Laborator 11

Programarea orientată pe obiecte. Clase și obiecte. Moștenire,constructori,destructori.Funcții și clase prietene

În acest capitol sunt prezentate consideraţii teoretice și probleme rezolvate privind diferențele semnificative între limbajele C și C++, definirea și utilizarea noțiunilor specifice programării orientate pe obiecte. Este de asemenea ilustrat modul în care se definesc clasele și componentele lor, operatorul de rezoluție, constructorii și destructorii, funcțiile prietene, conceptul de moștenire, etc.

CONSIDERAŢII TEORETICE

Programarea orientata pe obiecte (POO)= o metodă de implementare în care programele sunt organizate ca și colecții de obiecte ce cooperează între ele, fiecare obiect reprezentând instanța unei clase; Fiecare clasă aparține unei ierarhii de clase, clasele fiind unite prin relații de mostenire.

Proprietățile POO sunt:

- obiectele și nu algoritmii sunt blocurile logice fundamentale;
- fiecare obiect este o instanță a unei clase;
- clasele sunt legate între ele prin relații de moștenire.

Limbaj de programare bazat pe obiecte = Un limbaj de programare care oferă suport pentru utilizarea claselor și a obiectelor .

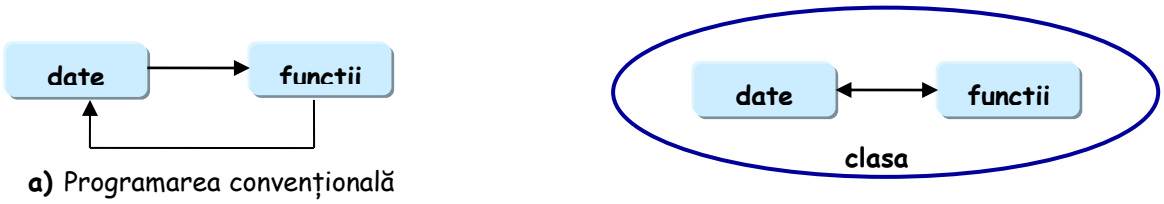
Caracteristicile POO:

- **Abstractizarea: O abstracțiune** exprimă **toate caracteristicile esențiale ale unui obiect**, care fac ca acesta să se distingă de alte obiecte;
- **Încapsularea:**este conceptul complementar abstractizării, și reprezintă **procesul de compartimentare a elementelor care formează structura și comportamentul unei abstracțiuni.** Ex.: încapsularea este un mecanism care leagă într-o variabilă numită clasa, date și funcții, restricționând accesul la acestea .
- **Modularitatea: Clasele și obiectele obținute în urma abstractizării și încapsulării** trebuie grupate și apoi stocate într-o formă fizică, denumită **modul**. Modularizarea constă în divizarea programului într-un număr de module care pot fi compilate separat, dar care sunt conectate între ele. Ex. în C++ modulele sunt fișiere ce pot fi compilate separat. Interfața modulului este plasata într-un fișier header (extensii uzuale sunt: ".h" , ".hpp", ".hh"), iar implementarea acestuia se va regăsi într-un fișier sursă (extensii uzuale sunt: ".cc", ".cpp", ".c").
- **Ierarhizarea:** reprezintă o **ordonare a abstracțiunilor**. Cele mai importante ierarhii de clase în paradigma obiectuală sunt: ierarhia de clase (relație de tip "is a") și ierarhia de obiecte (relație de tip "part of").
- **Moștenirea:** definește o **relație între clase** în care o clasa împărtășește structura și comportarea definită în una sau mai multe clase (după caz vorbim de moștenire simplă sau multiplă).

Cea mai semnificativă diferență între programarea convențională și POO este ilustrată în Fig. 1:

- în **programarea convențională** datele sunt preluate și retransmise de funcții

- în **programarea orientată pe obiecte** , clasele încapsulează datele și funcțiile într-o singură entitate



a) Programarea convențională

b) Programarea orientată pe obiecte(POO)

Fig.1. Programarea convențională - POO

Clasa = abstractizare similară cu o structură**,** utilizată pentru declararea unui nou tip de date care încapsulează date și funcții;

Obiect =exemplar (**instanță**) a unei clase

Elementele unei clase = o clasă în C++ are asociate 4 tipuri de elemente:
- colecție de **date membre** (atribute)
- colecție de **funcții membre** (metode)
- **nivele de acces** ale programului (public, private, ...)
- **nume**

La **declararea clasei**, se specifică:
- **numele clasei**,
- **lista claselor de bază** din care e derivată clasa, dacă există, și
- **membrii clasei**, atât membrii de date cât și funcții.

Implementarea clasei constă în
- **definițiile funcțiilor** componente care indică comportamentul tipului de date reprezentat de clasa respectivă. Dacă funcțiile prezente în declararea clasei sunt corect adecvate scopului propus, atunci utilizatorul nu mai are nevoie de definirea lor și implementarea se poate face într-un fișier separat (*.cpp*).

Formatul de declarare a unei **clase** care **nu moștenește altă clasă:**

class nume_clasă
{
variabile și funcții particulare
specificator de acces:
variabile și funcții
specificator de acces:
variabile și funcții
…
specificator de acces:
variabile și **funcții**
} [**Listă_de_obiecte**];

unde :
- **variabile** se declară cu tip: int,double,float,char,etc.
- **specificatori de acces** pot fi:
  - **public**: variabilele (datele) și funcțiile declarate cu acest specificator sunt vizibile (accesibile) din

orice zonă a programului.
  - **private**: variabilele și funcțiile declarate cu acest specificator pot fi accesate doar de către membrii clasei.
  - **protected**: variabilele și funcțiile declarate cu acest specificator pot fi modificate în cadrul clasei sau în clasele derivate de ea
- Funcțiile se implementează astfel:
  - **funcții inline** (în interiorul clasei)
  - **operatorul de rezoluție** ::(în afara clasei)
- **Lista_de_obiecte** este opțională, iar declararea obiectelor se poate face și astfel:
  - nume_clasă Listă_de_obiecte;

Referitor la specificatorii de acces (vizibilitate) se pot face următoarele observații:
- implicit, variabilele și funcțiile declarate într-o clasă sunt proprii (private) acelei clase și numai membrii ei au access la ele.
- specificatorii de acces au **efect** până când se întâlnește un alt specificator de acces sau se ajunge la sfârșitul declarației de clasă.
- specificatorii de acces pot **alterna** în declararea unei clase. public, private,public…

Pentru exemplificarea modului în care se utilizează clasele, se consideră exemplul definirii unui **articol tip produs** care se poate realiza fie printr-o structură fie printr-o clasă:

Ex:
a) utilizând o structură
```
struct Produs {
      char comp_id[4];   // identificator companie max. 4 caractere.
      char prod_id[8];   // identificator produs max 8 caractere
      int  pret;       // pretul produsului in USD.
      int  stoc;  // cantitate existenta in stoc
    };
```

b) utilizând o clasă:
```
class Produs {
  public:
      char comp_id[4];   // 4 char code for the manufacturer.
      char prod_id[8];   // 8-char code for the product
      int  pret;       // price of the product in dollars.
      int  stoc;  // quantity on hand in inventory
    };
```

În ce priveste definirea funcțiilor membre ale claselor acestea se pot defini fie în interior fie în exteriorul clasei.

Ex.
a) Definirea funcției membre în interiorul clasei
```
class adunare {
public:
        int suma(int a,int b) {
        int rez;
        rez=a+b;
        return rez;} };
```

b) Definirea funcției membre în exteriorul clasei

```
class adunare {
public:
        int suma(int a,int b)
};
int adunare::suma(int a,int b) {
        int rez;
        rez=a+b;        return rez;
}
```

Operatorul de rezoluție :: este un operator de specificare a domeniului și este utilizat pentru a specifica din ce domeniu (clasă) face parte o anumită funcție membră. Trebuie luat în considerare și faptul că mai multe clase diferite pot sa folosească același nume de funcție .

Accesul la componentele unei clase se realizează similar cu accesul la elementele unei structuri, prin:

- **Instanțierea** (declararea) unui obiect

  nume_clasă nume obiect;
- Accesarea componentelor clasei cu operatorul .

  nume_obiect.nume_funcție;
- Accesarea componentelor clasei prin pointeri cu operatorul "sageată" ->

  nume_pointer->nume_funcție;

Noțiunea de **moștenire** a fost introdusă în C++ pentru a permite construirea unei ierarhii de clase. Procesul de ierarhizare constă în :

- crearea unei **clase de baza (parinte)=** cea mai generală descriere care stabilește calitățile comune ale tuturor obiectelor ce vor deriva din această bază
- crearea **claselor derivate (copii)** din clasa de bază, care vor include :
  - toate **caracteristicile clasei de bază** și în plus
  - **calități proprii** clasei respective.

Forma generală de declarare a claselor derivate prin moștenire este :

```
class nume_nou_clasă: [specificator_acces] clasa_moștenită
 {
 //corpul noii clase
 }
```

Modul de acces corespunzător fiecărui specificator de acces este prezentat în tabelul 1.

Tabelul 1

Specificator acces	Acces permis pentru
public	Toate celelalte clase
private	Nici o altă clasă
protected	Numai clasele derivate

Funcție constructor = funcție specială care este membru al unei clase și **are același nume cu acea clasă,** nu poate să returneze valori și nu conțin nici tipul de returnat

Un **constructor al unui obiect** este apelat atunci când este creat acel obiect. Un **destructor al unui obiect** va produce distrugerea obiectului creat (pentru eliberarea de memorie, etc.)

Ordinea de apelare a constructorilor și destructorilor

- Constructori: clasă bază ⇒clasă derivat1 ⇒clasă derivat2
- Destructori: clasă derivat2 ⇒clasă derivat1 ⇒clasă bază

Ex. : Utilizarea constructorului și destructorului unei stive:

```
class stiva {
        int stiv[SIZE];
        int vis;
public:
         stiva() ;//constructor
        ~stiva() ;//Destructor
        void pune(int i);
        int scoate();
};
stiva::stiva()    //functia constructor pentru stiva
{  vis=0;  cout << "Stiva initializata\n" ;  }
stiva::~stiva()          //functia destructor pentru stiva
{  cout << "Stiva distrusa!\n";  }
```

Funcție prietenă (friend) = funcție care are acces la membrii private și protected ai clasei căreia ii este prietenă.

Format de declarare:

```
class nume_clasa {
        variabile și funcții private
public:
        friend prototip_fctie();
        variabile și funcții publice;
};
```

Funcțiile prietene sunt utile la:

- supraîncărcarea operatorilor
- simplifică creearea funcțiilor de I/O
- scrierea mai eficientă a programelor

Clasa prietenă (friend) = clasa care are acces la variabilele particulare (nume de tipuri și enumerări de constante) definite în cadrul celeilalte clase.

PROBLEME REZOLVATE

Ex.1: Programul în C++ definește o clasă de bază numită clădire cu

- **datele membre:** camere, etaje și supraf și
- **funcțiile membre:** nr_camere(),nr_etaje(), cate_etaje(), ce_supraf(), cat_supraf()

și 2 clase derivate, una numită casa cu

- **datele membre:** dormit, bai și
- **funcțiile membre:** nr_dormit(), cate_dormit(), nr_bai(), cate_bai() și alta numită scoala cu
- **datele membre:** saliclasa, lab și

- **funcțiile membre:** nr_saliclasa(), cate_saliclasa(), nr_lab(), cate_lab();

Se cere să se declare obiecte de tipul casă și școală și utilizând funcțiile membre să se afișeze diferite valori ale datelor membre.

Varianta in C++

```
#include <iostream>
using namespace std;
class cladire {
        int camere;
        int etaje;
        int supraf;
public:
        void nr_camere(int num);
        int cate_camere();
        void nr_etaje(int num);
        int cate_etaje();
        void ce_supraf(int num);
        int cat_supraf();
};
//clasa casa derivata din cladire
class casa: public cladire {
        int dormit;
        int bai;
public:
        void nr_dormit(int num);
        int cate_dormit();
        void nr_bai(int num);
        int cate_bai();
};
//clasa scoala derivata din cladire
class scoala: public cladire {
        int saliclasa;
        int lab;
public:
        void nr_saliclasa(int num);
        int cate_saliclasa();
        void nr_lab(int num);
        int cate_lab();
};
//definirea functiilor din clasa de baza
void cladire::nr_camere(int num)
{camere=num;}
void cladire::nr_etaje(int num)
{etaje=num;}
void cladire::ce_supraf(int num)
{supraf=num;}
int cladire::cate_camere()
{return camere;}
int cladire::cate_etaje()
{return etaje ;}
int cladire::cat_supraf()
{return supraf ; }
//definirea functiilor din clasa derivata casa
void casa::nr_dormit (int num)
{dormit=num; }
```

```
void casa::nr_bai (int num)
{bai=num; }
int casa::cate_dormit ()
{return dormit; }
int casa::cate_bai ()
{return bai; }
//definirea functiilor din clasa derivata scoala
void scoala::nr_saliclasa(int num)
{saliclasa=num; }
void scoala::nr_lab(int num)
{lab=num;}
int scoala::cate_saliclasa()
{return saliclasa; }
int scoala::cate_lab()
{return lab; }
//functia main()
int main(void)
{casa c; //obiect de tip casa
scoala s; //obiect de tip scoala
c.nr_camere(12);    c.nr_etaje(3);
c.ce_supraf(4500); c.nr_dormit(5);  c.nr_bai(3);
cout << "Casa are suprafata:" <<c.cat_supraf() <<" mp," << c.cate_etaje() << " etaje, si "   ;
cout << c.cate_camere() << " camere, din care:\n";
cout << "- dormitoare: " << c.cate_dormit() <<endl;
cout << "- camere diferite de dormitoare: " << c.cate_camere() - c.cate_dormit() <<endl;
cout << "- bai: " << c.cate_bai() <<endl;
s.nr_camere(200);   s.nr_etaje(3);
s.nr_saliclasa(180); s.nr_lab(5);  s.ce_supraf(25000);
cout << "\nScoala are suprafata: " << s.cat_supraf() <<" mp," ;
cout<<s.cate_etaje()<<"etaje, si"<<s.cate_camere() << "incaperi, din care:\n"; //atentie linie continuata!
cout << "-sali de clasa: " << s.cate_saliclasa() << endl;
cout << "-laboratoare: " << s.cate_lab() <<endl;
cout << "-alte sali(magazii, cancelarie, etc) : " << s.cate_camere() - s.cate_saliclasa()- s.cate_lab() <<endl;
//atentie linie continuata!
return 0;}
```

Rezultate:

```
Casa are suprafata:4500 mp,3 etaje, si 12 camere, din care:
- dormitoare: 5
- camere diferite de dormitoare: 7
- bai: 3

Scoala are suprafata: 25000 mp,3etaje, si200incaperi, din care:
-sali de clasa: 180
-laboratoare: 5
-alte sali(magazii, cancelarie, etc) : 15
```

Aplicație:

Să se modifice programul astfel încât să se declare și alte obiecte de tipul casă și școală și utilizând funcțiile membre să se afișeze diferite valori ale datelor membre.

Ex. 2. Programul C++ implementează clasa student cu următoarele funcții membru:

- ***funcție prin care să se citească numele, prenumele și 5 note obținute la o sesiune de examene de către fiecare student;***
- ***funcție pentru afișare;***
- ***funcție pentru calculul mediei de promovare.***

Varianta in C++

```
#include <iostream>
#include<conio.h>
#define N 25
using namespace std;
class student{
        char nume[25];
        char prenume[30];
        int nota[5];
public:
        void id_student();
        void afis();
        double media();
};
int n;
void student::id_student(){
        cout<<"\nNume student:";       cin>>nume;
        cout<<"Prenume student:";      cin>>prenume;
        for(int k=1;k<=5;k++){
                cout<<"Nota"<<k<<":";  cin>>nota[k];}}
void student::afis(){
        cout<<"\nNume student:";
        cout<<nume;
        cout<<"\nPrenume student:";
        cout<<prenume<<"\n";
        for(int k=1;k<=5;k++){
                cout<<"Nota:"<<k <<":" <<nota[k] <<endl;        }
        cout<<"Media:"<<media() << endl;}
double student::media(){
        double m=.0;
        for(int k=1;k<=5;k++)
                if(nota[k]<5){
                        cout<<"\n Student restantier"; return 0;}
                else m+=nota[k];          return m/5;}
int main(){
        student s[N];
        cout<<"\n Numarul de studenti:";
        cin>>n;
        for(int i=1;i<=n;i++){
                s[i].id_student();        s[i].afis();}
  return 0;}
```

Rezultate:

```
 Numarul de studenti:

Nume student:Popa
Prenume student:Petre
Nota1:10
Nota2:9
Nota3:5
Nota4:8
Nota5:7

Nume student:Popa
Prenume student:Petre
Nota:1:10
Nota:2:9
Nota:3:5
Nota:4:8
Nota:5:7
Media:7.8
```

Aplicație:

Să se modifice programul astfel încât să calculeze și afișeze nr. studenților care au media peste 8.50 și numele acestora

Ex.3. Programul C++ implementează tipul de date complex utilizând clase și realizează calculul sumei a n numere complexe.

Varianta in C++

```
#include <iostream>
using namespace std;
int n;
class complex{
        double re[20],im[20];
public:
        double cit();
        void afis();
        double suma_1();
        double suma_2();
};
double complex::cit(){
        cout<<"n=";
        cin>>n;
        for(int i=1;i<=n;i++){
                cout<<"Partea reala a numarului ";
                cout<<"complex"<<" "<<i <<": ";
                cin>>re[i];
                cout<<"Partea imaginara a numarului ";
                cout<<"complex"<<" "<<i << ": ";
                cin>>im[i];
        }
        return 1;
}
double complex::suma_1(){
        double s1=0;
        for(int i=1;i<=n;i++)
                s1+=re[i];
        return s1;}
double complex::suma_2(){
        double s2=0;
        for(int i=1;i<=n;i++)
                s2+=im[i];
        return s2;}
void complex::afis(){
        cout<<"\n Suma celor"<<" "<<n <<" numere complexe:";
        if(suma_2()>0)
                cout<<suma_1()<<"+"<<suma_2()<<"*i";
        else cout<<suma_1()<<suma_2()<<"*i" << endl;}
int main(){
complex c;
c.cit();   c.afis(); cout<<endl;
return 0; }
```

Rezultate:

```
n=2
Partea reala a numarului complex 1: 5.5
Partea imaginara a numarului complex 1: 2.3
Partea reala a numarului complex 2: 1.5
Partea imaginara a numarului complex 2: 4.8

 Suma celor 2 numere complexe:7+7.1*i
```

Aplicație:

Să se modifice programul astfel încât să calculeze și afișeze modulul fiecărui număr complex (valoarea absolută)

Ex.4. Programul C++ calculeaza si afiseaza rezistenta echivalenta serie a unui circuit electric.

Varianta in C++

```
#include <iostream>
#include <vector>
using namespace std;
// Define a class representing a resistor
class Resistor {
private:
   double resistance; // Resistance value
public:
   // Constructor to initialize the resistance value
   Resistor(double resistance) : resistance(resistance) {}
   // Get method to retrieve the resistance value
   double getResistance() const {
      return resistance;    }
};
// Define a class representing an electric circuit
class Circuit {
private:
   vector<Resistor> resistors; // Vector to store resistors in the circuit
public:
   // Method to add a resistor to the circuit
   void addResistor(const Resistor& resistor) {
      resistors.push_back(resistor);     }
   // Method to calculate the total resistance of the circuit
   double getTotalResistance() const {
      double totalResistance = 0.0;
      // Iterate through each resistor and sum up their resistances
      for (const auto& resistor : resistors) {
         totalResistance += resistor.getResistance();       }
      return totalResistance; // Return the total resistance
   }
};
int main() {
   Circuit circuit; // Create an instance of the Circuit class
   // Input resistor values from the keyboard
   int numResistors;
   cout << "Enter the number of resistors: ";
   cin >> numResistors; // Read the number of resistors from the user
   cout << "Enter resistance values (in ohms):" << endl;
   for (int i = 0; i < numResistors; ++i) {
      double resistance;
      cout << "Resistor " << i + 1 << ": ";
      cin >> resistance; // Read resistance value from the user
      circuit.addResistor(Resistor(resistance)); // Add the resistor to the circuit
   }
   // Calculating and printing the total resistance of the circuit
   cout << "Total resistance of the circuit: " << circuit.getTotalResistance() << " ohms" << endl;
   return 0; // Exit the program
}
```

Rezultate:

```
Enter the number of resistors: 4
Enter resistance values (in ohms):
Resistor 1: 50
Resistor 2: 10
Resistor 3: 30
Resistor 4: 20
Total resistance of the circuit: 110 ohms
```

Aplicație:

Să se modifice programul astfel încât sa se calculeze rezistenta echivalenta pentru conexiunea in paralel.

Ex.5. Programul C++ calculeaza si afiseaza consumul de energie electrica intr-o casa

Varianta in C++

```
#include <iostream>
using namespace std;
// Define a class representing a device
class Device {
private:
   string name;     // Name of the device
   double power;    // Power consumption of the device in watts
   double time;     // Operating time of the device in hours
public:
   // Constructor to initialize device properties
   Device(string name, double power, double time) : name(name), power(power), time(time) {}
   // Method to calculate energy consumption
   double calculateEnergyConsumption() const {
      return power * time; // Energy consumption = power * time    }
   // Method to get the name of the device
   string getName() const {
      return name;    }
};
int main() {
   // Create instances of three different devices
   Device device1("TV 1", 100, 8); // Device 1 consumes 100 watts and operates for 8 hours
   Device device2("TV2 2", 150, 6); // Device 2 consumes 150 watts and operates for 6 hours
   Device device3("Refrigerator", 200, 4); // Device 3 consumes 200 watts and operates for 4 hours
   Device device4("Induction hob", 200, 2); // Device 4 consumes 200 watts and operates for 2 hours
   // Calculate energy consumption for each device
   double energy1 = device1.calculateEnergyConsumption();
   double energy2 = device2.calculateEnergyConsumption();
   double energy3 = device3.calculateEnergyConsumption();
   double energy4 = device4.calculateEnergyConsumption();
   // Print energy consumption for each device
   cout << "Energy consumption for " << device1.getName() << ": " << energy1 << " watt-hours" << endl;
   cout << "Energy consumption for " << device2.getName() << ": " << energy2 << " watt-hours" << endl;
   cout << "Energy consumption for " << device3.getName() << ": " << energy3 << " watt-hours" << endl;
   cout << "Energy consumption for " << device4.getName() << ": " << energy4 << " watt-hours" << endl;
   // Calculate and print total energy consumption
   double totalEnergy = energy1 + energy2 + energy3+ energy4;
   cout << "Total energy consumption for all devices: " << totalEnergy << " watt-hours" << endl;
   return 0;}
```

Rezultate:

```
Energy consumption for TV 1: 800 watt-hours
Energy consumption for TV2 2: 900 watt-hours
Energy consumption for Refrigerator: 800 watt-hours
Energy consumption for Induction hob: 400 watt-hours
Total energy consumption for all devices: 2900 watt-hours
```

Aplicație:

Să se modifice programul astfel încât sa se citeasca de la tastatura valorile puterii si timpului de utilizare pentru fiecare aparat si sa se calculeze si afiseze consumul fiecarui aparat si consumul total.

Ex.6. Programul C++ calculeazasi afiseaza impedanta totala a unui circuit RLC. Explicatii implementare:

Ierarhia componentelor:

- clasa Component= este o clasă de bază abstractă care reprezintă componentele electrice, care are o funcție virtuală impedance() care trebuie implementată la clasele derivate.
- Clasele Resistor, capacitor și inductor= clase derivate care reprezintă tipuri specifice de componente. Ele suprascriu funcția impedance() pentru a calcula impedanța pe baza proprietăților lor (R, L sau C).
- Clasa Circuit = reprezinta o colectie de componente. Are ca element de tip date un vector care stocheaza pointeri la obiectele componente , o metoda addComponent() care adauga o componenta in circuit si o metoda totalImpedance() care calculeaza impedanta totala a circuitului pentru o frecventa data prin insumarea tuturor impedantelor componentelor calculate la acea frecventa.
- In functia main(), se creeaza un obiect numit Circuit.Se adauga componente (resistor, capacitor, inductor) in circuit utilizand metoda addComponent() .Impedanta totala a circuitului la o frecventa specificata (ex. 1000 Hz) este calculata utilizand metoda totalImpedance().

Varianta in C++

```
#include <iostream>
#include <vector>
#include <cmath>
// Define M_PI if not provided by the compiler
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif
using namespace std;
// Clasa de baza pentru componente
class Component {
public:
   virtual double impedance(double frequency) const = 0; };
// Resistor class
class Resistor : public Component {
private:
   double resistance;
public:
   Resistor(double resistance) : resistance(resistance) {}
```

```
    virtual double impedance(double frequency) const override {
        return resistance;    }
};
// Capacitor class
class Capacitor : public Component {
private:
    double capacitance;
public:
    Capacitor(double capacitance) : capacitance(capacitance) {}
    virtual double impedance(double frequency) const override {
        return 1.0 / (2 * M_PI * frequency * capacitance);    }
};
// Inductor class
class Inductor : public Component {
private:
    double inductance;
public:
    Inductor(double inductance) : inductance(inductance) {}
    virtual double impedance(double frequency) const override {
        return 2 * M_PI * frequency * inductance;    }
};
// Circuit class
class Circuit {
private:
    std::vector<Component*> components;
public:
    void addComponent(Component* component) {
        components.push_back(component);
    }
    double totalImpedance(double frequency) const {
        double totalImpedance = 0.0;
        for (const auto& component : components) {
            totalImpedance += component->impedance(frequency);
        }
        return totalImpedance;    }
};
int main() {
    // Create a circuit
    Circuit circuit;
    // Add components to the circuit
    circuit.addComponent(new Resistor(100));  // 100 ohm resistor
    circuit.addComponent(new Capacitor(0.001));  // 1 mF capacitor
    circuit.addComponent(new Inductor(0.1));  // 0.1 H inductor
    // Calculate total impedance at a given frequency
    double frequency = 1000;  // 1000 Hz
    double totalImpedance = circuit.totalImpedance(frequency);
    cout << "Total impedance of the circuit at " << frequency << " Hz: " << totalImpedance << " ohms" << endl;
    return 0;}
```

Rezultate:

```
Total impedance of the circuit at 1000 Hz: 728.478 ohms
```

Aplicație:

Să se modifice programul astfel încât sa se adauge inca 2 componente rezistive si 2 inductive

PROBLEME PROPUSE

1. ***Sa se scrie un program C++ in care se citesc valorile a 3 rezistente intr.-un circuit si se calculeaza valoarea curentului si caderea de tensiune in circuit utilizand legile lui Kirckhoff***

$$(\mathrm{I}) = \frac{1}{\frac{1}{R_1}+\frac{1}{R_2}+\frac{1}{R_3}} \quad (\mathrm{V}) = \sum_{i=1}^{3} I \cdot R_i$$

Clasele se pot implementa astfel:

```
class Resistor {
private:
   double resistance; // Resistance value in ohms
public:
   // Constructor
   Resistor(double resistance) : resistance(resistance) {}
   // Method to get the resistance value
   double getResistance() const {
      return resistance;
   }
};

class KirchhoffCircuit {
private:
   vector<Resistor> resistors; // Vector of resistors in the circuit
public:
   // Method to add a resistor to the circuit
   void addResistor(const Resistor& resistor) {
      resistors.push_back(resistor);
   }
   // Method to apply Kirchhoff's current law (KCL)
   double applyKCL() const {
      double totalCurrent = 0.0;
      for (const auto& resistor : resistors) {
         double resistance = resistor.getResistance();
         totalCurrent += 1.0 / resistance; // Sum of (1/R) for each resistor
      }
      return 1.0 / totalCurrent; // Total current = 1 / (Sum of (1/R))
   }
   // Method to apply Kirchhoff's voltage law (KVL)
   double applyKVL(double voltageSource) const {
      double totalVoltageDrop = 0.0;
      for (const auto& resistor : resistors) {
         double resistance = resistor.getResistance();
         totalVoltageDrop += resistance * applyKCL(); // V = IR
      }
      return voltageSource - totalVoltageDrop; // Voltage across the source - total voltage drop
   }
};
```

2. ***Să se scrie un program în C++ care realizează un program de gestiune a unui magazin de echipamente electronice, pentru care se introduc de la tastatură datele: denumirea, cod, cantitate și preț și se cere ordonarea produselor după fiecare din aceste date.***

3. ***Să se implementeze o clasă numită Vector utilă pentru operaţiile cu tablourile de numere întregi, având ca date membru un tablou de numere întregi (int elem[20];) şi numărul efectiv de elemente ale acestuia (int n;). Să se includă în clasă metode corespunzătoare pentru citirea unui vector de la tastatură, înmulţirea tuturor elementelor cu un scalar, adăugarea unui nou element în vector, eliminarea unui element din vector şi afişarea respectivului vector. În exteriorul clasei, să se definească o funcţie de adunare a două elemente de tip Vector.***