# PROGRAMARE ORIENTATĂ PE OBIECTE

# LABORATOR 8

**Anul universitar 2023 – 2024**

**Semestrul II**

## *1 Noțiuni teoretice*

### *1.1 Agregare și Compoziție*

#### 1.1.1 Introducere

Atât agregarea, cât și compoziția sunt forme de relații între clase, de tip „has-a” (are un). Ele sunt moduri prin care o clasă folosește alte clase ca parte din structura sa internă. Diferența cheie dintre ele este cine controlează viața obiectelor conținute. Dacă obiectul conținut este creat și distrus de clasă, avem compoziție. Dacă obiectul este creat în altă parte și doar referit, avem agregare.

Agregarea și compunerea se referă la prezența unei referințe pentru un obiect într-o altă clasă. Acea clasă practic va refolosi codul din clasa corespunzătoare obiectului.

- Agregarea (aggregation) - obiectul-container poate exista și în absența obiectelor agregate, de aceea este considerată o asociere slabă (weak association). În exemplul de mai jos, un raft de bibliotecă poate exista și fără cărți.
- Compunerea (composition) - este o agregare puternică (strong), indicând că existența unui obiect este dependentă de un alt obiect. La dispariția obiectelor conținute prin compunere, existența obiectului container încetează. De exemplu, o carte nu poate exista fără pagini.

Inițializarea obiectelor conținute poate fi făcută în 3 momente de timp distincte:

- la definirea obiectului (înaintea constructorului: folosind fie o valoare inițială, fie blocuri de inițializare)
- în cadrul constructorului
- chiar înainte de folosire (acest mecanism se numește inițializare leneșă (lazy initialization)

Agregarea și compoziția sunt procedee prin care se creează un nou tip de date (o noua clasă) folosind tipuri de date (clase) existente.

Relația de agregare sau de compoziție este o relație între clase de **tipul “has a”, “has many”**.

Dacă reutilizăm codul ⇒ evitam să avem cod duplicat, ceea ce ne conduce la:

- eficiență în materie de timp, proiectul o să fie implementat mai repede
- lizibilitatea codului (arhitectură robustă, modularizare)

```
//cod
class A {
    //lista atribute;
    //lista metode;
};
class B{
    private:
    A a;
    //alte atribute si metode
};
```

Ce observăm?

- clasa B are un atribut de tipul clasei A
- clasa B nu are acces direct la atributele din A, dar poate să utilizeze funcțiile acesteia pentru a realiza operațiile de interes
- se protejează încapsularea datelor din A
- se reutilizează codul deja implementat din A

Când să folosești fiecare tip de relație?

- Compoziție – când componenta este o parte esențială a clasei.
- Agregare – când componenta poate fi partajată sau schimbată independent.

| Aspect | Compoziție | Agregare |
|---|---|---|
| *Declarare* | Obiectul este membru direct | Pointer sau referință |
| *Cine creează obiectul?* | Clasa care îl conține | Obiectul este creat în altă parte și transmis |
| *Durata de viață* | Legată de viața clasei părinte | Independentă de clasa părinte |
| *Constructor* | Inițializezi direct în lista de inițializare | Primești pointer/referință în constructor |
| *Distrugere* | Automat distrus odată cu obiectul părinte | NU este distrus automat (trebuie gestionat separat) |
| *Posibil partaj* | NU – fiecare instanță are propria componentă | DA – mai multe obiecte pot folosi același component |
| *Null-safety* | Mereu valid – nu există nullptr | Poate fi nullptr – trebuie verificat |

### 1.1.2 Compoziția

Compoziția implică faptul că obiectul conținut este o parte esențială a clasei care îl conține. Dacă obiectul conținător este distrus, atunci și componentele sale sunt distruse automat. De exemplu, dacă o carte este ștearsă, și paginile sale dispar – pentru că fac parte integrantă din ea. Este utilizată atunci când componenta nu are sens să existe separat.

Este o relație de **tip „parte din” (part-of)**:

- Clasa conține un obiect al altei clase ca membru direct (non-pointer, non-smart pointer).
- Obiectul membru este creat și distrus odată cu clasa părinte.
- Este o relație mai puternică.

```
#include <iostream>
using namespace std;
class A {
    private:
        int atr;
    public:
        A():atr(0) { /*pana sa se intre in implementarea constructorului,
            se apeleaza pseudoconstructorul pentru atributul atr
            care aloca spatiu ptr atr si il initializeaza cu 0 */
            cout << "Constructor fara parametri" << endl;
        }
        A(const A &x):atr(x.atr) {
            /*se apeleaza pseudoconstructorul pentru atributul atr,
            care aloca spatiu ptr atr si il initializeaza cu x.atr*/
            cout << "Constructor de copiere" << endl;
        }
        A &operator = (const A &x) {
            //nu se apeleaza pseudoconstructor pentru atr, deoarece are spatiu alocat
            cout << "Operator = " << endl;
            atr = x.atr;
            return *this;
        }
        ~A() {
            cout << "Destructor" << endl;
        }
};
class B {
    private:
        A a; // B are un atribut de tip A
        /*in mod automat se genereaza un constructor fara parametri,
        constructor de copiere, operator =, destructor */
};
int main() {
    B b;
    B c(b);
    c = b;
    return 0;
}
```

Ce observăm?

- pentru crearea unui obiect de tip B se apeleaza in mod implicit constructorul default din A (daca nu este specificat altul)

- constructorul de copiere din B apeleaza automat constructorul de copiere din A (daca nu este specificat altul)
- operatorul = (generat default) al lui B, il apeleaza pe cel al tipului de date A
- destructorul clasei B apeleaza automat destructorul clasei A

### 1.1.3 Agregarea

#### 1.1.3.1 Formă generală

Agregarea este o relație mai slabă, în care clasa doar face referire la un obiect care există în afară. Clasa nu este responsabilă pentru crearea sau distrugerea acelui obiect. Este ideală când vrei să reutilizezi un obiect existent sau când componenta poate fi partajată între mai multe clase. De exemplu, un șofer poate folosi o mașină, dar nu „deține" motorul – poate doar îl controlează temporar.

Este o relație **de tip „are un"**, dar mai slabă.

- Clasa conține un pointer sau referință către alt obiect.
- Obiectul agregat nu este creat/distrus de clasa părinte.
- Relația este mai flexibilă.

```
#include <iostream>

using namespace std;

class A
{
private:
    int atr;

public:
    A() : atr(0)
    {
        cout << "Constructor implicit (default) \n";
    }

    A(int a)
    {
        this->atr = a;
        cout << "Constructor cu parametri \n";
    }

    A(const A& other)
    {
        this->atr = other.atr;
        cout << "Constructor de copiere (copy) \n";
    }

    ~A()
    {
```

```
        cout << "Destructor A \n";
    }

    A& operator=(const A& other)
    {
        /// a = a
        cout << "Operator = \n";
        if(this != &other)
        {
            this->atr = other.atr;
        }
        return *this;
    }
};

class B
{
private:
    A a;
};

class C
{
private:
    A* a;

public:
    C(A* ptr)
    {
        a = ptr;
    }

    ~C()
    {
        cout << "Destructor C \n";
    }
};

int main()
{
    /*B b;
    B c(b);
    c = b;
    */

    /// agregare
    A* a = new A;
    C c(a);
    delete a;
    return 0;
}
```

Ce se întâmplă:

- C nu creează obiectul A, ci primește un pointer la el.
- Nu se apelează constructorul de copiere, pentru că nu se copiază obiectul A.
- Nu se face `delete` → deci C nu deține obiectul → e agregare.
- În `main()`, `A a;` e creat în exterior, iar `C d(&a);` îl folosește.

Ce output te aștepți să vezi?

```
Constructor fara parametri      // B b;
Constructor de copiere          // B c(b);
Operator =                      // c = b;
Destructor                      // b.a
Destructor                      // c.a
Constructor fara parametri      // A a;
Destructor C (nu distrug a)     // d
Destructor                      // a
```

Observație:

- Destructorul lui A se apelează doar o dată pentru a, deoarece C nu îl distruge.
- Destructorii pentru a din b și c se apelează pentru că sunt parte din B (compoziție).

În C++, **doar pointerii și referințele** pot face ca obiectul să fie *doar folosit*, fără a-l *deține.* Pentru ca o relație între două clase să fie considerată agregare, obiectul conținut trebuie declarat ca pointer sau referință; în caz contrar, dacă este declarat ca membru direct (non-pointer, non-referință), relația este de compoziție, nu de agregare. Diferența dintre agregare cu pointer și agregare cu referință e subtilă, dar importantă în C++.

| Aspect | Pointer | Referință |
|---|---|---|
| Poate fi null? | Da | Nu |
| Se poate schimba ulterior? | Da | Nu |
| Verificare nullptr necesară? | Da | Nu |
| Inițializare obligatorie? | Nu (poți omite) | Da (în constructor) |
| Risc dangling (obiect șters prea devreme)? | Da | Da |
| Polimorfism? | Da | Da |
| Ideal pentru componente opționale? | Da | Nu |
| Ideal pentru obiecte „obligatorii”? | Nu | Da |

#### 1.1.3.2 Agregare cu pointer

Prin folosirea unui pointer (A*) către un alt obiect, se poate crea o relație de agregare. Această abordare oferă flexibilitate maximă, deoarece pointerul poate fi schimbat, poate fi nul (opțional), și poate fi folosit pentru polimorfism. Cu toate acestea, programatorul trebuie să verifice dacă pointerul e valid înainte de a-l folosi și să se asigure că obiectul nu este distrus în altă parte.

```
class Motor {
public:
    void porneste()
    {
        cout << "Motor pornit\n";
    }
};

class Masina {
    // pointer = agregare
    Motor* motor;
public:
    Masina(Motor* m) : motor(m) {}

    void porneste()
    {
        if (motor){
            motor->porneste();
        }
    }
};
```

Avantaje:

- Poate fi nullptr – ideal pentru componente opționale.
- Se poate reatribui – obiectul agregat poate fi schimbat în timpul execuției.
- Permite inițializare întârziată (lazy init).
- Se integrează ușor cu alocare dinamică (new) și polimorfism.
- Potrivit pentru structuri flexibile (ex: vector de pointeri).

Dezavantaje:

- Necesită verificare nullptr înainte de utilizare.
- Risc de dangling pointer dacă obiectul este distrus în altă parte.
- Poate fi mai greu de urmărit – mai puțin clar semantic.
- Necesită atenție la gestionarea memoriei, mai ales cu new/delete.

#### 1.1.3.3 Agregare cu referință

Agregarea cu referință (A&) este o variantă mai sigură a agregării, deoarece obiectul referit trebuie să existe la momentul inițializării și nu poate fi nul. În același timp, referința nu poate fi schimbată după inițializare. Este ideală atunci când ai nevoie ca obiectul să existe cu siguranță și nu vrei să ai de-a face cu verificări de tip `nullptr`.

```
class Masina {
    // referinta = agregare
    Motor& motor;
```

```
public:
    Masina(Motor& m) : motor(m) {}

    void porneste() {
        motor.porneste();
    }
};
```

Avantaje:

- Nu poate fi null – acces garantat la un obiect valid.
- Nu necesită verificare de validitate la acces.
- Codul este mai clar semantic: obiectul este obligatoriu.
- Este mai simplu și mai sigur pentru utilizare imediată.

Dezavantaje:

- Trebuie inițializat în constructor – nu poate fi neinițializat.
- Nu poate fi reatribuit – referința rămâne legată de același obiect.
- Nu este potrivit pentru componente opționale.
- Poate deveni dangling dacă obiectul referit este distrus prematur.

*Notă: Un dangling pointer (sau o dangling referință) apare atunci când un pointer sau o referință indică spre o zonă de memorie invalidă – adică spre un obiect care a fost deja distrus sau care nu mai există. Este una dintre cele mai periculoase probleme în C++, pentru că duce la comportament nedefinit (undefined behavior): crăpări, date corupte, bug-uri greu de găsit.*

## *1.2 Moștenirea claselor*

### 1.2.1 Generalități

Derivarea este o relație între clase **de tipul “is a” / “is many”**.

Moștenirea (numită si derivare) este un mecanism de refolosire a codului. Totodată, oferă posibilitatea de a defini o clasă care ,,extinde” o clasă existentă (la codul de bază din clasa existenta adăugându-se noi atribute și/sau metode).

```
//cod
class Produs {
    private:
        char *nume;
        int pret;
        char cod[10];
};

class Electrocasnic {
    private:
```

```
        char *nume;
        int pret;
        char cod[10];
        int durata_garantie;
};

class Aliment {
    private:
        char *nume;
        int pret;
        char cod[10];
        int durata_expirare;
};

class Jucarie {
    private:
        char *nume;
        int pret;
        char cod[10];
        int varsta_recomandata[2];
};

class TV {
    private:
        char *nume;
        int pret;
        char cod[10];
        int durata_garantie;
        double diagonala;
};

class Masina_cafea {
    private:
        char *nume;
        int pret;
        char cod[10];
        int durata_garantie;
        char *tip; //espresor, filtru
};
/* iar exemplele pot continua */
//cod
```

Putem observa ca avem o clasa de bază, Produs, și că putem distinge mai multe categorii de produse. Aceste categorii de produse sunt niște clase care, pe lângă faptul că au comportamentul clasei Produs, au în plus atribute și comportamente specifice (clase specializate).

*De exemplu: Masina_cafea si TV-ul sunt electrocasnice, dar acestea au comportamente (și atribute) specifice. (Un TV nu poate să facă cafea, iar masina_cafea nu difuzează știrile de la ora 17:00).*

Problema întâmpinată: Am rescris foarte mult cod. Cum fac să evit asta?

```
//cod
class Produs { //clasa de baza
    protected:
        char *nume;
        int pret;
        char cod[10];
};

class Electrocasnic : public Produs {//clasa derivata (din Produs), dar si clasa de baza
    protected:
        int durata_garantie;
};

class Aliment : public Produs{//clasa derivata (din Produs)
    private://daca implementam o clasa care sa fie derivata din clasa ALiment, folosim
protected
        int durata_expirare;
};

class Jucarie : public Produs{//clasa derivata (din Produs)
    private://daca implementam o clasa care sa fie derivata din clasa Jucarie, folosim
protected
        int varsta_recomandata[2];
};

class TV : public Electrocasnic{//clasa derivata (din Electrocasnic)
    private:
        double diagonala;
};

class Masina_cafea : public Electrocasnic{//clasa derivata (din Electrocasnic)
    private:
        char *tip; //expresor, filtru
};
/* iar exemplele pot continua */
//cod
```

Am rezolvat problema întâmpinată! Acum putem afirma că:

- Derivarea este un procedeu prin care se creează un nou tip de date (o noua clasă) folosind o clasă existentă, mai exact, adăugăm un cod nou (atribute noi, metode noi) codului deja existent ⇒ se preia interfața clasei de bază.
- În clasa derivată sunt moștenite toate atributele și metodele clasei de bază și le pot accesa direct dacă sunt declarate `public` sau `protected`.

Deci, clasele în C++ pot fi extinse, creându-se noi clase care păstrează caracteristici ale clasei de bază. Acest proces, cunoscut ca moștenire, implică **o clasă de bază** și **o clasă derivată**: clasa derivată moștenește membrii clasei de bază, iar la aceștia se pot adăuga proprii membri.

De exemplu, să ne imaginăm o serie de clase pentru a descrie două tipuri de poligoane: dreptunghiuri și triunghiuri. Aceste două tipuri de poligoane au anumite proprietăți comune, precum valorile necesare pentru calcularea ariilor: ambele pot fi descrise simplu ca înălțime și lățime (sau bază).

Clasa Poligon conține membri comuni pentru ambele tipuri de poligoane. În cazul nostru: latime și inaltime. Iar clasele Dreptunghi și Triunghi reprezintă clase derivate, având caracteristici și funcționalități diferite de la un tip de poligon la altul.

Clasele derivate din alte clase moștenesc toți membrii accesibili ai clasei de bază. Aceasta înseamnă că dacă o clasă de bază are un membru A și creăm o clasă derivată care are un membru B, aceasta din urmă conține atât pe A cât și pe B.

Relația de moștenire între două clase se declară în clasa derivată. Definițiile claselor derivate respectă următoarea sintaxă:

```
class nume_clasă_derivată: public nume_clasă_bază
{ /*...*/ };
```

unde `nume_clasă_derivată` este numele clasei derivate și `nume_clasă_bază` este numele clasei care stă la bază. Specificatorul de acces public poate fi înlocuit cu orice alt specificator de acces (`protected` sau `private`). Acest specificator limitează la nivelul cel mai accesibil pentru membrii moșteniți de la clasa de bază: membrii cu un nivel de accesibilitate mai mare sunt moșteniți cu acest nivel, în timp ce membrii cu un nivel de acces egal sau mai restrictiv își vor păstra restricția în clasa derivată.

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

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

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

class Triunghi: public Poligon {
  public:
    int aria ()
      { return latime * inaltime / 2; }
  };
```

```
int main () {
  Dreptunghi drept;
  Triunghi trg;
  drept.seteaza_valori (4,5);
  trg.seteaza_valori (4,5);
  cout << drept.aria() << '\n';
  cout << trg.aria() << '\n';
  return 0;
}

20
10
```

Obiectele clasei Dreptunghi și Triunghi conțin membrii moșteniți de la Poligon. Aceștia sunt: latime, inaltime și seteaza_valori.

Specificatorul de acces `protected` folosit în clasa Poligon este similar lui `private`. Singura diferență apare în legătură cu moștenirea: când o clasă o moștenește pe alta, membrii clasei derivate pot accesa membrii protejați moșteniți de la clasa de bază, dar nu și membrii privați.

Declarând `latime` și `inaltime` cu `protected` în loc de private, acești membri vor fi accesibili și pentru clasele derivate Dreptunghi și Triunghi, față de accesarea doar de către membrii clasei Poligon. Dacă s-ar fi declarat cu `public`, ar fi putut fi accesați de oriunde.

Putem să sintetizăm diversele tipuri de acces în raport cu funcțiile care pot să îi acceseze astfel:

| Access | public | protected | private |
|---|---|---|---|
| membrii aceleiași clase | da | da | da |
| membrii claselor derivate | da | da | nu |
| ne-membri | da | nu | nu |

unde "ne-membri" reprezintă acces de oriunde din exteriorul clasei, ca de exemplu din `main`, din altă clasă sau dintr-o funcție oarecare.

În exemplul de mai sus, membrii moșteniți de Dreptunghi și Triunghi au aceleași permisiuni de acces ca și cum ar fi aparținut clasei de bază Poligon:

```
Poligon::latime           // protected access
Dreptunghi::latime         // protected access

Poligon::seteaza_valori()    // public access
Dreptunghi::seteaza_valori()  // public access
```

Aceasta se datorează relației de moștenire declarată prin folosirea cuvântului cheie public pentru fiecare dintre clasele derivate:

```
class Dreptunghi: public Poligon { /* ... */ }
```

Cuvântul cheie `public` de după (:) permite membrilor moșteniți de la clasa părinte (în cazul acesta Poligon) accesibilitate maximă pe care îl vor avea clasele derivate (în acest caz Dreptunghi). Deoarece cuvântul `public` rezervă nivelul maxim de accesibilitate, utilizarea sa permite clasei derivate să moștenească toți membrii cu același nivel ca în clasa de bază.

Cu ajutorul cuvântului cheie `protected`, toți membrii publici ai clasei de bază sunt moșteniți ca `protected` în clasa derivată. Reciproc, dacă este specificat cel mai restrictiv nivel (`private`), toți membrii clasei de bază sunt moșteniți ca `private` și de aceea nu pot fi accesați din clasa derivată.

De exemplu, dacă fiica ar fi o clasă derivată a clasei mama, atunci am defini astfel:

```
class Fiica: protected Mama;
```

Aceasta ar face ca membrii clasei Fiica moșteniți de la mamă să fie setați cu nivelul de acces cel mai puțin restrictiv cu ajutorul lui `protected`. Adică, toți membrii cu specificatorul de acces public din Mama vor deveni `protected` în Fiica. Bineînțeles, aceasta nu restricționează Fiica de la a avea declarați proprii membri publici. Acel cel mai puțin restrictiv nivel de acces este setat numai pentru membrii moșteniți de la Mama.

Dacă nu este precizat niciun nivel de acces pentru moștenire, compilatorul consideră privat pentru clasele declarate cu cuvântul cheie `class` și public pentru cele declarate cu `struct`.

În principiu, o clasă derivată moștenește toți membrii clasei de bază, exceptând:

- constructorii și destructorii
- operatorul de atribuire a membrilor (operatorul=)
- prietenii
- membrii privați

Deși constructorii și destructorii clasei de bază nu sunt moșteniți drept constructori și destructori ai clasei derivate, pot fi încă apelați de către constructorul clasei derivate. Numai dacă nu este altfel precizat, constructorii clasei derivate apelează constructorii impliciți ai claselor de bază (de exemplu, constructorul fără parametri), care trebuie să existe.

Apelarea altui constructor al clasei de bază este posibilă, prin folosirea aceleiași sintaxe ca la inițializarea variabilelor membru din lista de inițializare:

```
nume_constructor_derivat (parameteri) : nume_constructor_bază (parameteri) {...}
```

De exemplu:

```
// constructori si clase derivate
#include <iostream>
using namespace std;

class Mama {
  public:
    Mama ()
      { cout << "Mama: fara parametri\n"; }
    Mama (int a)
      { cout << "Mama: parametru de tip int\n"; }
};

class Fiica : public Mama {
  public:
    Fiica (int a)
      { cout << "Fiica: parametru de tip int\n\n"; }
};

class Fiu : public Mama {
  public:
    Fiu (int a) : Mama (a)
      { cout << "Fiu: parametru de tip int\n\n"; }
};

int main () {
  Fiica kelly(0);
  Fiu bud(0);

  return 0;
}
Mama: fara parameteri
Fiica: parametru de tip int

Mama: parametru de tip int
Fiu: parameteru de tip int
```

Să observăm diferența între varianta cu apelarea constructorului clasei Mama la crearea unui nou obiect Fiica și varianta cu un obiect Fiu. Se datorează declarațiilor diferite de constructor pentru Fiica și Fiu:

```
Fiica (int a)          // nimic specificat: apel al constructorului implicit
Fiu (int a) : Mama (a)  // constructor specificat: apelul acelui constructor
```

### 1.2.2 Comportamentul metodelor din clasa derivată

În programarea orientată pe obiect, clasele derivate extind sau specializează comportamentul claselor de bază. De aceea, este important să înțelegem modul în care constructorii, operatorii și destructorii se comportă în contextul moștenirii, precum și ce elemente se moștenesc sau nu între clase. Acest

comportament influențează atât inițializarea corectă a obiectelor, cât și eliberarea memoriei și menținerea unei arhitecturi corecte și sigure.

#### 1.2.2.1 Constructorii

- Din punct de vedere al ordinii, prima oară se apelează constructorul/clasa constructorilor din clasa de bază și abia apoi cei din clasa derivată.
- În cazul în care clasa derivată nu are niciun constructor declarat, se generează unul default care va apela constructorul fără parametri al clasei de bază (acesta trebuie să existe).
- Dacă în clasa derivată nu avem un constructor de copiere, se generează unul automat care îl va apela pe cel din clasa de bază (similar cu comportamentul de mai sus).
- Dacă implementăm un constructor pentru clasa derivată care nu apelează explicit un constructor al clasei de bază, atunci, în mod implicit, se va apela constructorul fără parametri al clasei de bază. Dacă acesta nu există, rezultă eroare de compilare.

#### 1.2.2.2 Operatorul =

- Dacă operatorul = este generat automat, se va apela operatorul = din clasa (sau clasele) de bază.
- Dacă operatorul = este implementat de noi în clasa derivată, trebuie să ne asigurăm că acesta apelează explicit operatorul = din clasa de bază, pentru a copia corect partea moștenită.

#### 1.2.2.3 Destructorul

- Ordinea apelării destructorilor este inversă față de constructori: mai întâi se apelează destructorul clasei derivate, apoi cel al clasei de bază.
- Apelul se face automat, nu este nevoie să apelăm explicit destructorul bazei, deoarece el va fi invocat automat la distrugerea obiectului.

În concluzie:

- Constructorii, destructorii și funcțiile friend nu se moștenesc în clasele derivate.
- În general, operatorii implementați ca funcții membre se moștenesc. Excepție: operatorul de atribuire =, care nu se moștenește – trebuie implementat explicit dacă este necesar în clasa derivată.
- Funcțiile friend pot accesa toate atributele (`private / protected / public`) din clasa în care sunt declarate ca prietene.
- Funcțiile membre statice se comportă ca orice altă funcție membră și sunt moștenite în clasa derivată.

### *1.3 Agregare și compoziție vs derivare*

Amândouă mecanismele reutilizează codul scris pentru o clasa de bază / simplă într-o altă clasă mai complexă. În ambele cazuri se folosesc liste de inițializare pentru constructori pentru a crea obiectele de bază (chiar daca apelul diferă prin sintaxă).

#### 1.3.1 Diferențe

Agregarea și compoziția

- folosită când se dorește reutilizarea unui tip de date A pentru generarea altui tip de date B (fără a prelua interfața lui A)
- se integrează în clasa mai complexă un atribut (sau mai multe) de tipul clasei mai simple
- utilizatorii noii clase vor vedea doar interfața acesteia
- nu va mai fi de interes interfața clasei de bază

Derivarea

- folosita când se dorește preluarea interfeței clasei de bază
- utilizatorul va vedea atât interfața clasei de bază, cât și a celei derivate

Nu există nicio diferență în termeni de memorie ocupată și durată de execuție.

#### 1.3.2 Avantajele derivării

- Putem sa adăugăm cod nou fără a introduce bug-uri în codul existent.
- Erorile se găsesc în codul nou ⇒ mai ușor de căutat / găsit.
- Clasele sunt clar separate. (Codul poate fi refolosit și în alte locuri fără a avea erori).

### *1.4 Optimizarea implementării*

#### 1.4.1 Implementare în clase diferite

Pentru a implementa o problemă cu mai multe clase în CodeBlocks folosind #ifndef, #define și #endif, urmați acești pași:

1. Creați fișierele header pentru fiecare clasă:
    a. Creați un fișier header pentru fiecare clasă pe care o aveți în programul dvs. Acesta ar trebui să conțină definiția clasei și orice prototipuri de funcții sau metode.
    b. De exemplu, pentru o clasă numită MyClass, creați un fișier numit MyClass.h.
2. Definiți macro-urile de includere multiplă în fișierele header:
    a. În fiecare fișier header, adăugați protecția împotriva includerii multiple folosind #ifndef, #define și #endif.
    b. Aceasta va preveni erorile de redefinire atunci când includeți fișierele header în alte fișiere.

3. Includeți fișierele header în fișierele sursă:
    a. În fișierele sursă (.cpp), includeți fișierele header folosind directiva #include.

Fișierele header (.h) și fișierele sursă (.cpp) sunt create manual de către programatori pentru a organiza codul lor într-o manieră modulară și ușor de gestionat.

- Fișierele header (.h): Aceste fișiere conțin definițiile claselor, prototipurile funcțiilor și alte declarații necesare pentru utilizarea acestora în alte fișiere sursă. Fișierele header sunt adesea incluse în alte fișiere sursă pentru a permite compilatorului să cunoască structura și interfața claselor și funcțiilor.
- Fișierele sursă (.cpp): Aceste fișiere conțin implementarea detaliată a claselor și funcțiilor definite în fișierele header. În fișierele sursă, programatorii scriu codul care definește comportamentul fiecărei clase și a fiecărei funcții. De asemenea, în fișierele sursă pot fi incluse și alte fișiere header necesare pentru program.

Aceste fișiere sunt create manual de către programatori în editorul lor preferat de text și sunt organizate în funcție de structura și logica programului. De exemplu, o clasă ar putea avea un fișier header dedicat pentru definiția sa și un fișier sursă separat pentru implementarea sa. Acest lucru face ca codul să fie mai ușor de gestionat și de înțeles, deoarece fiecare componentă a programului are propriul său fișier asociat.

Într-un proiect CodeBlocks, puteți crea în cadrul unui proiect fișiere .h sau .cpp apelând comanda File, apoi New, apoi File și alegeți tipul de fișier.

### 1.4.2 Regula celor trei

Regula celor 3 este un principiu important în C++ care spune că dacă o clasă gestionează resurse proprii (ex: memorie alocată dinamic, fișiere, socket-uri etc.), atunci trebuie să definești explicit următoarele trei metode speciale:

1. Destructorul
2. Constructorul de copiere
3. Operatorul de atribuire (=)

Dacă o clasă deține un pointer (`int*, char*, A*`, etc.) care alocă memorie cu new, atunci:

- Destructorul trebuie să facă `delete` sau `delete[]` pentru a evita memory leaks.
- Constructorul de copiere trebuie să copieze corect obiectul, nu doar adresa pointerului (altfel am avea două obiecte care „cred” că dețin aceeași zonă de memorie).
- Operatorul de atribuire trebuie să facă deep copy și să elibereze memoria veche înainte de a copia.

# *2 Aplicații*

## *2.1 Laturi*

Într-un proiect, implementați următoarele clase:

```
class Latura
{

private:
    int lungime;

public:

    Latura();
    Latura(int);

    int getL() const;

    Latura& operator=(const Latura&);
    friend istream& operator>>(istream&, Latura&);

};

class Figura
{
protected:
    Latura* lat;
    int nrLat;

public:
    Figura();
    Figura(int);
    Figura(Latura*, int);
    Figura(const Figura&);
    ~Figura();

    friend ostream& operator<<(ostream&, const Figura&);
    Figura& operator=(const Figura&);

    int perimetru();

};

class Triunghi : public Figura
{
public:
    Triunghi();

    double arie();
};

class Patrat : public Figura
```

```
{
public:
    Patrat();
    double arie();
};

int main()
{

    Triunghi t;
    cout << t;
    cout << "Perimetrul triunghiului este: " << t.perimetru() << "\n";
    cout << "Aria triunghiului este: " << t.arie() << "\n";

    Patrat p;
    cout << p;
    cout << "Perimetrul patratului este: " << p.perimetru() << "\n";
    cout << "Aria patratului este: " << p.arie() << "\n";

    return 0;
}
```

Implementați două clase, Latura și Figura, care reprezintă laturile și formele geometrice, respectiv. Clasa Latura are un membru privat lungime și implementează metode pentru obținerea lungimii, atribuirea și citirea acesteia. Clasa Figura are un membru protejat care este un vector de laturi și un membru pentru numărul de laturi ale figuri. Clasa Figura implementează metode pentru calcularea perimetrului formei și este suprascrisă de clasele derivate Triunghi și Patrat. Clasa Triunghi inițializează un triunghi cu 3 laturi (constructor), iar clasa Patrat inițializează un pătrat cu 4 laturi (constructor). Pentru fiecare formă geometrică, să se afișeze perimetrul și aria calculată conform regulilor geometrice corespunzătoare.

Problema va fi implementată folosind clase în fișiere diferite și într-un singur fișier.

# *3 Rezolvări*

## *3.1 Laturi*

### 3.1.1 Versiune cu un singur fișier

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

using namespace std;

class Latura
{

private:
    int lungime;

public:

    Latura();
    Latura(int);

    int getL() const;

    Latura& operator=(const Latura&);
    friend istream& operator>>(istream&, Latura&);

};

class Figura
{
protected:
    Latura* lat;
    int nrLat;

public:
    Figura();
    Figura(int);
    Figura(Latura*, int);
    Figura(const Figura&);
    ~Figura();

    friend ostream& operator<<(ostream&, const Figura&);
    Figura& operator=(const Figura&);

    int perimetru();

};

class Triunghi : public Figura
{
public:
    Triunghi();
```

```
    double arie();
};

class Patrat : public Figura
{
public:
    Patrat();
    double arie();
};

Latura::Latura(): lungime(0) {}

Latura::Latura(int lungime): lungime(lungime) {}

int Latura::getL() const
{
    return lungime;
}

Latura&  Latura::operator=(const Latura& l)
{
    if (this != &l)
    {
        lungime = l.lungime;
    }

    return *this;
}

istream& operator>>(istream& in, Latura& l)
{
    cout << "Lungimea laturii este: ";
    in >> l.lungime;
    return in;
}

Figura::Figura()
{
    lat = NULL;
    nrLat = 0;
}

Figura::Figura(int nr)
{
    nrLat = nr;
    lat = new Latura[nrLat];
}

Figura::Figura(Latura* lat, int nrLat)
{
    this->nrLat = nrLat;
    this->lat = new Latura[nrLat];
    for(int i = 0; i < nrLat; i++)
```

```
    {
        this->lat[i] = lat[i];
    }
}

Figura::Figura(const Figura& f)
{
    this->nrLat = f.nrLat;
    this->lat = new Latura[f.nrLat];
    for(int i = 0; i < f.nrLat; i++)
    {
        this->lat[i] = f.lat[i];
    }
}

Figura::~Figura()
{
    delete[] lat;
}

ostream& operator<<(ostream& out, const Figura& f)
{
    out << "Aceasta este figura cu laturile: ";
    for(int i = 0; i < f.nrLat; i++)
    {
        out << f.lat[i].getL() << " ";
    }
    out << "\n";

    return out;
}

Figura& Figura::operator=(const Figura& f)
{
    if(this != &f)
    {
        delete[] lat;
        this->nrLat = f.nrLat;
        this->lat = new Latura[f.nrLat];
        for(int i = 0; i < f.nrLat; i++)
        {
            this->lat[i] = f.lat[i];
        }
    }
    return *this;
}

int Figura::perimetru()
{
    int suma = 0;
    for(int i = 0; i < nrLat; i++)
    {
        suma += lat[i].getL();
```

```
    }

    return suma;
}

Triunghi::Triunghi() : Figura(3)
{
    for(int i = 0 ; i < 3; i++)
    {
        cin >> lat[i];
    }
}

// acest constructor este echivalent cu cel de mai sus
/*
Triunghi::Triunghi()
{
    nrLat = 3;
    lat = new Latura[3];
    for(int i = 0 ; i < 3; i++)
    {
        cin >> lat[i];
    }
}
*/

double Triunghi::arie()
{
    int p = 0;
    for(int i = 0; i < 3; i++)
    {
        p += lat[i].getL();
    }
    p /= 2;
    return sqrt (p * (p - lat[0].getL()) * (p - lat[1].getL()) * (p - lat[2].getL()) );
}



Patrat::Patrat() : Figura(4)
{
    cin >> lat[0];
    for(int i = 1; i < 4; i++)
    {
        lat[i] = lat[0];
    }
}

double Patrat::arie()
{
    return lat[0].getL() * lat[0].getL();
}
```

```
int main()
{

    Triunghi t;
    cout << t;
    cout << "Perimetrul triunghiului este: " << t.perimetru() << "\n";
    cout << "Aria triunghiului este: " << t.arie() << "\n";

    Patrat p;
    cout << p;
    cout << "Perimetrul patratului este: " << p.perimetru() << "\n";
    cout << "Aria patratului este: " << p.arie() << "\n";

    return 0;
}
```

### 3.1.2 Versiune cu fișiere multiple

#### 3.1.2.1 Fișierul latura.h

```
#ifndef LATURA_H
#define LATURA_H

#include <iostream>

using namespace std;

class Latura
{

private:
    int lungime;

public:

    Latura();
    Latura(int);

    int getL() const;

    Latura& operator=(const Latura&);
    friend istream& operator>>(istream&, Latura&);

};
#endif
```

#### 3.1.2.2 Fișierul latura.cpp

```
#include "Latura.h"

#include <iostream>

using namespace std;
```

```
Latura::Latura(): lungime(0) {}

Latura::Latura(int lungime): lungime(lungime) {}

int Latura::getL() const
{
    return lungime;
}

Latura&  Latura::operator=(const Latura& l)
{
    if (this != &l)
    {
        lungime = l.lungime;
    }

    return *this;
}

istream& operator>>(istream& in, Latura& l)
{
    cout << "Lungimea laturii este: ";
    in >> l.lungime;
    return in;
}
```

### 3.1.2.3 Fișierul figura.h

```
#ifndef FIGURA_H
#define FIGURA_H

#include <iostream>
using namespace std;

#include "Latura.h"

class Figura
{
protected:
    Latura* lat;
    int nrLat;

public:
    Figura();
    Figura(int);
    Figura(Latura*, int);
    Figura(const Figura&);
    ~Figura();

    friend ostream& operator<<(ostream&, const Figura&);
    Figura& operator=(const Figura&);

    int perimetru();
```

```
};

#endif
```

### 3.1.2.4 Fișierul figura.cpp

```
#include <iostream>
#include "Figura.h"

using namespace std;

Figura::Figura()
{
    lat = NULL;
    nrLat = 0;
}

Figura::Figura(int nr)
{
    nrLat = nr;
    lat = new Latura[nrLat];
}

Figura::Figura(Latura* lat, int nrLat)
{
    this->nrLat = nrLat;
    this->lat = new Latura[nrLat];
    for(int i = 0; i < nrLat; i++)
    {
        this->lat[i] = lat[i];
    }
}

Figura::Figura(const Figura& f)
{
    this->nrLat = f.nrLat;
    this->lat = new Latura[f.nrLat];
    for(int i = 0; i < f.nrLat; i++)
    {
        this->lat[i] = f.lat[i];
    }
}

Figura::~Figura()
{
    delete[] lat;
}

ostream& operator<<(ostream& out, const Figura& f)
{
    out << "Aceasta este figura cu laturile: ";
    for(int i = 0; i < f.nrLat; i++)
```

```
    {
        out << f.lat[i].getL() << " ";
    }
    out << "\n";

    return out;
}

Figura& Figura::operator=(const Figura& f)
{
    if(this != &f)
    {
        delete[] lat;
        this->nrLat = f.nrLat;
        this->lat = new Latura[f.nrLat];
        for(int i = 0; i < f.nrLat; i++)
        {
            this->lat[i] = f.lat[i];
        }
    }
    return *this;
}

int Figura::perimetru()
{
    int suma = 0;
    for(int i = 0; i < nrLat; i++)
    {
        suma += lat[i].getL();
    }

    return suma;
}
```

### 3.1.2.5 Fișierul triunghi.h

```
#ifndef TRIUNGHI_H
#define TRIUNGHI_H

#include "Figura.h"

class Triunghi : public Figura
{
public:
    Triunghi();

    double arie();
};

#endif
```

### 3.1.2.6 Fișierul triunghi.cpp

```
#include "Triunghi.h"
```

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

Triunghi::Triunghi() : Figura(3)
{
    for(int i = 0 ; i < 3; i++)
    {
        cin >> lat[i];
    }
}

// acest constructor este echivalent cu cel de mai sus
/*
Triunghi::Triunghi()
{
    nrLat = 3;
    lat = new Latura[3];
    for(int i = 0 ; i < 3; i++)
    {
        cin >> lat[i];
    }
}
*/

double Triunghi::arie()
{
    int p = 0;
    for(int i = 0; i < 3; i++)
    {
        p += lat[i].getL();
    }
    p /= 2;
    return sqrt (p * (p - lat[0].getL()) * (p - lat[1].getL()) * (p - lat[2].getL()) );
}
```

### 3.1.2.7 Fișierul patrat.h

```
#ifndef PATRAT_H
#define PATRAT_H

#include "Figura.h"

class Patrat : public Figura
{
public:
    Patrat();

    double arie();
};

#endif
```

#### 3.1.2.8 Fișierul patrat.cpp

```
#include "Patrat.h"
#include <iostream>
#include <cmath>
using namespace std;

Patrat::Patrat() : Figura(4)
{
    cin >> lat[0];
    for(int i = 1; i < 4; i++)
    {
        lat[i] = lat[0];
    }
}

double Patrat::arie()
{
    return lat[0].getL() * lat[0].getL();
}
```

#### 3.1.2.9 Fișierul main.cpp

```
#include <iostream>

using namespace std;

#include "Latura.h"
#include "Figura.h"
#include "Triunghi.h"
#include "Patrat.h"

int main() {
    Triunghi t;
    cout << t;
    cout << "Perimetrul triunghiului este: " << t.perimetru() << "\n";
    cout << "Aria triunghiului este: " << t.arie() << "\n";

    Patrat p;
    cout << p;
    cout << "Perimetrul patratului este: " << p.perimetru() << "\n";
    cout << "Aria patratului este: " << p.arie() << "\n";

    return 0;
}
```

## *4 Teme*

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

### *4.1 Pagini*

Se dă următoarea declarație de clase:

```
#include <iostream>

using namespace std;

class Pagina {
private:
    const char* content;
    int pg;  //numarul paginii

public:
    Pagina();
    Pagina(const char* content, int pg);

    friend istream& operator>>(istream&, Pagina&);
    friend ostream& operator<<(ostream&, const Pagina&);

    const char* continut() const;
    int getPg() const;
};

class Carte {
protected:
    Pagina* pagini;
    int npg;  // numar de pagini
    const char* author;

public:
    Carte();
    Carte(const char* author, int size);
    Carte(const Carte&);
    ~Carte();

    friend istream& operator>>(istream&, Carte&);
    friend ostream& operator<<(ostream&, const Carte&);
    Carte& operator=(const Carte&);

    int getNpg() const;
    void print() const;
};

class CarteFictiune : public Carte {
private:
    const char* gen;
```

```
public:
    CarteFictiune(const char* author, const char* gen, int size);

    void print() const;
};

class CarteNonFictiune : public Carte {
private:
    const char* subiect;

public:
    CarteNonFictiune(const char* author, const char* subiect, int size);

    void print() const;
};

int main()
{
    CarteFictiune c1("", "", 0);
    cin >> c1;

    CarteNonFictiune c2("", "", 0);
    cin >> c2;

    cout << c1 << c2;

    c1.print();
    c2.print();

    return 0;
}
```

Implementați aceste clase și testați instrucțiunile din funcția `main()`.