PROGRAMARE ORIENTATĂ PE OBIECTE

LABORATOR 10

Anul universitar 2023 – 2024

Semestrul II

1 Noțiuni teoretice

1.1 Polimorfism

1.1.1 Generalități

Cele 4 mari concepte ale Programarii Obiectuale sunt: Incapsulare, Polimorfism, Mostenire si Abstractizare. (Echivalent in engleza, pentru documentare suplimentara: Encapsulation, Polymorphism, Inheritance, Abstraction).

Polimorfism = abilitatea unei functii cu un anumit nume sa aiba comportamente diferite, in functie de parametrii de intrare.

In C++, avem mai multe tipuri de polimorfism:

- Compile Time Polymorphism
- Run Time Polymorphism

1.1.2 Compile Time Polymorphism

Acest tip de polimorfism este obtinut prin supraincarcarea unei functii (Function Overloading) sau al unui operator (Operator Overloading).

Overloading (supraincarcarea) = posibilitatea de a avea intr-o clasa mai multe metode cu acelasi nume, fiind diferentiate prin semnatura.

1.1.3 Run Time Polymorphism

Acest tip de polimorfism este obtinut cand suprascriem o functie (Function Overriding).

Overriding (suprascriere) = redefinirea metodelor care sunt existente in clasa parinte de catre clasa copil in vederea specializarii lor. Metodele in clasa parinte nu sunt afectate/modificate. Observatie:

- Se pot suprascrie doar metodele vizibile pe lantul de mostenire (public, protected)
- metoda din clasa copil poate suprascrie o metoda din clasa parinte daca au aceeasi semnatura si acelasi tip de return.

Semnatura unei metode consta in:

- numele metodei

- numarul si tipul parametrilor

```
#include <iostream>
using namespace std;
class OOP {
    public:
        void fct (int x) { //Avem un parametru de tip int
            cout << "x = " << x << endl;
        }
        void fct (float x) { //Avem un parametru de tip float
            cout << "x = " << x << endl;
        }
        void fct (int x, int y) { //Avem doi parametri
            cout << "x + y = " << x + y << endl;
        }
};

int main () {
    OOP laborator;

    OOP.fct(10); //o sa fie apelata prima metoda (cea care are parametrul de tip int)
    OOP.fct(9.99); //o sa fie apelata a doua metoda (cea care are parametrul de tip
float)
    OOP.fct(5,5); //o sa fie apelata a treia metoda (cea cu doi parametri)

    return 0;
}
```

Observam ca, desi au acelasi nume, semnatura metodelor difera doar prin tipul si numarul parametrilor.

1.2 Funcții virtuale

1.2.1 Clase si Functii Abstracte

O metoda virtuala se numeste abstracta (pura) daca nu are implementare:

```
virtual tip_returnat metoda(lista parametrii) = 0;
```

O clasa se numeste abstracta daca are cel putin o metoda abstracta.

Uneori implementarea tuturor functiilor dintr-o clasa de baza nu poate fi realizata, deoarece nu stim cu certitudine implementarea acestora. Sa presupunem ca avem o clasa de baza Shape. Nu putem implementa metoda draw() in Shape, insa stim cu siguranta ca fiecare clasa derivata o sa aiba aceasta metoda implementata.

```
#include <iostream>
using namespace std;

//Clasa abstracta
class Shape {
    //Membrii clasei
    protected:
```

```
        int id;
    public:
        //Functie abstracta
        virtual void draw() = 0;
};

//Clasa care mosteneste Shape si implementeaza draw
class Circle: public Shape {
    public:
        void draw() { cout << "DRAW CIRCLE!" << endl; }
};

int main () {
    Circle c;
    c.draw();   //Se va afisa DRAW CIRCLE!
    return 0;
}
```

Functiile abstracte sunt folosite in cazul in care metoda nu are o implementare posibila in clasa de baza, insa dorim sa o declaram virtuala.

NU se pot instantia obiecte de tipul unei clase abstracte, DAR se pot declara pointeri de acel tip.

Suntem obligati, ca eventual, sa implementam functia intr-o clasa derivata, altfel, si aceasta clasa derivata o sa fie tot abstracta.

1.2.2 Interfete

O clasa fara atribute si cu toate metodele abstracte se numeste interfata.

Interfetele si clasele abstracte sunt foarte importante pentru dezvoltarea ierarhiilor de clase – acestea reprezinta o schita a ceea ce trebuie implementat/suprascris in clasele derivate.

Clasele derivate din una sau mai multe interfete sunt obligate ca, eventual, sa furnizeze implementari pentru toate metodele abstracte.

Ca exemplu, consideram urmatoarea interfata Shape, cu metodele abstracte perimeter() si area(). Avand in vedere faptul ca, pentru fiecare figura geometrica in parte exista alte formule pentru calculul perimetrului si al ariei, fiecare clasa derivata va avea propria implementare pentru cele doua metode.

```
//Interfata
class Shape {
    //Nu are atribute, toate metodele sunt abstracte
    public:
        virtual double perimeter() = 0;
        virtual double area() = 0;
};

//Clasa care implementeaza interfata Shape
```

```
class Square: public Shape {
    protected:
        double L;
    public:
        double perimeter() { return 4 * L; }
        double area() { return L * L; }
};

//Alta clasa care implementeaza interfata Shape
class Circle: public Shape {
    protected:
        double R;
    public:
        double perimeter() { return 2 * 3.14 * R; }
        double area() { return 3.14 * R * R; }
};
```

Interfetele pot fi folosite pentru a declara liste neomogene de obiecte diferite.

Spre exemplu, daca avem obiecte de tip Student si obiecte de tip Angajat, putem declara o interfata pe care cele doua tipuri sa o mosteneasca. Astfel putem declara o lista de angajati si studenti.

1.2.3 Legare statică și dinamică

Functiile virtuale permit claselor derivate sa inlocuiasca implementarea metodelor din clasa de baza - suprascriere/supraincarcare/override- si pun la dispozitie mecanismul de legare dinamica.

O functie virtuala este membra a clasei de baza si este redefinita(overriden) de o clasa derivata.

Pentru a intelege mai bine importanta folosirii functiilor virtuale, introducem conceptul de binding(legare).

Legarea (Binding) reprezinta conectarea unui apel de functie cu functia in sine (adresa functiei).

Legarea poate fi:

- Statica/timpurie (la compilare)
- Dinamica/tarzie (in momentul executie)

1.2.4 Legarea Statica

Legarea statica (Early binding):

- se realizeaza la compilare (compile time, inainte de rularea aplicatiei)
- este realizata de catre compilator si editorul de legaturi (linker)

Asa cum sugereaza si numele, compilatorul(sau linker-ul) asociaza in mod direct apelului de functie o adresa.

Orice apel normal de functie(fara virtual)este legat static.

In C toate apelurile de functii presupun realizarea unei legaturi statice.

Functiile membre ale unei clase primesc adresa obiectului care face apelul.

- tip_date obiect;
- obiect.functie();

In functie de tipul obiectului de la acea adresa - compilatorul si editorul de legaturi stabilesc daca:

- nume_functie() e membra a clasei tip_date (e declarata)
- este implementata
- daca este implementata, se face legarea statica si se poate apela acea functie
- daca functia era declarata ca membra a clasei, dar nu e implementata arunca eroarea

```
[Linker error] undefined reference to 'tip_date::nume_functie()'
```

In cazul apelului prin intermediul pointerilor.

```
tip_date_baza *obiect = new tip_date_baza;
```

obiect->functie();

Procedeul este similar:

- compilatorul si editorul de legaturi stabilesc daca functia invocata de un pointer este de tipul acelui pointer (daca poate fi apelata)
- compilatorul foloseste tipul static al pointerului pentru a determina daca invocarea functiei membre este legala.

```
#include<iostream>
using namespace std;

class Baza
{
public:
    void afisare() { cout<<" In Baza \n"; }
};

class Derivata: public Baza
{
public:
    void afisare() { cout<<"In Derivata \n"; }
};

int main(void)
{
    Baza *bp = new Derivata;
```

```
    // Compilatorul vede tipul pointerului si
    // apeleaza functia din clasa Baza
    bp->afisare();

    return 0;
}
```

In urma rularii programului, se va afisa:

```
In Baza
```

Daca vrem sa apelam functia de afisare pentru obiectul catre care pointeaza bp, este necesara o conversie explicita a pointerului bp din (Baza*) in (Derivata*), astfel incat legatura sa se faca pentru tipul de date al obiectului catre care pointeaza bp.

```
((Derivata*)bp)->afisare();
```

Dar

- aceasta solutie nu e robusta
- trebuie mereu sa ne punem problema catre ce tip de obiect pointeaza pointerul de tip clasa de baza si sa facem conversii explicite pentru a apela functia dorita
- e predispusa erorilor logice

Alternativa pusa la dispozitie in C++ este mecanismul de legare dinamica/tarzie (dynamic/late binding) - nu trebuie sa memorez catre ce tip de obiect se pointeaza in timpul executiei

1.2.5 Legarea Dinamica

Legare dinamica/tarzie(Late binding):

- legarea se face dupa compilare, la rulare
- in functie de tipul dinamic al pointerului (tipul obiectului catre care se pointeaza)
- se poate realiza doar in contextul apelului de functii prin intermediul pointerilor

In acest caz, compilatorul identifica tipul obiectului la momentul rularii si apoi apeleaza functia potrivita.

Pentru a folosi acest tip de legare, compilatorul trebuie informat ca exista posibilitate ca, la rulare, sa se doreasca apelarea unei functii de tipul dinamic al obiectului.

Pentru acest lucru vom introduce un semnal in cod → functii virtuale

Legarea dinamica poate fi implementata doar in cazul limbajelor compilate!

```
#include<iostream>
using namespace std;
```

```
    class Baza
    {
    public:
        virtual void afisare() { cout<<" In Baza \n"; }
    }; //legarea pentru functia afisare se face la rulare

    class Derivata: public Baza
    {
    public:
        void afisare() { cout<<"In Derivata \n"; }  //supraincarcare
    };  //legarea pentru functia afisare se face la rulare; e virtuala
    int main(void)
    {
        Baza *bp = new Derivata; //legare dinamica (in functie de tipul dinamic), apel
Derivata::afisare()
        bp->afisare();
        return 0;
    }
```

Observam asadar ca de data aceasta, se va afisa

```
 In Derivata
```

Legarea dinamica se poate face doar folosind functii virtuale si pointeri.

1.3 Polimorfism si functii virtuale

Polimorfismul se poate realiza prin:

- supradefinirea functiilor - functii cu acelasi nume, dar semnaturi diferite - care se comporta diferit in functie de context – in functie de modul in care sunt apelate – chiar daca au acelasi nume; polimorfism ad hoc
- suprascrierea functiilor – functii virtuale: acelasi pointer poate sa aiba comportamente diferite la apelul unei metode, in functie de tipul lui dinamic

Avand in vedere importanta lor, de ce nu folosim exclusiv functii virtuale? De ce nu realizeaza C++ decat legaturi dinamice? Pentru ca legarea dinamica nu este la fel de eficienta ca legarea statica (asa ca o folosim doar cand e nevoie).

Cand folosim functii virtuale atunci? De fiecare data cand vrem ca in clasa derivata sa modificam/adaptam/suprascriem comportamentul unei functii deja implementate in clasa de baza.

Cand este important acest mecanism?

- Functiile virtuale sunt folosite pentru a implementa polimorfismul in momentul rularii (” Run time Polymorphism“)
- Un alt avantaj al functiilor virtuale este ca permit realizarea de liste neomogene de obiecte

- Dezvoltarea de biblioteci in spiritul POO.

1.4 Moștenire multiplă

Putem să derivăm mai multe clase din clasa de bază? Da, se poate face acest lucru. Putem să avem oricâte clase derivate dintr-o clasă de bază și oricâte niveluri de derivare (cât timp este logic).

Pot să am mai multe clase părinte în același timp? Desigur, acest mecanism se numește moștenire multiplă. Exemplu:

Avem următoarele clase de bază:

- Angajat (are salariu, sperăm că mare, și lucrează la o firmă)
- Student (are note, învață la o facultate)

Presupunem ca studentul nostru se angajează, astfel se creează o nouă clasa Student_Angajat (are note, are salariu, lucrează la o firmă și studiază la o facultate). Observăm că noua clasă, Student_Angajat, are ca părinți ambele clase (Angajat, Student).

În C++, o clasă poate moșteni de la mai multe clase doar prin simpla precizare a mai multor clase de bază, separate prin virgulă, în lista claselor de bază (adică, după două puncte). De exemplu, dacă programul conține o anumită clasă care să afișeze pe ecran, clasă numită Iesire, și vrem ca Dreptunghi și Triunghi să o moștenească și pe ea, pe lângă clasa Poligon, vom scrie:

```
class Dreptunghi: public Poligon, public Iesire;
class Triunghi: public Poligon, public Iesire;
```

În continuare, avem exemplul complet:

```
// mostenire multipla
#include <iostream>
using namespace std;

class Poligon {
  protected:
    int latime, inaltime;
  public:
    Poligon (int a, int b) : latime(a), inaltime(b) {}
};

class Iesire {
  public:
    static void print (int i);
};

void Iesire::print (int i) {
  cout << i << '\n';
}
```

```
class Dreptunghi: public Poligon, public Iesire {
  public:
    Dreptunghi (int a, int b) : Poligon(a,b) {}
    int aria ()
      { return latime*inaltime; }
};

class Triunghi: public Poligon, public Iesire {
  public:
    Triunghi (int a, int b) : Poligon(a,b) {}
    int aria ()
      { return latime*inaltime/2; }
};

int main () {
  Dreptunghi drept (4,5);
  Triunghi trng (4,5);
  drept.print (drept.aria());
  Triunghi::print (trng.aria());
  return 0;
}
```

Un alt exemplu:

```
#include <iostream>
using namespace std;
class Baza1 {
    protected:
        int atribut1;
    public:
        Baza1(int i = 0) : atribut1(i){}
        void set_atribut1(int i) {
            atribut1 = i;
        }
        void afisare_atribut1() {
            cout << "\nAtribut1 = " << atribut1 << endl;
        }
};
class Baza2 {
    protected:
        int atribut2;
    public:
        Baza2(int i = 0) : atribut2(i){}
        void set_atribut2(int i) {
            atribut2 = i;
        }
        void afisare_atribut2() {
            cout << "\nAtribut2 = " << atribut2 << endl;
        }
};
class Derivata : public Baza1, public Baza2 {
    private:
```

```
        int atribut3;
    public:
        Derivata() {}
        Derivata(int a1, int a2, int a3) : Baza1(a1),Baza2(a2),atribut3(a3){}
        void set_atribut3(int i) {
            atribut3 = i;
        }
        void set_atribute(int a1, int a2, int a3) {
            atribut1 = a1;
            atribut2 = a2;
            atribut3 = a3;
        }
        void afisare_atribute() {
            cout << "\nAtribut1 = " << atribut1 << "\nAtribut2 = "
             << atribut2 << "\nAtribut3 = " << atribut3 << endl;
        }
};
int main() {
    Derivata obiect(1,2,3);
    obiect.afisare_atribute();
    cout << "\n" << "_________________" << endl;
    obiect.set_atr1(5);
    obiect.set_atr2(6);
    obiect.set_atr3(7);
    obiect.afisare_atribut1();
    obiect.afisare_atribute();

    return 0;
}
```

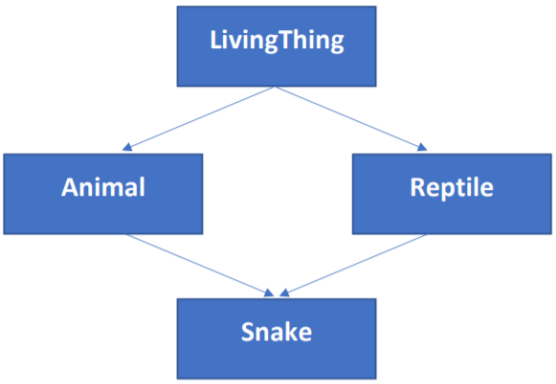
Când utilizăm moștenirea multiplă, putem întâmpina următoarele probleme:

- atribute si metode cu același nume în clasele de bază
- derivare dublă indirectă din clasa de bază

altele

1.5 Moștenirea, polimorfismul și funcțiile virtuale

Desi mostenirea multipla pare a fi un procedeu util, aceasta poate duce la multe ambiguitati, cea mai mare dintre acestea fiind Problema Diamantului (numita si “death diamond”). Sa consideram urmatoarea ierarhie de clase:



In acest exemplu, avem o clasa de baza numita LivingThing avand metoda breathe(). Clasele Animal si Reptile mostenesc clasa LivingThing si suprascriu in moduri diferite metoda breathe(). Clasa Snake mosteneste ambele clase, Animal si Reptile, insa nu suprascrie metoda breathe().

In momentul de fata, daca apelam metoda breathe() din Snake, acesta nu va sti ce metoda sa apeleze, daca sa fie cea suprascrisa in clasa Animal sau cea suprascrisa in clasa Reptile.

Cum rezolvam insa Problema Diamantului? Folosind derivarea virtuala.

```
class Animal: virtual public LivingThing
class Reptile: virtual public LivingThing
```

Clasele de baza virtuale sunt utile in cadrul mostenirii multiple cand o serie de clase sunt derivate din aceeasi clasa de baza, iar aceasta urmeaza a fi clase parinte pentru o alta clasa.

Efectul acestei mosteniri virtuale nu este sesizat in clasele Animal si Reptile, ci se observa in urmatorul nivel de derivare, clasa Snake.

Pentru a se crea un obiect de tip Snake, se vor apela constructorii claselor Animal and Reptile, dar se va apela o singura data constructorul clasei LivingThing, astfel Snake va avea o singura instanta a clasei LivingThing.

2 Aplicații

2.1 Raft

Implementați un sistem simplu de gestionare a produselor dintr-un magazin. Definiți o clasă de bază Produs care să conțină informații despre nume, preț, stoc și codul produsului. Apoi, derivați două clase, Aliment și Electronic, care să reprezinte două tipuri diferite de produse. Clasa Aliment ar trebui să conțină informații suplimentare despre perioada de expirare a alimentului, în timp ce clasa Electronic ar trebui să conțină informații despre perioada de garanție a produsului electronic. Implementați metode specifice pentru fiecare tip de produs, cum ar fi afișarea informațiilor produsului, calcularea valorii stocului și afișarea etichetei produsului.

Creați, de asemenea, o clasă Raft care să reprezinte un raft din magazin. Această clasă ar trebui să permită adăugarea și eliminarea de produse de pe raft, să afișeze informații despre produsele de pe raft și să afișeze etichetele acestora.

```
class Produs
{
protected:
    char* nume;
    double pret;
    int stoc;
    char cod[10];
public:
    Produs();
    Produs(const char*, double, int, const char[]);
    Produs(const Produs&);
    virtual ~Produs();
    virtual void Print();
    virtual double Stoc();
    virtual void Eticheta();
    virtual int Garantie();
    virtual void Citeste(istream&);
    friend istream& operator>>(istream& in, Produs& p)
    {
        p.Citeste(in);
        return in;
    }
    virtual Produs* Clona();
};

class Aliment : public Produs
{
private:
    int expira;
public:
    Aliment();
    Aliment(const char*, double, int, const char[], int);
    Aliment(const Aliment&);
```

```
    ~Aliment();
    void Citeste(istream&);
    void Print();
    double Stoc();
    void Eticheta();
    Produs* Clona();
};

class Electronic : public Produs
{
private:
    int garantie;
public:
    Electronic();
    Electronic(const char*, double, int, const char[], int);
    Electronic(const Electronic&);
    ~Electronic();
    void Citeste(istream&);
    void Print();
    void Eticheta();
    int Garantie();
    Produs* Clona();
};

class Raft
{
private:
    int nrProd;
    Produs* produse[100];
public:
    Raft();
    Raft(const Raft&);
    ~Raft();

    Raft operator=(const Raft&);

    Raft& Adauga(Produs*);
    Produs* Elimina();
    void InfoProdRaft();
    void ContinutRaft();
};
```

3 Rezolvări

3.1 Raft

```
#include <iostream>
#include <string.h>

using namespace std;

class Produs
{
protected:
    char* nume;
    double pret;
    int stoc;
    char cod[10];
public:
    Produs();
    Produs(const char*, double, int, const char[]);
    Produs(const Produs&);
    virtual ~Produs();
    virtual void Print();
    virtual double Stoc();
    virtual void Eticheta();
    virtual int Garantie();
    virtual void Citeste(istream&);
    friend istream& operator>>(istream& in, Produs& p)
    {
        p.Citeste(in);
        return in;
    }
    virtual Produs* Clona();
};

// clasa Produs - constructor implicit
Produs::Produs()
{
    nume = NULL;
    pret = 0;
    stoc = 0;
    for(int i = 0; i < 10; i++)
    {
        cod[i] = '\0';
    }
}

// clasa Produs - constructor cu paramteri
Produs::Produs(const char* n, double p, int s, const char c[10])
{
    nume = new char[strlen(n) + 1];
    strcpy(nume, n);
    pret = p;
    stoc = s;
    strcpy(cod, c);
```

```
    cod[9] = '\0';
}

//clasa Produs - constructor de copiere
Produs::Produs(const Produs& p)
{
    nume = new char[strlen(p.nume) + 1];
    strcpy(nume, p.nume);
    pret = p.pret;
    stoc = p.stoc;
    strcpy(cod, p.cod);
    cod[9] = '\0';
}

// clasa Produs - destructor
Produs::~Produs()
{
    delete[] nume;
}

void Produs::Citeste(istream& in)
{
    cout << "Tipul produsului este Produs. Introdu datele: \n";

    char buffer[100];
    cout << "Introdu numele: ";
    in >> buffer;
    delete[] nume;
    nume = new char[strlen(buffer) + 1];
    strcpy(nume, buffer);

    cout << "Introdu pretul: ";
    in >> pret;

    cout << "Introdu stocul: ";
    in >> stoc;

    cout << "Introdu codul: ";
    in >> buffer;
    strcpy(cod, buffer);
    cod[9] = '\0';

    cout << "\n";
}

// clasa Produs - afisare informatii produs
void Produs::Print()
{
    cout << "Nume: " << nume << "\n";
    cout << "Pret: " << pret << "\n";
    cout << "Stoc: " << stoc << "\n";
    cout << "Cod: " << cod << "\n";
}
```

```
// clasa Produs - returnare stoc
double Produs::Stoc()
{
    return pret * stoc;
}

// clasa Produs - afisare nume produs
void Produs::Eticheta()
{
    cout << "Ati selectat produsul cu numele " << nume << "\n";
}

// clasa Produs - afisare garantie
int Produs::Garantie()
{
    return 0;
}

Produs* Produs::Clona()
{
    return new Produs(*this);
}

class Aliment : public Produs
{
private:
    int expira;
public:
    Aliment();
    Aliment(const char*, double, int, const char[], int);
    Aliment(const Aliment&);
    ~Aliment();
    void Citeste(istream&);
    void Print();
    double Stoc();
    void Eticheta();
    Produs* Clona();
};
// clasa Aliment - constructor implicit
Aliment::Aliment() : Produs()
{
    expira = 0;
}

// clasa Aliment - constructor cu parametri
Aliment::Aliment(const char* n, double p, int s, const char c[10], int e) : Produs(n, p, s,
c)
{
    expira = e;
}

// clasa Aliment - constructor de copiere
```

```
Aliment::Aliment(const Aliment& a) : Produs(a)
{
    expira = a.expira;
}

// clasa Aliment - destructor
Aliment::~Aliment() {}

void Aliment::Citeste(istream& in)
{
    cout << "Tipul produsului este Aliment. Introdu datele: \n";
    Produs::Citeste(in);
    cout << "Zile pana la expirare: ";
    in >> expira;
    cout << "\n";
}

// clasa Aliment - afisare detalii aliment
void Aliment::Print()
{
    Produs::Print();
    cout << "Perioada de expirare: " << expira << "\n";
}

// clasa Aliment - afisare valoare stoc
double Aliment::Stoc()
{
    return pret * stoc;
}

// clasa Aliment - afisare nume produs
void Aliment::Eticheta()
{
    cout << "Ati selectat alimentul cu numele: " << nume << "\n";
}

Produs* Aliment::Clona()
{
    return new Aliment(*this);
}

class Electronic : public Produs
{
private:
    int garantie;
public:
    Electronic();
    Electronic(const char*, double, int, const char[], int);
    Electronic(const Electronic&);
    ~Electronic();
    void Citeste(istream&);
    void Print();
    void Eticheta();
```

```
    int Garantie();
    Produs* Clona();
};


Electronic::Electronic() : Produs()
{
    garantie = 0;
}

Electronic::Electronic(const char* n, double p, int s, const char c[10], int g) : Produs(n,
p, s, c)
{
    garantie = g;
}

Electronic::Electronic(const Electronic& e) : Produs(e)
{
    garantie = e.garantie;
}

Electronic::~Electronic() {}

void Electronic::Citeste(istream& in)
{
    cout << "Tipul produsului este Electronic. Introdu datele: \n";
    Produs::Citeste(in);
    cout << "Introdu garantia: ";
    in >> garantie;
    cout << "\n";
}

void Electronic::Print()
{
    Produs::Print();
    cout << "Perioada de garantie: " << garantie << " zile\n";
}

void Electronic::Eticheta()
{
    cout << "Ati selectat produsul electronic cu numele: " << nume << "\n";
}

int Electronic::Garantie()
{
    return garantie;
}

Produs* Electronic::Clona()
{
    return new Electronic(*this);
}
```

```
class Raft
{
private:
    int nrProd;
    Produs* produse[100];
public:
    Raft();
    Raft(const Raft&);
    ~Raft();

    Raft operator=(const Raft&);

    Raft& Adauga(Produs*);
    Produs* Elimina();
    void InfoProdRaft();
    void ContinutRaft();
};

Raft::Raft()
{
    nrProd = 0;
};

Raft::Raft(const Raft& r)
{
    nrProd = r.nrProd;
    for(int i = 0; i < r.nrProd; i++)
    {
        produse[i] = r.produse[i]->Clona();
    }
}

Raft Raft::operator=(const Raft& r)
{
    if(this != &r)
    {
        for(int i = 0; i < nrProd; i++)
        {
            delete produse[i];
        }

        nrProd = r.nrProd;
        for(int i = 0; i < r.nrProd; i++)
        {
            produse[i] = r.produse[i]->Clona();
        }
    }
    return *this;
}

Raft::~Raft()
{
    for(int i = 0; i < nrProd; i++)
```

```
    {
        delete produse[i];
    }
}

Raft& Raft::Adauga(Produs* p)
{
    if(nrProd == 100)
    {
        Produs* u = produse[0];
        for(int i = 1; i < nrProd; i++)
        {
            produse[i - 1] = produse[i];
        }
        delete u;
        nrProd--;
    }
    produse[nrProd++] = p;
    return *this;
}

Produs* Raft::Elimina()
{
    if(nrProd == 0)
    {
        return 0;
    }
    Produs* p = produse[nrProd - 1];
    nrProd--;
    return p;
}

void Raft::InfoProdRaft()
{
    for(int i = 0; i < nrProd; i++)
    {
        produse[i]->Print();
    }
}

void Raft::ContinutRaft()
{
    for(int i = 0; i < nrProd; i++)
    {
        produse[i]->Eticheta();
    }
}

int main()
{

    Produs *p1, *p2, *p3;
    p1 = new Produs();
```

```
    p2 = new Aliment();
    p3 = new Electronic();
    cin >> *p1 >> *p2 >> *p3;

    Raft r;
    r.Adauga(p1).Adauga(p2).Adauga(p3);
    cout << "Raftul contine urmatoarele produse: \n";
    r.ContinutRaft();

    cout << "--------------------\n";

    cout << "\n Informatii despre produse: \n";
    r.InfoProdRaft();

    cout << "--------------------\n";

    delete r.Elimina();
    cout << "\n Raftul dupa stergere: \n";
    r.ContinutRaft();

    cout << "--------------------\n";

    return 0;
}
```

4 Teme

Denumiți proiectele realizate de voi cu denumirea problemei și numele vostru. Exemplu: ***matrice_Bold Nicolae***.

4.1 Interfata

Lucrați la dezvoltarea unui sistem de interfață grafică pentru o aplicație software. Trebuie să implementați diferitele elemente de interfață pentru a permite utilizatorilor să interacționeze cu aplicația într-un mod intuitiv și eficient.

Pentru aceasta, trebuie să definiți o clasă de bază Element, care să reprezinte elementele generice de interfață grafică. Această clasă ar trebui să conțină informații despre poziția și dimensiunea fiecărui element.

Ulterior, trebuie să derivați două clase, Buton și Caseta, care să reprezinte butoanele și casetele de text din interfață.

În plus, trebuie să implementați o clasă Fereastra, care să reprezinte o fereastră de aplicație. Această clasă ar trebui să permită adăugarea și eliminarea de elemente de interfață grafică și să afișeze elementele prezente în fereastră.

În acest scop, se dă următoarea declarație de clase:

```
#include <iostream>

class Element {
protected:
    int x, y; // pozitia elementului
    int w, h; // dimensiunea elementului

public:
    Element();
    Element(int, int, int, int, int);
    Element(const EI&);

    virtual ~Element();

    virtual void afis();
    virtual double arie();
};

class Buton : public Element {
private:
    char* text; // Textul butonului

public:
    Buton();
    Buton(int, int, int, int, const char*);
```

```
    Buton(const Buton&);

    virtual ~Buton();

    virtual void afis();
    virtual double arie();
    char* getText();
};

class Caseta : public Element {
private:
    int lungime; // textul din caseta de text

public:
    Caseta();
    Caseta(int, int, int, int, int);
    Caseta(const Caseta&);

    virtual ~Caseta();

    virtual void afis();
    virtual double arie();
    int getLungime();
};

class Fereastra {
private:
    int count;
    Element* elemente[100];

public:
    Fereastra();
    ~Fereastra();

    Fereastra& addElement(Element*);
    Element* getElement();
    void elements();
    void infoElemente();
};
```

Folosiți modelul prezentat în problema Raft.