline** şi **membrii statici**.

Beneficii:

- utilizarea pointerilor la metode aduce o flexibilitate sporită în conceperea programelor;
- utilizarea funcțiilor **inline** creşte viteza de execuție a programelor;
- utilizarea membrilor statici ajută la reducerea numărului de variabile globale;

1. Pointeri la metode

Deşi o funcție nu este o variabilă, ea are o localizare în memorie, care poate fi atribuită unui pointer. Adresa funcției respective este punctul de intrare în funcție; ea poate fi obținută utilizându-se numele funcției, fără nici un argument (similar cu obținerea adresei unei matrici).

Un **pointer la metodă** desemnează adresa unei funcții membru "m". Metodele unei clase au implicit un parametru de tip "pointer la obiect", care se transmite ascuns. Din acest motiv, parametrul în cauză nu apare in lista de parametri a funcției desemnate de pointer.

Exemplu:

```
class clasa
{
     int contor;
 public:
     void init (int nr = 0)
     {
          contor = nr;
     }
     int increment (void)
     {
          return contor++;
     }
};

/*
   tipul "pointerLaMetoda" este un pointer la o metoda a clasei
   "clasa", metoda care nu are parametri si care returneaza "int"
*/

typedef int (clasa::*pointerLaMetoda)(void);

void main(void)
{
     clasa c1, *pc1 = &c1;
```

```
        pointerLaMetoda pM = &(clasa :: increment);
        c1.init (1);
        pc1->init(2);
        int i = (c1.*pM)();
        i = (pc1->*pM)();
    }
```

După cum se observă în acest exemplu, pointerul poate "pointa" către orice metodă a clasei "**clasa**".

2. Funcții inline

Funcțiile "**inline**" sunt eficiente în cazurile în care transmiterea parametrilor prin stivă (operație lentă) este mai costisitoare (ca şi timp de execuție) decât efectuarea operațiilor din corpul funcției.

Exemplu:

```
class coordonate_3D
{
      int X,Y,Z;

// ATENTIE:
// prin definirea metodei in interiorul declararii clasei,
// functia membru devine implicit "inline" !!

void translateaza(int TX, int TY, int TZ)
{
      X+=TX;
      Y+=TY;
      Z+=TZ;
}
void tipareste(void); // declararea unei metode care nu este
                      // implicit ”inline”
};

// Prefixarea definitiei metodei cu cuvintul cheie "inline" este
// echivalenta cu definirea functiei membru in cadrul declaratiei
// clasei

inline void coordonate_3D::tipareste(void)
{
      cout << "\n\tX=" << X << "\tY=" << Y << "\tZ=" << Z << "\n";
}
```

Deoarece procedura "translatează" este de tip *inline*, ea nu este apelată, ci expandată atunci când este apelată.

Definirea unei funcții în cadrul unei clase se mai numeşte şi *declarare inline implicită*. Metoda "*tipareste*" este declarată *explicit*, ea fiind doar declarată în cadrul clasei, iar în locul unde este definită este prefixată de cuvântul cheie "*inline*".

Metodele care nu sunt membru al unei clase nu pot fi declarate **inline** decât explicit. Este interzisă folosirea în cadrul funcțiilor **inline** a structurilor repetitive ("for", "while", "do while"), şi a funcțiilor recursive.

3. Membri statici

Principala utilizare a variabilelor membru statice constă în eliminarea în cât mai mare măsură a variabilelor globale utilizate într-un program.

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 instanțierilor unei clase (obiectelor).**

Variabilele membru statice pot fi prefixate doar de numele clasei, urmat de operatorul de de rezoluție "::". Apelul metodelor statice se face exact ca şi accesarea variabilelor membru statice. Deoarece metodele statice nu sunt apelate de un obiect anume, nu li se transmite pointer-ul ascuns "**this**".

Dacă într-o metodă statică se folosesc variabile membru nestatice, e nevoie să se furnizeze un parametru explicit de genul *obiect*, *pointer la obiect* sau *referință la obiect*.

Toți membrii statici sunt doar declarați în cadrul clasei, ei urmând a fi obligatoriu inițializați.

Exemplu

```
class exemplu
{
      int i;
public:
      static int contor;               // variabila membru statica
      static inc (void) {i++;}         // metoda statica
      void inc_contor (void) {contor++;}
      void init (void){i = 0;}
      static void functie (exemplu *); // metoda statica
} ob1, ob2, ob3;

int exemplu::contor = 0;    // initializarea variabilei statice

void exemplu::functie(exemplu *ptrEx)
{
      // i += 76       // eroare - nu se cunoaste obiectul de care
                       // apartine i
      ptrEx -> i++;    // corect
```

```
        contor ++;      // corect
    }

    void main(void)
    {
        ob1.init();
        ob2.init();
        ob3.init();

        ob1.inc();
        ob2.inc();
        ob3.inc();

        ob1.functie(&ob1);          // corect
        exemplu :: functie(&ob2);   // corect

        // functie();    // incorect - in afara cazului in care
                         // exista o metoda ne-membru cu acest nume

        ob1.inc_contor();
        ob2.inc_contor();
        ob3.inc_contor();

        exemplu :: contor+=6;
    }
```

Partea practică a laboratorului:

Aplicația 1

Să se scrie un program care implementează o stivă de numere întregi, utilizând o clasă. Membrii variabili ai clasei indică vârful stivei şi tabloul de întregi în care se reţin elementele stivei. Funcţiile membru ale clasei vor fi:

- o funcție pentru inițializarea pointerului în stivă;
- o funcție pentru introducerea unei noi valori în stivă;
- o funcție pentru scoaterea unei valori din stivă;
- o funcție pentru afişarea întregii stive;

Se va folosi un meniu de selecție care va avea opțiuni cerințele programului.

Timp de rezolvare: 50 min.

Întrebări:

1. Ce sunt pointerii?
2. Ce sunt pointerii la metode?
3. Cum se declară pointerii la metode?
4. Ce sunt funcțiile inline?

5. Cum se declară funcțiile inline?
6. Care este avantajul folosirii funcțiilor inline?
7. Care sunt restricțiile impuse funcțiilor inline?
8. Ce sunt membri statici?
9. Membri statici aparțin unei clase?
10. Cum se declară u membru static?

Teme de casă:

Aplicația 1

Implementați o *coadă* de numere întregi (structură de date de tip **FIFO** - first-in, first-out), utilizându-se o clasă. Membrii variabili ai clasei indică vârful stivei şi tabloul de întregi în care se reține elementele stivei. Funcțiile membru ale clasei vor fi:

- o funcție pentru inițializarea pointerului în stivă;
- o funcție pentru introducerea unei noi valori în stivă;
- o funcție pentru scoaterea unei valori din stivă;

Se va folosi un meniu de selecție care va avea ca opțiuni cerințele programului.

Aplicația 2

Să se scrie un program care implementează o listă simplu înlănțuită utilizând o clasă. Membrii variabili ai clasei indică următorul element şi conținutul unui element al listei (un întreg). Funcțiile membru ale clasei vor fi:

- o funcție pentru inițializarea listei;
- o funcție pentru introducerea unei noi valori în listă;
- o funcție pentru scoaterea unei valori din listă;

Se va folosi un meniu de selecție care va avea ca opțiuni cerințele programului.

Laborator 4 POO:

Constructori şi destructori

Scopul laboratorului: prezentarea mecanismelor de iniţializare şi de "distrugere" a unor proprietăţi ale obiectelor, folosind constructorii şi a destructorii.

Beneficii: utilizarea constructorilor şi a destructorilor oferă programatorului instrumentele necesare iniţializării unor proprietăţi ale obiectelor dintr-o clasă, precum şi a dezalocării (distrugerii) acestora, în mod automat, fără a fi nevoie de aplearea unor funcţii separate pentru aceasta. Folosirea contructorilor şi a destructorilor permite scrierea programelor într-un mod mai compact şi mai uşor de înteles.

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, 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;
- 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 poate defini în două moduri:

a. definind un constructor fără nici un parametru;
b. 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 conține nici o instrucțiune.

O clasă poate conține, 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);
```

2. Destructorii

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

Exemplu:

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

Destuctorii au următoarele **caracteristici speciale**:

- sunt apelați implicit în două situații:
  1. când se realizează eliberarea memoriei alocate dinamic pentru memorarea unor obiecte, folosind operatorul "**delete**" (a se vedea linia 10 din programul de mai sus);
  2. la părăsirea domeniului de existență al unei variabile (vezi linia 17, variabila pb). Dacă în al doilea caz este vorba de variabile globale sau definite în "main", distrugerea lor se face după ultima instrucțiune din "main", dar înainte de încheierea execuției programului.

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 menționată 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
}                                                    // 17
```

Întrebări:

1. Ce sunt constructorii?
2. Cum se declară constructorii?
3. Ce tip de dată poate returna un constructor?
4. Constructorii pot avea parametri?
5. Cum se apelează constructorii?
6. Ce sunt constructorii de copiere?
7. O clasa poate avea mai mulți constructori? Dacă da, atunci cum ştie compilatorul să facă diferențierea între aceştia?
8. Ce este un destructor?
9. Câți destructori poate avea o clasă?
10. Cum se poate apela un destructor?

Partea practică a laboratorului:

Aplicația 1

Să se realizeze o listă simplu înlanțuită de şiruri de caractere (albume). Prototipul clasei este următorul:

```
class node
{
      static node *head;          //pointer la lista
      node *next;                 //pointer catre urmatorul
                                  //element
      char *interpret;            //numele unui obiect din lista
      char *melodie;              //numele unei melodii din
                                  //lista
 public:
      node (char * = NULL);       //declararea constructorului
      void display_all ();        //afiseaza nodurile listei
      void citireAlbum ();        //citirea informatiilor despre
                                  //album

};

node *node :: head = NULL;         //initializare cap lista
node :: node(char *ptr, char *ptr1)   // constructorul clasei
{
      …
}

void node :: display_all ()
{
      …
}

void node:: citireAlbum ()
{
      …
}
```

```
void main()
{
      …
}
```

Se va folosi un meniu, care să implementeze cerințele programului: citire album, creare listă şi afişarea întregii liste (a întregului album).

Teme de casă:

Aplicația 1

Să se dezvolte aplicația de la laborator astfel încât să se poate modifica o valoare (a albumului), ordona crescător după melodie şi şterge un nod din listă (un album).

Aplicația 2

Aceleaşi cerințe, pentru o listă dublu înlănțuită.

Laborator 5 POO:

Functii si clase prietene

Scopul laboratorului: prezentarea mecanismelor de acces la datele membre ale unei clase prin intermediul funcțiilor *friend* (prietene) şi a claselor *friend*.

Beneficii: utilizarea funcțiilor *friend* şi a claselor *friend* oferă programatorului mai multă flexibilitate în dezvoltarea unui program, dându-i posibilitatea să acceseze variabilele membru ale unei clase. Astfel, dacă într-un program este necesară accesarea variabilelor membru ale unei clase se poate utiliza o funcție *friend* în locul unei funcții membre.

1. Funcții friend

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

Exemplu:

```
class exemplu {
     int a;
     int f (void);
     friend int f1(exemplu &);
     public:
          friend int M::f2(exemplu &, int);
};

int f1(exemplu &ex)
{
     return ex.f ();
}

int M :: f2(exemplu &ex, int j = 0)
{
     if (ex.a > 7) return j++;
     else return j--;
}
```

După cum se observă, nu contează dacă o funcție este declarată *friend* în cadrul secțiunii *private* sau *public* a unei clase.

2. Clase friend

Dacă se dorește ca toți 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”.

Exemplu:

```
      class M {
                // ...
      };

     class B
     {
                // ...
          friend class M;
     };
```

Relația de *friend* **nu este tranzitivă**, adică dacă clasa A este *friend* clasei B, iar clasa B este *friend* clasei C, aceasta nu implică faptul că, clasa A este implicit *friend* clasei C.

Funcțiilor *friend* nu li se transmite parametrul ascuns **this**. Această carență este suplinită prin transmiterea unor parametrii obişnuiți de tip pointer, obiect sau referință la obiect.

Exemplu:

```
class rational;    // declarare incompleta

class complex
{
     double p_reala, p_imaginara;
     friend complex& ponderare (complex&, rational&);
     public:
          complex (double r, double i) : p_reala (r), p_imaginara (i)
          double get_real (void) {return p_reala;}
          double get_imaginar (void) {return p_imaginara;}
};

class rational
{
          int numarator, numitor;
          double val;
     public:
          friend complex& ponderare (complex& ,rational&);
          rational (int n1, int n2) : numarator (n1)
          {
               numitor = n2!=0 ? n2 : 1;
               val = ((double)numarator)/numitor;
          }
          double get_valoare(void)  { return val; }
     };
// fiind "friend", functia ponderare are acces la membrii privati ai
// claselor "complex" si "rational"

complex &ponderare(complex& c,rational& r)
{
     complex *t = new complex (c.p_reala *r.val, c.p_imaginara
          *r.val);
          return *t;
}
```

```
// nefiind "friend", "ponderare_ineficienta" nu are acces la membrii
// privati ai claselor "complex" si "rational"

complex &ponderare_ineficienta (complex &c, rational &r)
{
     complex *t = new complex (c.get_real()*r.get_valoare(),
     c.get_imaginar()*r.get_valoare());
     return *t;
}

void main(void)
{
     complex a(2,4),b(6,9);
     rational d(1,2),e(1,3);

     a = ponderare(a,d);
     b = ponderare(b,e);

     a = ponderare_ineficienta(a,d);
     b = ponderare_ineficienta(b,e);
}
```

Partea practică a laboratorului

Aplicația 1

Folosind funcții şi clase *friend*, să se realizeze un program care implementează gestiunea unui magazin de CD-uri, folosind o clasă CD. Fiecare obiect de tip CD are câmpurile: interpret, titlu şi melodii_componente. Melodiile trebuie memorate sub forma unei liste simplu înlănțuite. Programul trebuie să permită 2 opțiuni: introducerea informațiilor pentru CD-uri şi afişarea tuturor CD-urilor în ordinea introducerii (titlu, interpret, melodiile care le cuprind). Cele 2 funcții care realizează aceste operații vor fi funcții friend ale clasei CD şi nu metode ale acesteia.

Timp de rezolvare: 50 min.

Întrebări:

1. Ce sunt funcțiile prietene?
2. Cum se declară funcțiile prietene?
3. Ce sunt clasele prietene?
4. Cum se declară clasele prietene?
5. În ce secțiune a clasei se declară funcțiile prietene?

Teme de casă

Aplicația 1

Să se dezvolte aplicația de la laborator astfel încât să se poate modifica orice informație despre CD-uri, să permită ordonarea în mod crescător după melodii precum şi ştergerea unui

nod din lista de CD-uri. De asemenea se vor folosi funcții prietene pentru accesul la membri privați ai clasei

Aplicația 2

Aceleaşi cerințe, dar pentru o listă dublu înlănțuită.

Laborator 6 POO:

Moştenirea (derivarea) claselor

Scopul laboratorului: prezentarea moştenirii claselor, precum şi a utilizării constructorilor claselor derivate.

Beneficiul: utilizarea moştenirii permite realizarea de ierarhi de clase, ceea ce duce la o mai bună modularizare a programelor.

1. Principiul moştenirii (derivării) claselor

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".

În declarația clasei derivate nu mai apar informațiile care sunt moştenite, ele fiind automat luate în considerare de către compilator. Nu mai trebuie rescrise funcțiile 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 funcționalitate.

Exemplu:

```
//clasa "hard_disk" este derivata din clasa "floppy_disk"

enum stare_operatie   {REUSIT, ESEC };
enum stare_protectie_la_scriere  { PROTEJAT, NEPROTEJAT }

class floppy_disk
{
      protected:       // cuvantul cheie "protected" permite
                       // declararea unor membrii nepublici, care
                       // sa poata fi accesati de catre
                       // eventualele clase derivate din
                       // "floppy_disk"
            stare_protectie_la_scriere indicator_protectie;
            int capacitate, nr_sectoare;
      public:
            stare_operatie formatare ();
            stare_operatie citeste_pista (int drive, int
            sector_de_start, int numar_sectoare, void *buffer);
            stare_operatie scrie_pista (int drive, int
            sector_de_start, int numar_sectoare, void *buffer);
            stare_operatie protejeaza_la_scriere(void);
};

class hard_disk : public floppy_disk
{
      int numar_partitii;
      public:
            stare_operatie parcheaza_disc ();
};
```

```
stare_operatie hard_disk :: parcheaza_disc ()
{
      // CORECT: accesarea unui membru de tip "protected"
      indicator_protectie = PROTEJAT;

      //...
      return REUSIT;
}

void functie ()
{
      hard_disk hd1;
      hd1.formatare();                        //CORECT
      hd1.indicator_protectie = NEPROTEJAT; // EROARE: incercare
                              // de accesare a unui membru protejat
}
```

Prin enunțul:

```
      class hard_disk : public floppy_disk
```

se indică compilatorului următoarele informații:

- se crează o clasă numită "*hard_disk*", care este derivată (moştenită) din clasa "*floppy_disk*";
- toți membrii de tip "*public*" ai clasei "*floppy_disk*" vor fi moşteniți (şi deci vor putea fi folosiți) ca "*public*" de către clasa "hard_disk";
- toți membrii de tip "*protected*" ai unui obiect de tip "*floppy_disk*" vor putea fi utilizați ca fiind "*protected*" în cadrul clasei "*hard_disk*";

2. Declararea unei clase derivate

Pentru a declara o clasă derivată dintr-o clasă de bază se foloseşte următoarea sintaxă:

```
specificator_clasa   nume_clasa_derivata:   [modificator_acces_1]
nume_clasa_baza_1  [ , [ modificator_acces_2 ] nume_clasa_baza_2
] [ , ... ] ]
      {
      [ [private: ]
            lista_membri_1
      ]
      [ [protected: ]
            lista_membri_2
      ]
      [ [public : ]
            lista_membri_3
      ]
      };

      [lista_obiecte];
```

În funcție de tipul lui ***specificator_clasa*** ("*class*" sau "*struct*"), ***modificator_acces_1*** va lua valoarea implicită *private* sau *public*. Rolul acestui ***modificator_acces_1*** este ca, împreună cu specificatorii "*public*", "*private*" sau

"*protected*" întâlniți în cadrul declarării clasei de bază, să stabilească drepturile de accesare a membrilor moşteniți de către o clasă derivată.

Tabelul următor sintetizează drepturile de accces a membrilor unei clase derivate în funcție de drepturile de accesare a membrilor clasei de bază şi valoarea lui ***modificator_acces_1***.

Drept de acces în clasa de bază	Modificator de acces	Drept de acces în clasa derivată
public	public	public
private	public	inaccesibil
Protected	public	protected
Public	private	private
private	private	inaccesibil
PROTECTED	PRIVATE	PRIVATE

Cuvântul cheie "*protected*" nu poate fi folosit ca modificator de acces în cadrul unei relații de moştenire. Rolul său se limitează la a permite accesarea din cadrul unei clase derivate a membrilor nepublici ai clasei de bază. Funcțiile membre ale clasei derivate **nu** au acces la membrii privați ai clasei de bază.

O altă observație este aceea că orice *friend* (funcție sau clasă) a unei clase derivate are exact aceleaşi drepturi şi posibilități de a accesa membrii clasei de bază ca oricare alt membru al clasei derivate.

Exemplu:

```
class Angajat
{
 private:
      char nume[30];
      // alte caracteristici: data de naştere, adresa etc.
public:
      Angajat ();
      Angajat (const char *);
      char *getname () const;
      double calcul_salariu (void);

};

class Muncitor: public Angajat
{
private:
      double plata;
      double ore;
public:
      Angajat_2 (const char *nm);
      void calcul_plata (double plata);
      void nr_ore (double ore);
      double calcul_salariu ();
};

class Vânzător: public Muncitor
{
private:
```

```
        double comision;
        double vânzări;
    public:
        Vânzător (const char *nm);
        void setare_comision (double comis);
        void setare_vânzări (double vanz);
        double calcul_plata ();
    };

    class Manager: public Angajat
    {
    private:
                double salariu_săpt;
    public:
                Manager (const char *nm);
                void setare_salariu ( double salary);
                double calcul_plata ();
    };
```

Cuvântul cheie *const* folosit dupa lista de parametri ai unei funcții declară funcția membru ca şi funcție "*read-only*" – funcția respectivă nu modifică obiectul pentru care este apelată.

Funcția *calcul_plata ()* poate fi scrisă pentru diferitele tipuri de angajați :

```
    double Angajat_plata_ora :: calcul_plata () const
    {
        return plata * ore;
    }

    double Vânzător :: calcul_plata () const
    {
        return Angajat_plata_ora::calcul_plata() + commision
            * vânzări;
    }
```

Această tehnică este folosită de obicei atunci când se redefineşte o funcție membru într-o clasă derivată. Versiunea din clasa derivată apelează versiunea din clasa de bază şi apoi efectuează celelalte operații necesare.

3. Constructorii claselor derivate

O instanțiere a unei clase derivate conține toti membrii clasei de bază şi toți aceştia trebuie inițializați. Constructorul clasei de bază trebuie apelat de constructorul clasei derivate.

```
    // constructorul clasei Angajat_plata_ora
    Angajat_plata_ora::Angajat_plata_ora(const char *nm):Angajat(nm)
    {
        plata = 0.0;
        ore = 0.0;
    }
```

```
// constructorul clasei Vânzător
Vânzător::Vânzător(const char *nm) : Angajat_plata_ora(nm)
{
      commision = 0.0;
      vânzări = 0.0;
}

// constructorul clasei Manager
Manager :: Manager (const char *nm) : Vânzător (nm)
{
      weeklySalary = 0.0;
}
```

Când se declară un obiect dintr-o clasă derivată, compilatorul execută întâi constructorul clasei de bază, apoi constructorul clasei derivate (dacă clasa derivată conține obiecte membru, constructorii acestora sunt executați după constructorul clasei de bază, dar înaintea constructorului clasei derivate).

4. Conversii între clasa de bază şi clasa derivată

Limbajul **C++** permite conversia implicită a unei instanțieri a clasei derivate într-o instanțiere a clasei de baza.

De exemplu:

```
Muncitor  mn;
Vânzator vanz ("Popescu Ion");
mn = vanz;  // conversie derivat => bază
```

De asemenea, se poate converti un pointer la un obiect din clasa derivată într-un pointer la un obiect din clasa de bază.

Conversia **derivat*** => **baza*** se poate face:

- **implicit** - dacă pointer-ul *derivat* moşteneşte pointer-ul *bază* prin specificatorul *public*;
- **explicit**: dacă pointer-ul *derivat* moşteneşte pointer-ul *bază* prin specificatorul *private;*

Conversia **baza*** -> **derivat*** nu poate fi făcută decât **explicit**, folosind operatorul *cast*.

Când se accesează un obiect printr-un pointer, tipul pointer-ului determină care funcții membre pot fi apelate. Dacă se accesează un obiect din clasa derivată printr-un pointer la clasa de bază, pot fi apelate doar funcțiile definite în clasa de bază. Dacă se apelează o funcție membru care este definită atât în clasa de bază cât şi în cea derivată, funcția care este apelată depinde de tipul pointerului:

```
double brut, total;
brut = munc -> calcul_salariu ();
// se apelează Muncitor :: calcul_salariu ()
total = vanz -> calcul_salariu ();
// se apelează Vanzator :: calcul_salariu ()
```

Conversia inversă trebuie făcută explicit:

```
      Muncitor *munc = &munc_1;
      Vanzator *vanz;
      vanz = (Vanzator *) munc;
```

Această conversie nu este recomandată, deoarece nu se poate şti cu certitudine către ce tip de obiect pointează pointer-ul la clasa de bază.

O problemă care poate apare este, de exemplu, calcularea salariului fiecărui angajat din listă. După cum s-a menționat anterior, funcția apelată depinde de tipul pointerului, ca urmare este nesatisfăcatoare apelarea funcția *calcul_salariu ()* folosind doar pointeri la clasa *Angajat*. Este necesară apelarea fiecarei versiuni a funcției *calcul_salariu ()* folosind pointeri generici, lucru posibil folosind funcțiile virtuale (vor fi prezentate în laboratorul 7).

Partea practică a laboratorului

Aplicația 1

Să se implementeze o aplicație care ține evidența studenților pe secțiuni: în campus respectiv în oraş. O posibilă structură este cea prezentată mai jos:

```
class student
{
      char *name, *prenume;
      int year, varsta;
public:
      void display();             // afisarea inform. (nume si an
                                  // de studiu) pentru un student
      student (char *, int);      // constructorul
      ~student();                 // destructorul
};

class on_campus : public student
{
      char *dorm,
      *room;
      public:
            void a_disp();   // afiseaza informații. (nume, an de
                             // studiu, camin, camera) pentru un
                             // student
            on_campus(char *, int, char *, char *);// constructor
            ~on_campus ();   // destructor
};

class off_campus : public student
{     // în mod analog cu clasa precedentă:
      // campuri care indica adresa off_campus (strada, oras,
      // nr.) a unui student
      ... // constructorul clasei
      ... // destructorul
      // functie de afisare a inform. (nume, an de studiu,
      // adresa) a unui student
};
```

```
void main ()
{
      // se declara cite o instanta a fiecarei clase
      // sa se afiseze informatiile referitoare la fiecare
      // persoana declarata
}
```

Întrebări:

1. Ce este moştenirea?
2. Cum se realizează moştenirea?
3. Care sunt drepturile de acces la membrii unei clase de bază în funcție de tipul moştenirii?
4. Ce sunt constructorii?
5. Cum se apelează constructorii clasei de bază?
6. Care constructor se apelează primul, al clasei de bază sau al clasei derivate?
7. Cum se fac conversiile între clasa de bază şi clasa derivată?
8. Când este folosită conversia implicită?
9. Când este necesară conversia explicită?
10. Cum se poate apela un destructor?

Teme de casă:

Aplicația 1

Să se dezvolte aplicația de la laborator folosind pentru implementare liste de obiecte.

Aplicația 2

Aceleaşi cerințe pentru o listă dublu înlănțuită.

Laborator 7 POO:

Metode virtuale. Utilizarea listelor eterogene

Scopul Lucrării: familiarizarea cu noțiunile de metode virtuale precum şi cu modul de utilizare a listelor eterogene.

Beneficiul:
- utilizarea metodelor virtuale va permite redefinirea / reutilizarea metodelor
- utilizarea metodelor virtuale ne va permite să creăm şi să utiliză liste eterogene

Scurtă prezentare: în cadrul acestui laborator se vor studia:
- metodele virtuale,
- listele eterogene,

Prezentarea laboratorului:

Prin definiție, un tablou omogen este un tablou ce conține elemente de acelaşi tip. Un pointer la o clasa "B" poate păstra adresa oricărei instanțieri a vreunei clase derivate din "B". Deci, având un şir de pointeri la obiecte de tip "B" , înseamnă că, de fapt, putem lucra şi cu tablouri neomogene. Astfel de tablouri neomogene se vor numi **ETEROGENE**. Una dintre caracteristicile limbajului C++ constă tocmai în faptul că mecanismul funcțiilor virtuale permite tratarea uniformă a tuturor elementelor unui masiv de *date eterogene*. Acest lucru este posibil datorită unor facilități ce țin de asocieri făcute doar în momentul execuției programului, nu în momentul compilării.

Fie o clasă "B" care posedă o metodă publică "M". Din clasa "B" se derivă mai multe clase "D1", "D2", "D3",...,"Dn". Dacă aceste clase derivate redefinesc metoda "M" se pune problema modului în care compilatorul este capabil să identifice corect fiecare dintre aceste metode.

În mod obişnuit identificarea funcției membru în cauză se poate face prin una din următoarele metode:

1. prin unele diferențe de semnatură (în cazul în care metoda a fost redeclarată)
2. prin prezența unui *scope resolution operator* (o exprimare de genul ***int a=B::M()*** este univocă)
3. cu ajutorul obiectului căruia i se aplică metoda. O secvență de genul:

```
D1 d;
d.M();
```

nu lasă nici un dubiu asupra metodei în cauză.

În aceste cazuri, decizia este deosebit de simpla şi poate fi luată chiar în faza de compilare.
În cazul prelucrării de liste eterogene situația este mai complicată, fiind implicată o rezolvare mai târzie a asociațiilor între numele funcției apelate şi funcția de apelat. Fie

şirul eterogen de pointeri la obiecte de tipul "B", "D1", "D2" etc. Se presupune, de exemplu, că într-o procedură se încearcă apelarea metodei "M" pentru fiecare obiect pointat de catre un element al şirului. Metoda de apelat nu va fi cunoscută în momentul compilării deoarece nu este posibil să se stabilească corect despre care din funcţii este vorba ("M"-ul din "B" sau cel al unei clase derivate din "B"). Imposibilitatea identificării metodei apare deoarece informaţiile privind tipul obiectului la care pointează un element al şirului nu vor fi disponibile decât în momentul executării programului.

În continuare se analizează un exemplu clasic de tratare a unei liste neomogene, care apoi este reluat utilizând funcţii virtuale.

Exemplul :
Se construieşte un şir de 4 elemente ce conţine pointeri atât la clasa "BAZA" cât şi la "DERIVAT_1" şi "DERIVAT_2".

```
#include <iostream.h>
typedef enum {_BAZA_, _DERIVAT_1, _DERIVAT_2} TIP;

class BAZA{
  protected:
        int valoare;
  public:
        void set_valoare(int a) (valoare = a}
        void tipareste_valoare(void)
        {
                cout<<"Element BAZA cu VALOARE = "<<valoare<<"\n";
        }
};

class DERIVAT_1 : public BAZA{
  public:
        void tipareste_valoare(void)
        {
            cout<<"Element DERIVAT_1 cu VALOARE = "<<valoare<<"\n";
        }
};

class DERIVAT_2 : BAZA {
  public:
        BAZA::set_val;  //metoda "set_val" va fi fortata la "public"
        void tipareste_valoare(void)
        {
             cout<<"Element DERIVAT_2 cu VALOARE = "<<valoare<<"\n";
        }
};
```

```
class LISTA_ETEROGENA{
        BAZA *pB;
        TIP  t;
   public:
        void set(BAZA *p, TIP tp=_BAZA_)
        {
          pB = p; t = tp;
        }
        void tipareste_valoare(void)
        {   if(t==_BAZA_)
                pB->tipareste_valoare();
            else
                if(t==_DERIVAT_1)
                        ((DERIVAT_1 *) pB->tipareste_valoare();
                else
                        ((DERIVAT_2 *) pB->tipareste_valoare();
        }
};

void main()
{
        BAZA a[2];
        DERIVAT_1 b1;
        DERIVAT_2 b2;
        LISTA_ETEROGENA p[4];

        a[0].set_val(1);
        a[1].set_val(2);
        b1.set_val(3);
        b2.set_val(4);

        p[0].set(&a[0]);
        p[1].set(&b1,_DERIVAT_1); //se face o conversie implicita
                                        //"DERIVAT_1"->"BAZA*"
        p[2].set((BAZA *)&b2, _DERIVAT_2); //se face o conversie explicita
                                        //"DERIVAT_2"->"BAZA *"
        p[3].set(&a[1]);
        for(int i=0; i<4; i++)
                p[i].tipareste_valoare();
}
```

În urma execuției programului, pe ecran se vor afișa următoarele mesaje:

Element BAZA cu VALOARE = 1
Element DERIVAT_1 cu VALOARE = 3
Element DERIVAT_2 cu VALOARE = 4
Element BAZA cu VALOARE = 2

Pentru a reuşi o tratare uniformă a celor 3 tipuri de obiecte a fost necesară crearea unei a 4-a clase, numită LISTA_ETEROGENA. Aceasta va păstra atât pointer-ul la obiectul respectiv, cât şi o informaţie suplimentară, referitoare la tipul obiectului referit de pointer. Tipărirea informaţiilor semnificative ale fiecarui obiect se va face în metoda ***LISTA_ETEROGENA::tipareste_valoarea***. În această funcţie membru sunt necesare o serie de teste pentru apelul metodei corespunzând fiecarui tip de obiect.

Exemplul:
Se prezintă o modalitate mult mai simplă şi elegantă de a trata omogen un masiv de date eterogene.

```
#include <iostream.h>

class BAZA{
  protected:
        int valoare;
  public:
        void set_val(int a) { valoare = a;}
        virtual void tipareste_valoare(void)
        {
            cout<<"Element BAZA cu VALOARE = "<<valoare<<"\n";
        }
};

class DERIVAT_1 : public BAZA {
  public:
        void tipareste_valoare(void)
        {
            cout<<"Element DERIVAT_1 cu VALOARE = "<<valoare<<"\n";
        }
};

class DERIVAT_2 : BAZA {
  public:
        BAZA::set_val;
        void tipareste_valoare(void)
        {
             cout<<"Element DERIVAT_2 cu VALOARE = "<<valoare<<"\n";
        }
};

class LISTA_ETEROGENA {
        BAZA *pB;
  public:
        void set(BAZA *p) {pB = p;}
```

```
        void tipareste_valoare(void)
        {
            pB->tipareste_valoare();
        }
};

void main()
{
        BAZA  a[2];
        DERIVAT_1 b1;
        DERIVAT_2 b2;
        LISTA_ETEROGENA p[4];

        a[0].set_val(1);
        a[1].set_val(2);
        b1.set_val(3);
        b2.set_val(4);
        p[0].set(&a[0]);
        p[1].set(&b1);
        p[2].set((BAZA *) &b2);
        p[3].set(&a[1]);

        for(int i+0; i<4; i++)
                p[1].tipareste_valoare();
}
```

În acest caz clasa **LISTA_ETEROGENA** are o metodă ***tipareste_valoarea*** mult simplificată. În noua funcție membru nu mai este necesară nici testarea tipului de pointer şi nici conversia pointerului memorat la tipul original. Acestea se realizează prin prefixarea metodei ***BAZA::tipareste_valoarea*** cu cuvântul cheie **virtual**. În urma întâlnirii cuvântului **virtual** compilatorul va lua automat deciziile necesare.

Prefixarea unei metode "F" a clasei "B" cu cuvântul cheie **virtual** are următorul efect: toate clasele ce o moştenesc pe "B" şi nu redefinesc funcția "F" o vor moşteni întocmai. Dacă o clasă "D" derivată din "B" va redefini metoda "F" atunci compilatorul are sarcina de a asocia un set de informații suplimentare, cu ajutorul cărora se va putea decide (în momentul execuției) care metodă "F" va fi apelată.

Observații:
1. Deoarece funcțiile virtuale necesită memorarea unor informații suplimentare, instanțierile claselor care au metode virtuale vor ocupa în memorie mai mult loc decât ar fi necesar în cazul în care nu ar exista decât metode ne-virtuale.
Nici FUNCȚIILE NE_MEMBRU şi nici METODELE STATICE NU POT FI DECLARATE VIRTUALE.

2. Pentru ca mecanismul metodelor virtuale să poata funcționa, metoda în cauză NU POATE FI REDECLARATĂ în cadrul claselor derivate ca având aceiaşi parametrii şi returnând un alt tip de dată. În schimb, este permisă redeclararea cu un alt set de argumente, dar cu acelaşi tip de dată returnat. dar în această situație, noua funcție nu va mai putea fi moştenită mai departe ca fiind metodă virtuală.

3. Evitarea mecanismului de apel uzual pentru funcțiile virtuale se face foarte uşor prin prefixarea numelui funcției cu numele clasei aparținătoare urmat de "**::**"

Exemplu:

```
void DERIVAT::f1(void)
{
    BAZA::tipareste_valoare();     //apelul metodei din clasa BAZA
    tipareste_valoare();           //apelul metodei din clasa DERIVAT
}
```

4. Constructorii nu pot fi declarați virtual, însa destructorii acceptă o astfel de prefixare.

5. O funcție virtuală "F", redefinită într-o clasă derivată "D" va fi văzută în noua sa formă de către toate clasele derivate din "D". Această redefinire nu afectează cu nimic faptul că ea este virtuală, metoda rămânând în continuare virtuala şi pentru eventualele clase derivate din "D".

6. Când se pune problema de a decide dacă o metodă sa fie sau nu virtuală, programatorul va lua în considerare următoarele aspecte:

- în cazul în care contează doar performanțele unui program şi se exclude intenția de a-l mai dezvolta în continuare, se va opta pentru metode ne-virtuale.
- în cazul programelor ce urmează să suporte dezvoltări ulterioare, situația este cu totul alta. Vor fi declarate virtuale toate acele metode ale clasei "CLASA" care se consideră că ar putea fi redefinite în cadrul unor clase derivate din "CLASA" şi, în plus, modificarea va trebui să fie sesizabilă la "nivelul" lui "CLASA". Toate celelalte metode pot rămâne ne_virtuale.

Partea practică a laboratorului:

Aplicația 1:
Dându-se clasa de bază **Lista**, să se construiască două clase derivate **Stiva** şi **Coada**, folosind funcțiile virtuale ***store()*** (adaugă un element în stivă sau coadă) şi ***retrieve()*** (şterge un element din stivă sau coadă).

Întrebări:

1. Ce sunt metodele virtuale?
2. Când se utilizează funcțiile virtuale?
3. Ce înseamnă cuvântul eterogen?

4. Cum se va face diferențierea între metodele(redefinite) claselor derivate?
5. Pot fi declarate ca fiind virtuale funcțiile nemembre?
6. Pot fi declarate ca fiind virtuale metodele statice ale unei clase?
7. Pot fi declarați constructorii ca fiind virtuali?
8. Diverse aspecte privitoare la membri virtuali.

Teme de casă:

Aplicația1
Dându-se clasa de baza **Lista**, să se construiască două clase derivate **ListaNeordonata** şi **ListăOrdonata**, folosind funcțiile virtuale pentru adăugare a unui element în listă şi pentru afişarea elementelor unei liste.

Aplicația2
Dându-se clasa de baza **Lista**, să se construiască două clase derivate **ListaSimplă** şi **ListăDublă**, folosind funcțiile virtuale pentru adăugare a unui element în listă şi pentru afişarea elementelor unei liste.

Laborator 8 POO:

Moştenire multiplă

Scopul lucrării: aprofundarea mecanismelor de moştenire, şi anume a unui nou concept de moştenire: moştenirea multiplă.

Beneficiul: utilizarea moştenirii multiple permite reutilizarea resurselor existente(a claselor) pentru a genera noi resurse. De exemplu, dacă avem o clasa numită **punct** care desenează un punct şi o clasă **culoare,** atunci putem crea o nouă clasă **linie** (linia este formată din puncte colorate), derivată din cele două clase, folosindu-ne astfel de codul scris în clasa punct şi în clasa culoare. Cu alte cuvinte putem realiza ierarhii de clase foarte complexe.

Scurtă prezentare: În cadrul acestui laborator se va studia mecanismul moştenirii multiple.

Prezentarea laboratorului:

Exemplu:

```
class B1
{
        //...
}
class B2
{
        //...
}
class B3
{
        //...
}
class D1 : B1
{
        B2 b1;
        B3 b2;
};
```

Din punct de vedere al structurii de date, clasa **D1** este echivalenta cu forma:

```
class D1 : B1, B2, B3
{
        //...
};
```

Din punct de vedere funcțional există totuşi diferențe, cum ar fi, de exemplu, modul de apel al metodelor claselor membre. Moştenirea multiplă elimină aceste neajunsuri. Astfel, o clasă de derivată poate avea simultan mai multe clase de bază.

Există totuşi şi unele restricții:
1. O clasă derivată **D** nu poate moşteni direct de doua ori aceeaşi clasă **B**.

2. O clasă derivată **D** nu poate moşteni simultan o clasă **B** şi o altă **D1**, derivată tot din "B". De asemenea, nu pot fi moştenite simultan două clase **D1** şi **D2** aflate în relaţia de moştenire **B** ->... ->**D1**->...->**D2**.
Exemplu:

```
class B
{
        //...
};
class C1 : B
{
        //…
};
class C2 : C1
{
        //…
};
class D1 : B, C2   //EROARE
{
        //…
};
class D2 : C1, C2   //EROARE
{
        //…
};
```

În schimb, este corectă o moştenire de genul :

```
class B
{
        //...
};
class D1 : B
{
        //…
};
class D2 : B
{
        //…
};
class D : D1, D2
{
        //…
};
```

Pentru a realiza o moştenire multiplă este suficient să se specifice după numele clasei derivate o listă de clase de bază separate prin virgulă (evident, aceste nume pot fi prefixate cu modificatori de acces ***public*** sau ***privat***).

Ordinea claselor de bază nu este indiferentă. Apelul constructorilor respectivi se va face exact în ordinea enumerării numelor claselor de bază. Ca şi în cazul moştenirii simple, apelul

constructorilor tuturor claselor de bază se va efectua înaintea eventualelor inițializări de variabile-membru. Mai mult, aceasta ordine nu poate fi modificată nici chiar de ordinea apelului explicit al constructorilor claselor de bază în cadrul constructorilor clasei derivate.

Exemplu:

```
class B1
{
        char c;
  public :
        B1(char c1) : c(c1) {};
};
class B2
{
        int c;
   public :
        B2(int c1) : c(c1) {};
};
class D : B1, B2
{
        char a, b;
   public :
        D(char a1, char b1) : a(a1), b(b1)
        B2((int) a1), B1(b)
  {
        //...
  }
};
void main(void)
{
  D d(3,3);
}
```

Indiferent de ordinea inițializării variabilelor şi de cea a constructorilor specificați explicit, în definirea constructorului clasei derivate:

```
        D(char a1,char b1) : a(a1),b(b1)
        B2((int) a1),B1(b1) {};
```

succesiunea operațiilor va fi următoarea : apelul constructorului **B1**, apoi **B2** şi, doar la sfârşit, inițializarea lui **a** şi **b**.

O situație cu totul particulară o au aşa-zisele CLASE VIRTUALE. Pentru a avea o imagine cât mai clară asupra problemei , să analizăm exemplul de mai jos :

Exemplu :

```
class B
{
        //…
};
```

```
class D1 : B
{
        //...
};

class D2 : B
{
        //…
};

class M1 : D1,public D2
{
        //…
};

class M2 : virtual D1, virtual public D2
{
        //…
};
```

În urma moştenirii claselor **D1** şi **D2**, clasa **M1** va îngloba două obiecte de tipul **B** (câte unul pentru fiecare din cele două clase de bază).

În practică, ar putea exista situaţii în care este de dorit moştenirea unui singur obiect de tip **B**. Pentru aceasta se va proceda ca şi în cazul clasei **M2**: se vor prefixa cele două clase de bază cu cuvântul cheie ***virtual***. În urma acestei decizii, clasa **M2** nu va conţine decât **O SINGURA INSTANTIERE A CLASEI "B"**.

Cui aparţine această unică instanţiere (lui **D1** sau **D2**)? Instanţierea în cauză va aparţine lui **D1**, adică primei clase virtuale (din lista claselor de bază) care a fost derivată din **B**.

Definirea claselor virtuale ne obligă să modificăm regulile de apelare a constructorilor claselor de bază. În cele ce urmează vom prezenta noul set de reguli :

R1. Constructorii claselor de bază virtuale vor fi apelaţi **INAINTEA** celor corespunzând unor clase de bază ne-virtuale.

R2. În caz că o clasă posedă mai multe clase de bază virtuale, constructorii lor vor fi apelaţi în ordinea în care clasele au fost declarate în lista claselor de bază. Apoi vor fi apelaţi (tot în ordinea declarării) constructorii claselor de bază ne-virtuale, şi abia în cele din urmă constructorul clasei derivate.

R3. Chiar dacă în ierarhia claselor de bază există mai mult de o instanţiere a unei clase de bază virtuale, constructorul respectiv va fi apelat doar o singură dată.

R4. Daca în schimb, în aceeaşi ierarhie există atât instanţieri virtuale cât şi ne-virtuale ale unei aceleiaşi clase de bază, constructorul va fi apelat:
O SINGURA DATA - pentru toate instanţierile virtuale;
DE "N" ORI - pentru cele ne-virtuale (dacă există "N" instanţieri de acest tip).

Partea practică a laboratorului:

Aplicația1
Studenții vor dezvolta structura de mai jos:

```
enum Boolean {false, true};

class Location {
protected:
   int X;
   int Y;
public:
   Location(int InitX, int InitY) {X = InitX; Y = InitY;}
   int GetX() {return X;}
   int GetY() {return Y;}
};

class Point : public Location {
protected:
   Boolean Visible;
public:
   Point(int InitX, int InitY);
   virtual void Show();          // Show si Hide sunt virtuale
   virtual void Hide();
   Boolean IsVisible() {return Visible;}
   void MoveTo(int NewX, int NewY);
};

class Circle : public Point {  // Derivata din  class Point care este
                               // derivata din  class Location
protected:
   int Radius;
public:
   Circle(int InitX, int InitY, int InitRadius);
   void Show();
   void Hide();
   void Expand(int ExpandBy);
   void Contract(int ContractBy);
};
```

Aplicația2
Aplicația de mai sus va fi extinsă prin crearea claselor Line şi Poligon.

Întrebări:

1. Ce este moştenirea?
2. Care sunt mecanismele moştenirii?
3. Ce este moştenirea multiplă?
4. O clasă poate moştenii mai multe clase de bază?
5. Ordinea claselor de bază, moştenite de o clasă, este importanţă?
6. Ce sunt constructorii de copiere?
7. Ce sunt clasele virtuale?
8. Cum se apelează constructorii claselor derivate?
9. Ordinea de apel a constructorilor claselor derivate este importantă?

Teme de casă:

Aplicatia1
Să se dezvolte, similar cu aplicația prezentată la laborator, o aplicație pentru clasele: Om, Student, OnCampus şi OffCampus. Clasele Om şi Student sunt considerate clase de bază.

Aplicația2
Aceleaşi cerințe pentru clasele: Roată, Vehicul, AutoLimuzina şi AutoTransport. Clasele Roată şi vehicul vor fi considerate clase de bază.

1. Octavian Catrina, Iuliana Cojocaru, *"TURBO C++"*, Editura Teora, Bucureşti 1993.
2. Vasile Stoicu-Tivadar, *„Programare Orientata pe Obiecte"*, Editura Orizonturi Universitare, Timişoara 2000, pp. 147-167, 223-259.

Laborator 9 POO:

Şabloane în C++

În acest laborator se va discuta despre şabloanele utilizate în C++. Veţi utiliza şabloanele pentru construirea unei clase tip colecţie CStack care poate fi folosită pentru stocarea oricărui tip de obiect. Colecţii şi containere sunt două denumiri date obiectelor care se folosesc pentru a stoca alte obiecte.

Ce sunt şabloanele?

Şabloanele sunt folosite cu clase şi funcţii care acceptă parametri la folosirea clasei. Un şablon de clasă sau de funcţie poate fi asimilat unui bon de comandă care include câteva spaţii albe care se completează la expedierea comenzii, În loc de a crea o clasă tip listă pentru matrice de pointeri şi o altă clasă de acelaşi tip pentru char, va putea fi folosită o singură clasă şablon drept clasă tip listă.

De ce să folosim şabloanele?

Şabloanele sunt foarte utile în colecţii, deoarece o singură clasă tip colecţie bine concepută care foloseşte şabloane poate fi imediat refolosită pentru toate tipurile incorporate. În plus, utilizarea şabloanelor pentru clasele tip colecţie contribuie la eliminarea unuia dintre motivele principale care necesită folosirea conversiilor forţate.

Cum se folosesc şabloanele?

Sintaxa folosită pentru declararea şi utilizarea şabloanelor poate părea dificilă la început. Ca în majoritatea cazurilor, practica este soluţia. Sintaxa pentru utilizarea unui şablon este relativ simplă. O colecţie de întregi din clasa CStack este declarata sub forma:

```
CStack<int> stackOfInt;
```

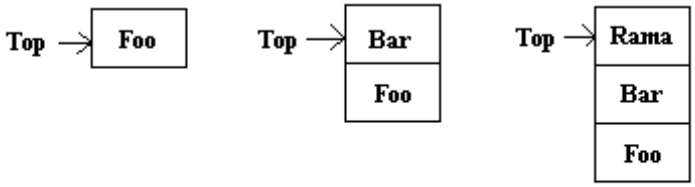
Componenta `<int>` reprezintă lista de parametri ai şablonului. Lista de parametri este utilizată pentru a instrui compilatorul cum să creeze acest exemplar al şablonului. Aceasta este cunoscută şi sub numele de multiplicare, deoarece urmează a fi creat un nou exemplar al şablonului, folosind argumentele date. Nu vă faceţi probleme în legătură cu semnificaţia efectivă a parametrilor de şablon folosiţi pentru CStack; vom discuta despre aceştia în continuare.
De asemenea se poate folosi cuvântul cheie `typedef` pentru a simplifica citirea declaraţiilor. Dacă folosiţi typedef pentru a crea un tip nou, se poate folosi noul nume în locul unei sintaxe de şablon mai lungi, după cum se prezintă în listingul 1.

Listing 1. Utilizarea cuvântului cheie typedef pentru a simplifica declarațiile de șablon.

```
typedef CStack <int> INTSTACK;
INTSTACK stivaDeInt;
INTSTACK TotStivaDeInt;
```

O clasă de şabloane CStack

Crearea unei clase bazate pe şabloane implică cu puțin mai multă muncă decât crearea unei clase non-şablon. În acest paragraf vă veți crea propria clasă stivă, bazându-vă pe şabloane.
O stivă este o colecție de obiecte care permite articolelor să fie eliminate sau adăugate dintr-o singură poziție logică, şi anume "vârful" stivei. Articolele sunt introduse (pushed on) sau extrase (popped off) din stivă. Dacă într-o stivă se află două sau mai multe articole, se poate accesa numai ultimul element din stivă, după cum se vede in Figura 2:



O stivă permite accesarea numai a elementului din vârful acesteia

Fig. 2.

La crearea unei definiții pentru şabloanele dumneavoastră, este un procedeu comun utilizarea înlocuitorului T pentru a reprezenta tipul care va fi specificat ulterior, la multiplicarea şablonului. Un exemplu de şablon tip stivă este prezentat în listingul 3.

Listing 3. Clasa de şabloane CStack

```
template <class T> class CStack
{
public:
    CStack ();
    virtual ~CStack ();
    int IsEmpty () const;
    T Pop ();
    void Push (const T& item);
private:
    CStack<T> (const CStack& T) {};
    T* m_p;
    int m_nStored;
    int m_nDepth;
    enum {GROW_BY = 5};
};

// Constructori
```

```
template <class T> CStack<T>::CStack ()
{
    m_p = 0;
    m_nStored = 0;
    m_nDepth = 0;
}

template <class T> CStack<T>::~CStack ()
{
    delete [] m_p;
}

// Operații
template <class T> int CStack<T>::IsEmpty () const
{
    return m_nStored == 0;
}

template <class T> void CStack<T>::Push (const T& item)
{
    if (m_nStored == m_nDepth)
    {
        T* p = new T [m_nDepth + GROW_BY];
        for (int i = 0; i <m_nDepth; i++)
        {
            p [i] = m_p [i];
        }
        m_nDepth += GROW_BY;
        delete [] m_p;
        m_p = p;
    }
    m_p [m_nStored] = item;
    m_nStored ++;
}

template <class T> int CStack<T>::Pop ()
{
    m_nStored --;
    return m_p [m_nStored];
}
```

Declarația unui şablon cere compilatorului să utilizeze un tip care va fi precizat mai târziu la multiplicarea efectivă a şablonului. La începutul declarației se foloseşte următoarea sintaxă:

```
template <class T> CStack
```

Aceasta arată compilatorului că un utilizator al clasei CStack va furniza un tip când şablonul va fi multiplicat şi că acel tip trebuie folosit oriunde este plasat T în întreaga declarație de şablon.
Funcțiile membre ale clasei CStack folosesc o declarație similară. Dacă CStack ar fi fost o clasă non-şablon, funcția membru IsEmpty ar fi avut următorul aspect:

```
int CStack<T>::IsEmpty ()
{
    return m_nStored;
```

```
}
```

Deoarece CStack este o clasă şablon, sunt necesare informaţiile referitoare la şablon. O altă modalitate de definire a funcţiei IsEmpty este inserarea informaţiilor despre şablon într-o linie separată, înainte de restul funcţiei.

```
template <class T>
int CStack<T>::IsEmpty () const
{
    return m_nStored;
}
```

Listingul 3 prezintă un exemplu simplu care foloseşte clasa CStack.

Listing 3. Utilizarea şablonului CStack

```
# include <afx.h>
# include <iostream.h>
# include "stack.h"

int main (int argc, char* argv[])
{
    CStack<int> theStack;
    int i = 0;
    while (i <5)
    {
        cout <<"Pushing a " <<i <<endl;
        theStack.Push (i ++);
    }
    cout <<"Toate articolele inserate" <<i <<endl;
    while (theStack.IsEmpty () != FALSE)
    {
        cout <<"Extrag un " <<theStack.Pop () <<endl;
    }
    return 0;
}
```

Utilizarea funcţiilor şablon

O altă utilizare importantă a şabloanelor o constituie funcţiile şablon. Dacă sunteţi familiarizat cu limbajul C, probabil aţi utilizat funcţii macro preprocesor pentru a crea funcţii cu suprasolicitare redusă (low-overhead functions). Din păcate, funcţiile macro pentru preprocesor nu sunt chiar funcţii, şi ca atare nu au nici un fel de modalitate de verificare a parametrilor.
Funcţiile şablon permit utilizarea unei funcţii pentru o gamă largă de tipuri de parametri. În listingul 4 se prezintă o funcţie şablon care comută valorile a doi parametri.

Listing 4. Exemplu de funcţie şablon care îşi permută parametri.

```
template <class T>
```

```
void SchArg (T& foo , T& bar)
{
    T temp;
    temp = foo;
    foo = bar;
    bar = temp;
}
```

Utilizarea unei funcții şablon este foarte simplă. Compilatorul se ocupă de toate. Exemplul din listingul 5 afişează un mesaj înainte şi după apelarea funcției şablonului SchArg.

Listing 5. Utilizarea SchArg pentru permutarea conținutului a două obiecte char*.

```
# include <afx.h>
# include <iostream.h>
# include "scharg.h"

int main ()
{
    char *szMsjUnu ("Salut");
    char *szMsjDoi ("La Revedere");
    cout <<szMsjUnu << "\t" <<szMsjDoi <<endl;
    SchArg (szMsjUnu, szMsjDoi);
    cout <<szMsjUnu << "\t" <<szMsjDoi <<endl;
    return EXIT_SUCCES;
}
```

Partea practică a laboratorului

Aplicația 1:

Să se scrie o aplicație C++ care conține o clasă şablon pentru implementarea unei liste simplu înlănțuite. Lista va putea avea noduri de orice tip de data fundamental (char, int, double etc.). Se vor implementa funcții pentru adăugarea, ştergerea şi căutarea unui nod din listă, precum şi o funcție pentru afişarea listei.

Aplicația 2:

Aceeaşi aplicație pentru o listă dublu înlănțuită.

.

Bibliografie:

1. Octavian Catrina, Iuliana Cojocaru, "*TURBO C++*", Editura Teora, Bucuresti, 1993.

2. Vasile Stoicu-Tivadar, „*Programare Orientata pe Obiecte*”, Editura Orizonturi Universitare, Timisoara 2000.

3. Erich Gamma, Richard Helm, R. Johnson, J. Vlissides, „*Design Patterns - Sabloane de proiectare*”, Editura Teora, Bucureşti 2002