PROGRAMARE ORIENTATĂ PE OBIECTE

LABORATOR 1 - PARTEA I

Anul universitar 2023 – 2024

Semestrul II

1 Noțiuni teoretice

1.1 Conceptul de clasă

Clasele sunt o extindere a conceptului de *structuri de date*: la fel ca structurile de date, clasele conțin membri, dar ele mai conțin și funcții ca membri.

C++ este un limbaj de programare orientat pe obiecte.

Totul în C++ este asociat cu clase și obiecte, împreună cu atributele și metodele sale. De exemplu: în viața reală, o mașină este un obiect. Mașina are atribute, cum ar fi greutatea și culoarea, și metode, cum ar fi conducerea și frâna.

Atributele și metodele sunt practic variabile și funcții care aparțin clasei. Aceștia sunt adesea denumiți „membrii clasei”.

O clasă este un tip de date definit de utilizator pe care îl putem folosi în programul nostru și funcționează ca un constructor de obiecte sau un „plan” pentru crearea obiectelor.

Clasele reprezintă tipuri de date definite de utilizator sau deja existente în sistem. O clasă poate conține:

- membri - variabile membru (câmpuri) și proprietăți, care definesc starea obiectului
- metode – funcții membru, ce reprezintă operații asupra stării.

Forma generală a unei clase este:

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

Dacă vom implementa în C++, va avea forma aceasta:

```
class ClasaMea {       // Clasa
  public:             // Specificatorul de acces
    int m1;        // membru 1
    string m2;  // membru 2
};
```

Clasele se definesc cu ajutorul cuvântului cheie `class` sau folosind cuvântul cheie `struct`, cu următoarea sintaxă:

```
class nume_clasă {
  specificator_acces_1:
    membru1;
  specificator_acces_2:
    membru2;
  ...
} denumiri_obiecte;
```

unde nume_clasă este un identificator valid pentru clasa respectivă, denumiri_obiecte este o listă opțională cu identificatorii obiectelor aparținând acelei clase. Corpul definiției poate conține membri (care pot fi date sau funcții) și, opțional, specificatori de acces.

```
// exemplu clase
#include <iostream>
using namespace std;

class Dreptunghi {
    int latime, inaltime;
  public:
    void seteaza_valori (int,int);
    int aria() {return latime*inaltime;}
};

void Dreptunghi::seteaza_valori (int x, int y) {
  latime = x;
  inaltime = y;
}

int main () {
  Dreptunghi drept;
  drept.seteaza_valori (3,4);
  cout << "aria: " << drept.aria();
  return 0;
}
```

Operatorul scope (::) precizează clasa căreia îi aparține membrul ce urmează a fi definit, acordându-i exact același domeniu ca în cazul în care funcția ar fi fost definită direct în interiorul definiției clasei. De exemplu, funcția seteaza_valori din exemplul anterior are acces la variabilele latime și inaltime, care sunt membri privati ai clasei Dreptunghi, fiind accesibili doar de către alți membri ai clasei precum acesta.

Singura diferență dintre definirea unei funcții membru complet în interiorul definiției clasei sau doar includerea declarației si definirea ulterioară în afara clasei constă în faptul că în primul caz funcția este

considerată automat de către compilator ca funcție membru *inline* , în timp ce în a doua situație este o funcție membru obișnuită (not-inline). Aceasta nu are consecințe în comportament, ci poate numai posibile optimizări de compilare.

1.2 Încapsularea claselor

Încapsularea datelor este un concept cheie în programarea orientată pe obiecte care se referă la ambalarea datelor (variabilele membru) și a funcțiilor (metodele) care operează pe aceste date într-o singură unitate logică, numită clasă. Ideea principală a încapsulării este ascunderea detaliilor de implementare și protejarea datelor de accesul direct și modificarea neautorizată din afara clasei.

Principalele aspecte ale încapsulării datelor includ:

- Accesul controlat la date: Încapsularea permite definirea membrilor unei clase ca fiind privați, protejând astfel datele de modificări accidentale sau neașteptate din alte părți ale programului. Accesul la aceste date este controlat prin intermediul metodelor publice ale clasei, care sunt utilizate pentru a obține și a modifica valorile acestor date.
- Ascunderea detaliilor de implementare: Încapsularea permite ascunderea detaliilor de implementare ale unei clase și furnizează o interfață simplificată și coerentă pentru utilizatorii clasei. Utilizatorii nu trebuie să cunoască cum sunt stocate și gestionate datele intern în clasa; ei doar utilizează metodele publice pentru a interacționa cu obiectele clasei.
- Modularitate și abstractizare: Încapsularea facilitează crearea de module independente și reutilizabile în cadrul unui program, deoarece clasele pot fi concepute și implementate separat și apoi pot fi utilizate ca și componente în alte părți ale aplicației. În plus, încapsularea permite abstractizarea funcționalității, permițând programatorilor să se concentreze pe utilizarea obiectelor și a interfețelor definite de clasă, fără a fi nevoie să cunoască detaliile interne.

În esență, încapsularea datelor încurajează organizarea și structurarea codului într-un mod care să fie ușor de înțeles, de întreținut și de extins, și promovează concepte cheie ale programării orientate pe obiecte, cum ar fi abstracția, modularitatea și reutilizarea codului.

Sensul încapsulării este de a vă asigura că datele „sensibile” sunt ascunse de utilizatori. Pentru a realiza acest lucru, trebuie să declarați variabilele/atributele clasei ca fiind private (nu pot fi accesate din afara clasei). Dacă doriți ca alții să citească sau să modifice valoarea unui membru privat, puteți furniza metode publice de obținere și setare.

1.3 Constructori

1.3.1 Definire

În programarea orientată pe obiecte, un constructor este o metodă specială a unei clase care este automat apelată atunci când este creat un nou obiect al acelei clase. Rolul principal al constructorului este de a

inițializa starea inițială a unui obiect și de a efectua orice alte operațiuni necesare pentru a pregăti obiectul pentru utilizare.

Câteva caracteristici importante ale constructorilor includ:

- Nume identic cu clasa: Constructorul are același nume cu clasa în care este definit și nu returnează nicio valoare.
- Apel automat: Constructorul este apelat automat atunci când se creează un nou obiect al clasei respective. Nu poate fi apelat explicit de către programator.
- Inițializare: Constructorul poate fi folosit pentru a inițializa membrii datelor și pentru a face alte inițializări necesare pentru obiect. Acest lucru poate include alocarea de memorie, inițializarea variabilelor și configurarea altor stări ale obiectului.

În C++, există mai multe tipuri de constructori pe care îi puteți defini într-o clasă, în funcție de nevoile și specificațiile dvs. Iată câteva tipuri comune de constructori:

1. **Constructor implicit**:

- Este constructorul care nu are niciun parametru.
- Este folosit pentru a crea un obiect fără a specifica explicit valori inițiale pentru membrii săi.
- Dacă nu există niciun constructor definit în clasă, compilatorul va furniza implicit un constructor implicit.

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

class MyClass {
public:
    int value;

    // Constructorul implicit
    MyClass() {
        cout << "Constructor implicit apelat." << endl;
        value = 0;
    }
};

int main() {
    // Crearea unui obiect folosind constructorul implicit
    MyClass obj;
    cout << "Valoare: " << obj.value << endl;

    return 0;
```

```
}
```

2. **Constructor cu parametri**:

- Este un constructor care primește unul sau mai mulți parametri.
- Este utilizat pentru a inițializa membrii unei clase cu valori specificate în momentul creării obiectului.
- Poate fi utilizat pentru a impune inițializarea membrilor obligatorii și pentru a oferi flexibilitate în crearea obiectelor.

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

class Point {
public:
    int x, y;

    // Constructor cu parametri
    Point(int xCoord, int yCoord) {
        cout << "Constructor cu parametri apelat." << endl;
        x = xCoord;
        y = yCoord;
    }
};

int main() {
    // Crearea unui obiect folosind constructorul cu parametri
    Point p1(3, 5);
    cout << "Coordonate: (" << p1.x << ", " << p1.y << ")" << endl;

    return 0;
}
```

3. **Constructor de copiere**:

- Este un constructor care primește un alt obiect al aceleiași clase ca și parametru.
- Este folosit pentru a inițializa un obiect nou folosind datele unui alt obiect deja existent.
- Este invocat automat atunci când un obiect este inițializat cu un alt obiect al aceleiași clase.

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

class MyClass {
public:
    int value;
```

```
    // Constructor de copiere
    MyClass(const MyClass& other) {
        cout << "Constructor de copiere apelat." << endl;
        value = other.value;
    }
};

int main() {
    // Crearea unui obiect original
    MyClass obj1;
    obj1.value = 5;

    // Crearea unui nou obiect folosind constructorul de copiere
    MyClass obj2 = obj1;
    cout << "Valoare obiect 2: " << obj2.value << endl;

    return 0;
}
```

4. **Constructor de deplasare (move constructor)**:

- Este un constructor special care transferă resursele unui obiect temporar în alt obiect.
- Este folosit pentru a eficientiza operațiile de deplasare a resurselor între obiecte.
- Este invocat automat atunci când se fac operații care implică deplasarea unui obiect temporar, cum ar fi returnarea unui obiect dintr-o funcție.

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

class Resursa {
private:
    int* date; // Datele pe care le deține clasa (de exemplu, o zonă de memorie alocată
dinamic)

public:
    // Constructorul de deplasare
    Resursa(Resursa&& other) noexcept : date(other.date) {
        other.date = nullptr; // Setam pointerul altui obiect la nullptr pentru a preveni
accesarea datelor duplicate
    }

    // Constructorul care initializeaza resursa
    Resursa(int val) : date(new int(val)) {}

    // Destructorul pentru a elibera memoria alocată dinamic
    ~Resursa() {
        delete date;
    }
```

```
    // Metoda pentru a obține valoarea resursei
    int getValue() const {
        return *date;
    }
};

int main() {
    // Crearea unui obiect temporar
    Resursa temp(10);

    // Utilizarea constructorului de deplasare pentru a transfera resursele de la obiectul
temporar la obiectul final
    Resursa final(move(temp));

    // Verificam ca obiectul final a preluat resursele si ca obiectul temporar a fost
eliberat
    cout << "Valoarea resursei in obiectul final: " << final.getValue() << endl;

    return 0;
}
```

5. **Constructor explicit**:

- Este un constructor care împiedică conversiile implicite și coerciția tipurilor în cazul în care este folosit pentru a inițializa un obiect.
- Este utilizat pentru a preveni ambiguitățile și erorile subtile cauzate de conversiile implicite nedorite.
- Se marchează prin utilizarea cuvântului cheie explicit înaintea declarației constructorului.

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

class MyClass {
public:
    int value;

    // Constructor explicit
    explicit MyClass(int v) {
        cout << "Constructor explicit apelat." << endl;
        value = v;
    }
};

int main() {
    // Crearea unui obiect folosind constructorul explicit
    MyClass obj(10);
    cout << "Valoare obiect: " << obj.value << endl;
```

```
    // Nu puteti crea un obiect implicit folosind constructorul explicit
    // MyClass obj2 = 20; // Eroare de compilare

    return 0;
}
```

Acestea sunt câteva tipuri de constructori pe care le puteți defini într-o clasă în C++.

1.3.2 Inițializare directă

Când se folosește un constructor pentru a inițializa alți membri, aceștia pot fi inițializați direct, fără a mai recurge la instrucțiuni în corpul lor. Acest lucru se poate face inserând, înaintea corpului cosntructorului, simbolul două puncte (:) și lista de inițializări pentru membrii clasei. De exemplu, să considerăm o clasă cu următoarea declarație:

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

Constructorul pentru aceasta clasă ar putea fi definit, ca de obicei, astfel:

```
Dreptunghi::Dreptunghi (int x, int y) { latime=x; inaltime=y; }
```

Dar se poate defini, de asemenea, folosind inițializarea membrilor astfel:

```
Dreptunghi::Dreptunghi (int x, int y) : latime(x) { inaltime=y; }
```

Sau chiar:

```
Dreptunghi::Dreptunghi (int x, int y) : latime(x), inaltime(y) { }
```

Să observăm că în acest ultim caz constructorul nu face altceva decât să inițializeze membrii, deci corpul funcției este vid.

1.3.3 Supraîncărcare

Ca orice altă funcție, un constructor poate fi supraîncărcat cu diferite versiuni de parametri: număr diferit de parametri și/sau tipuri de date diferite pentru parametri. Compilatorul îl va apela automat pe cel al căror parametri se potrivesc cu argumentele transmise:

```
// supraîncărcarea constructorilor de clasă
#include <iostream>
using namespace std;

class Dreptunghi {
    int latime, inaltime;
  public:
```

```
    Dreptunghi ();
    Dreptunghi (int,int);
    int aria (void) {return (latime*inaltime);}
};

Dreptunghi::Dreptunghi () {
  latime = 5;
  inaltime = 5;
}

Dreptunghi::Dreptunghi (int a, int b) {
  latime = a;
  inaltime = b;
}

int main () {
  Dreptunghi drept (3,4);
  Dreptunghi drepb;
  cout << "aria lui drept: " << drept.aria() << endl;
  cout << "aria lui dreptb: " << dreptb.aria() << endl;
  return 0;
}
```

1.4 Destructori

Un destructor este o metodă specială în C++ care este automat apelată atunci când un obiect este distrus sau când scopul său de viață se termină. Rolul principal al destructorului este de a elibera resursele pe care obiectul le-a alocat sau a efectuat alte operații de curățare necesare pentru a pregăti obiectul pentru eliminare din memorie.

Caracteristicile cheie ale destructorilor includ:

- Nume identic cu clasa: Similar cu constructorul, destructorul are același nume cu clasa în care este definit, precedat de caracterul tilde (~).
- Apel automat: Destructorul este apelat automat atunci când obiectul este eliminat din memorie. Acest lucru poate fi în momentul în care obiectul este destructurat din cadrul unei funcții, când un obiect local iese din domeniul său de aplicare, sau atunci când este eliberat din memoria dinamică folosind operatorul delete.
- Curățare și eliberare de resurse: Destructorul este folosit pentru a curăța și elibera resursele pe care obiectul le-a alocat. Acest lucru poate include dealocarea memoriei, închiderea de fișiere sau conexiuni la baze de date și alte operațiuni de curățare necesare pentru a evita scurgerile de memorie sau alte probleme de gestionare a resurselor.

Destructorii îndeplinesc functionalitatea inversă a *constructorilor*: eliberează resursele alocate printr-o clasă atunci când aceasta nu mai este necesară. Clasele pe care le-am definit în capitolele anterioare nu alocă niciun fel de resurse și de aceea nu avem ce să eliberăm.

Acum, să ne imaginăm că în clasa din ultimul exemplu se alocă dinamic memorie care să rețină stringul conținut ca dată membru; în acest caz, ar fi foarte utilă o funcție care să fie apelată automat la sfârșitul ciclului de viață al obiectului și care să elibereze memoria alocată. Pentru a face aceasta, folosim un *destructor*. Un destructor este o funcție membru foarte asemănătoare unui *constructor implicit*: nu are argumente și nu returnează nimic, nici chiar `void`. De asemenea, are ca nume chiar numele clasei, dar precedat de un simbol tilda (~):

```
class Exemplu4 {
    string* ptr;
  public:
    // constructori:
    Exemplu4() : ptr(new string) {}
    Exemplu4 (const string& str) : ptr(new string(str)) {}
    // destructor:
    ~Exemplu4 () {delete ptr;}
    // accesarea continutului:
    const string& continut() const {return *ptr;}
};
```

1.5 Obiecte și construirea lor

În programarea orientată pe obiecte, obiectele sunt instanțe ale claselor. Procesul de creare a unui obiect este cunoscut sub numele de construire a obiectului și implică inițializarea acestuia cu ajutorul constructorului clasei corespunzătoare.

Iată câțiva pași principali implicați în construirea unui obiect în C++:

1. Definirea clasei: Începeți prin definirea clasei din care va fi creat obiectul. O clasă este un șablon pentru crearea de obiecte care conține membri de date (variabile) și metode (funcții) care operează pe aceste date.

```
class MyClass {
private:
    int x;
    double y;

public:
    MyClass(int xValue, double yValue) : x(xValue), y(yValue) {}
};
```

2. Apelul constructorului: Utilizați constructorul clasei pentru a crea un obiect. Constructorul poate fi apelat în mod explicit folosind sintaxa ClassName objName(constructorArgs); sau implicit prin instanțierea obiectului fără nume de constructor.

```
MyClass obj1(10, 3.14); // Apel explicit al constructorului
MyClass obj2 = MyClass(20, 6.28); // Apel explicit al constructorului
MyClass obj3 = {30, 9.42}; // Inițializare uniformă (implicită) - apel implicit al
constructorului
```

3. Inițializarea membrilor: Constructorul este responsabil pentru inițializarea membrilor clasei. Aceasta poate fi realizată prin intermediul unei liste de inițializare în definiția constructorului sau prin instrucțiuni în corpul constructorului.

```
MyClass(int xValue, double yValue) : x(xValue), y(yValue) {} // Inițializare membrilor
folosind lista de inițializare
```

4. Alocarea de memorie: Dacă obiectul conține resurse care necesită alocare dinamică de memorie, constructorul poate efectua această alocare.
5. Efectuarea de operațiuni suplimentare: Constructorul poate efectua și alte operațiuni necesare pentru pregătirea obiectului pentru utilizare, cum ar fi validarea datelor, inițializarea altor obiecte sau apelarea altor metode.

În concluzie, construirea unui obiect în C++ implică crearea unei instanțe a unei clase și inițializarea acesteia cu ajutorul constructorului corespunzător. Constructorul este responsabil pentru inițializarea membrilor și pentru pregătirea obiectului pentru utilizare în cadrul aplicației.

Modalitatea de apelare a constructorilor prin includerea argumentelor între paranteze, așa cum am arătat mai sus, se numește *forma funcțională*. Dar constructorii pot fi apelați folosind și cu alte sintaxe:

În primul rând, constructorii cu un singur parametru pot fi apelați folosind sintaxa de inițializare a unei variabile (un semn egal urmat de argument):

```
nume_clasă id_obiect = valoare_de_inițializare;
```

Mai recent, C++ a introdus posibilitatea apelării constructorilor prin *inițializarea uniformă*, care este, în mare măsură, la fel ca forma funcțională, dar folosește acolade ({}) în loc de paranteze (()):

```
nume_clasă id_obiect { valoare, valoare, valoare, ... }
```

Opțional, această ultima sintaxă poate include un semn egal înainte de acolade.

Iată un exemplu cu cele patru modalități de a construi obiecte dintr-o clasă al cărei constructor are un singur parametru:

```
// clase și inițializarea uniformă
#include <iostream>
using namespace std;
```

```
class Cerc {
    double raza;
  public:
    Cerc(double r) { raza = r; }
    double circum() {return 2*raza*3.14159265;}
};

int main () {
  Cerc foo (10.0);   // forma funcțională
  Cerc bar = 20.0;   // inițializare prin atribuire
  Cerc baz {30.0};   // inițializare uniformă
  Cerc qux = {40.0}; // la fel ca POD

  cout << "circumferinta lui foo: " << foo.circum() << '\n';
  return 0;
}
```

2 Aplicații

2.1 Cerc

În cadrul unui proiect, implementați clasa *Cerc*, cu următoarele caracteristici:

- clasa va avea un singur atribut de tip ***double***, *raza*.
- clasa va avea trei metode:
    a. *circum*, care va returna valoarea circumferinței cercului.
    b. *getRaza*, care returnează raza cercului;
    c. *setRaza*, care actualizează raza unui cerc.
- clasa va conține trei constructori (un constructor implicit, un constructor cu parametri și un constructor de copiere) și un destructor.

Cerință: să se construiască un obiect de tip *Cerc* și să se calculeze operații conform metodelor asupra lui.

2.2 Produs

În cadrul unui proiect, implementați clasa *Produs*, cu următoarele caracteristici:

- atribute:
    a. *nume*, de tip ***char***, care va reține numele produsului;
    b. *pret*, de tip ***double***, care va reține prețul produsului;
    c. *stoc*, de tip ***int***, care va reține stocul disponibil.
- metode:
    a. *valoare_stoc*, care va returna valoarea totală a produselor din stoc.
    b. *getPret*, care returnează prețul unui produs;
    c. *setPret*, care actualizează prețul unui produs;
    d. *actualizare_stoc*, care actualizează stocul unui produs;
    e. *afisare_detalii*, care afișează detalii despre produs.
- clasa va conține doi constructori (un constructor implicit și un constructor cu parametri) și un destructor.

Cerință: să se construiască un obiect de tip *Produs* și să se calculeze operații conform metodelor asupra lui.

2.3 Fracție

În cadrul unui proiect, implementați clasa *Fractie*, cu următoarele caracteristici:

- atribute:
    a. *numarator*, de tip ***int***, care va reține numărătorul fracției;

b. *numitor*, de tip ***int***, care va reține numitorul fracției;

- metode:
  a. *getNumarator*, care returnează care returnează numărătorul fracției;
  b. *setNumarator*, care returnează numitorul fracției;
  c. *getNumitor*, care returnează care returnează numărătorul fracției;
  d. *setNumitor,* care returnează numitorul fracției;
  e. *adunare*, care adună două fracții;
  f. *scadere*, care scade două fracții;
  g. *produs*, care înmulțește două fracții;
  h. *cât*, care împarte două fracții;
  i. *simplifica*, care simplifică o fracție;
  j. *afisare*, care afișează o fracție;
  k. *valoare_fractie,* care va returna valoarea fracției ca un număr în virgulă mobilă.

Cerință: să se construiască un obiect de tip *Fractie* și să se calculeze operații conform metodelor asupra lui.

3 Rezolvări

3.1 Cerc

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

class Cerc {
private:
    double raza;

public:
    // Constructor implicit
    Cerc() : raza(0.0) {}

    // Constructor cu parametri
    Cerc(double r)
    {
        raza = r;
    }

    //initializare directa
    //Cerc(double r) : raza(r) {}

    // Constructor de copiere
    Cerc(const Cerc& other) : raza(other.raza) {}

    // Destructor
    ~Cerc() {}

    // Metoda pentru calculul circumferintei cercului
    double circum() const {
        return 2 * 3.14159 * raza;
    }

    // Metoda pentru obținerea razei cercului
    double getRaza() const {
        return raza;
    }

    // Metoda pentru actualizarea razei cercului
    void setRaza(double r) {
        raza = r;
    }

    double aria();
};
```

```
double Cerc::aria()
{
    return 3.14159 * raza * raza;
}

int main() {

    Cerc cerc1;

    Cerc cerc2 (10.0);   // forma funcțională
    Cerc cerc3 = 20.0;   // inițializare prin atribuire
    Cerc cerc4 {30.0};   // inițializare uniformă
    Cerc cerc5 = {40.0}; // la fel ca POD

    cout << cerc2.circum() << '\n';
    cout << cerc3.circum() << '\n';
    cout << cerc4.circum() << '\n';
    cout << cerc5.circum() << '\n';

    // Crearea unui obiect de tip Cerc
    Cerc cerc(5.0); // Constructor cu parametri

    // Afișarea circumferinței cercului
    cout << "Circumferinta cercului cu raza " << cerc.getRaza() << " este: " <<
cerc.circum() << endl;

    // Actualizarea razei cercului
    cerc.setRaza(7.0);

    // Afișarea noii circumferințe
    cout << "Circumferinta cercului cu noua raza " << cerc.getRaza() << " este: " <<
cerc.circum() << endl;

    return 0;
}
```

3.2 Produs

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

using namespace std;

class Produs
{
private:
    char* nume;
```

```
    double pret;
    int stoc;

public:

    // constructor implicit
    Produs(): nume(nullptr), pret(0.0), stoc(0) {}

    // constructor cu un parametru
    Produs(char* n)
    {
        nume = new char[strlen(n) + 1];
        strcpy(nume, n);
    }

    // constructor cu trei parametri
    Produs(char* n, double p, int s)
    {
        nume = new char[strlen(n) + 1];
        strcpy(nume, n);
        pret = p;
        stoc = s;
    }

    // destructor
    ~Produs()
    {
        delete nume;
    }

    // definire getter
    double getPret()
    {
        return pret;
    }

    // definire setter
    void setPret(double p)
    {
        pret = p;
    }

    // definire metoda valoare_stoc
    double valoare_stoc()
    {
        return pret * stoc;
```

```
    }

    // definire actualizare_stoc
    void actualizare_stoc(int s)
    {
        stoc = s;
    }

    // definire afiseaza detallii
    void afisare_detalii()
    {
        if(pret == 0 || stoc == 0 || nume == nullptr)
        {
            cout << "Datele produsului nu au fost definite!\n\n";
        }
        else
        {
            cout << "Produsul " << nume << " are urmatoarele caracteristici: \n";
            cout << " - pret: " << pret << "\n";
            cout << " - stoc: " << stoc << "\n\n";
        }
    }
};

int main()
{
    Produs p2;
    Produs p3("Laptop");
    Produs p1("Telefon",1000,100);

    p2.afisare_detalii();
    p3.afisare_detalii();

    p1.afisare_detalii();
    p1.setPret(1200);
    p1.afisare_detalii();

    cout << p2.valoare_stoc();

    return 0;
}
```

3.3 Fractie

```
#include <iostream>

using namespace std;
```

```
class Fractie
{
private:
    int numarator;
    int numitor;

    // cmmdc euclid
    int cmmdc(int a, int b)
    {
        if(b == 0){
            return a;
        }

        return cmmdc(b, a % b);
    }

public:

    // constructor implicit
    Fractie(): numarator(0), numitor(1) {}

    // constructor
    Fractie(int m, int n): numarator(m), numitor(n) {}

    // getter numarator
    int getNumarator()
    {
        return numarator;
    }

    // setter numarator
    void setNumarator(int m)
    {
        numarator = m;
    }

    //getter numitor
    int getNumitor()
    {
        return numitor;
    }

    //setter numitor
    void setNumitor(int n)
    {
```

```
    numitor = n;
}

// simplificare, nu pastram fractia originala
void simplificare()
{
    int f = cmmdc(numarator, numitor);
    numarator /= f;
    numitor /= f;
}

// adunare folosind membri nestatici
Fractie adunare(Fractie& f)
{
    // calculam numaratorul si numitorul noii fractii obtinute prin adunare
    int m = numarator * f.numitor + f.numarator * numitor;
    int n = numitor * f.numitor;

    Fractie fractie(m, n);
    fractie.simplificare();

    return fractie;
}

// scadere folosind membri statici
static Fractie scadere(Fractie& f1, Fractie& f2)
{
    int m = f1.numarator * f2.numitor - f2.numarator * f1.numitor;
    int n = f1.numitor * f2.numitor;

    Fractie fractie(m, n);
    fractie.simplificare();

    return fractie;
}

// produsul a doua fractii
Fractie produs(Fractie& f)
{
    int m = numarator * f.numarator;
    int n = numitor * f.numitor;

    Fractie fractie(m, n);
    fractie.simplificare();

    return fractie;
```

```
    }

    // catul a doua fractii
    Fractie cat(Fractie& f)
    {
        int m = numarator * f.numitor;
        int n = numitor * f.numarator;

        Fractie fractie(m, n);
        fractie.simplificare();

        return fractie;
    }

    // afisare
    void afisare()
    {
        cout << numarator << "/" << numitor << endl;
    }

    // valoare zecimala a fractiei
    double valoare_fractie()
    {
        return static_cast<double>(numarator) / numitor;
    }

};

int main()
{
    Fractie f1(1,2);
    Fractie f2(1,3);

    Fractie f_a, f_s, f_p, f_c;

    f_a = f1.adunare(f2);
    f_p = f1.produs(f2);
    f_c = f1.cat(f2);

    cout << "Rezultatul adunarii fractiei f1 cu fractia f2 este: "; f_a.afisare();
    //f_s.afisare();
    cout << "Rezultatul scaderii fractiei f1 cu fractia f2 este: "; Fractie::scadere(f1,
f2).afisare();
    cout << "Rezultatul produsului fractiei f1 cu fractia f2 este: "; f_p.afisare();
    cout << "Rezultatul catului fractiei f1 cu fractia f2 este: "; f_c.afisare();
```

```
    cout << "Valoarea zecimala a fractiei "; f_c.afisare(); cout<< " este " <<
f_c.valoare_fractie();

    return 0;
}
```