

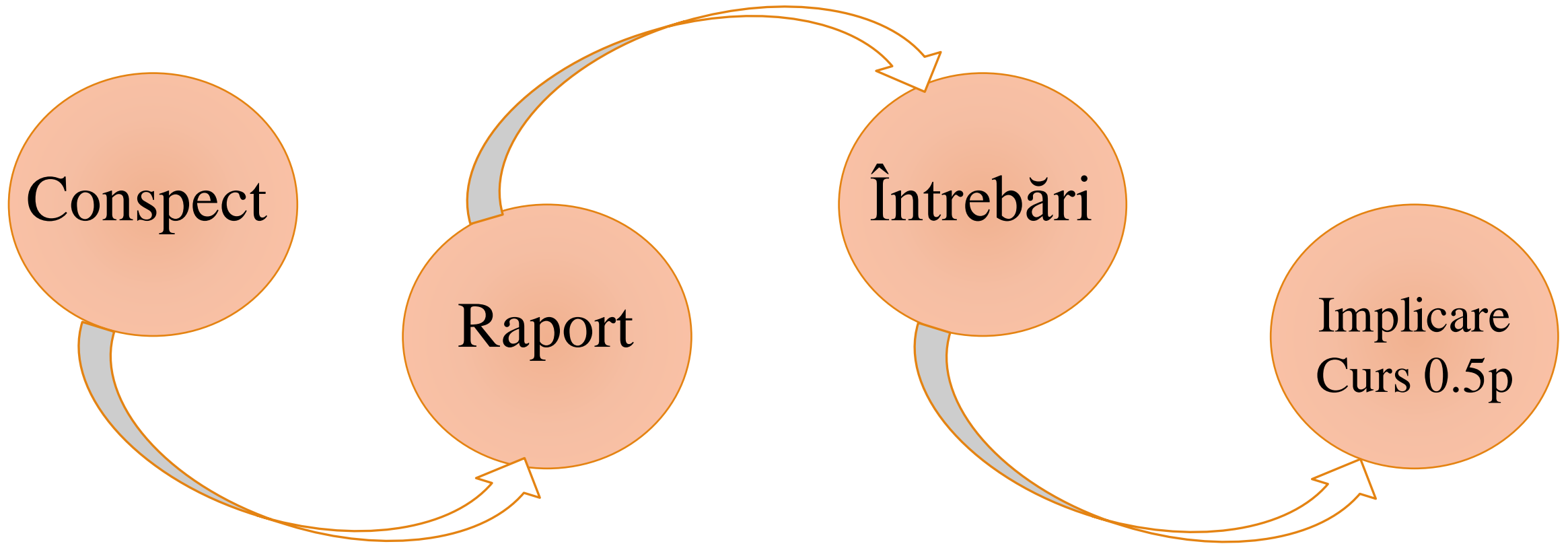
Asistență pentru programarea orientată pe obiecte

Moraru Magdalena

IV. Administrarea modulului

Semestrul	Numărul de ore			Ore de studiu individual	Forma de evaluare	Numărul de credite
	Total	Contact direct				
		Teorie	Laborator			
IV	90	30	30	30	examen	3

Nota pentru fiecare laborator



VII. Studiu individual ghidat de profesor

Materii pentru studiul individual	Produse de elaborat	Forma de evaluare	Termeni de realizare
1. Concepte ale programării orientată pe obiecte			
Clase și obiecte	Proiect individual: Produse program cu utilizarea claselor și obiectelor.	Prezentarea proiectului elaborat. Argumentarea orală.	Săptămâna 7
Conceptele programării orientate spre obiecte	Proiect individual: Produse program cu utilizarea ierarhiilor de clase.	Prezentarea proiectului elaborat. Argumentarea orală.	Săptămâna 14

Noțiuni generale despre cursul POO

- **Scop**

- Acumularea de cunoștințe și aptitudini necesare realizării unor aplicații orientate obiect

- **Obiective**

- Prezentarea și învățarea conceptelor programării orientate obiect
 - Abilitatea de a elabora aplicații orientate obiect

Limbaj de programare folosit pentru ilustrarea conceptelor de programare orientată obiect:

Noțiuni generale despre cursul POO

- **Scop**

- Acumularea de cunoștințe și aptitudini necesare realizării unor aplicații orientate obiect

- **Obiective**

- Prezentarea și învățarea conceptelor programării orientate obiect
 - Abilitatea de a elabora aplicații orientate obiect

Limbaj de programare folosit pentru ilustrarea conceptelor de programare orientată obiect:

Paradigme de programare.

Concepte de bază POO

Paradigme de programare

Programarea nestructurată

Programarea procedurală

Programarea modulară

Programarea orientată obiect

Noțiunea de **paradigmă de programare** se definește ca fiind o metodă de conceptualizare a modului de execuție al calculelor într-un calculator, precum și a modului de structurare și organizare al taskurilor responsabile cu execuția calculelor.

Paradigma este o idee, un set de reguli care precizează modul în care se construiește un program într-un anumit limbaj de programare. Paradigma de programare are o mare influență asupra modului în care se gândește rezolvarea unei probleme.

Noțiunea de **paradigmă de programare** se definește ca fiind o metodă de conceptualizare a modului de execuție al calculelor într-un calculator, precum și a modului de structurare și organizare al taskurilor responsabile cu execuția calculelor.

Paradigma este o idee, un set de reguli care precizează modul în care se construiește un program într-un anumit limbaj de programare. Paradigma de programare are o mare influență asupra modului în care se gândește rezolvarea unei probleme.

Exemplu.

Să zicem că la reședința unei familii se strică încuietoarea de la ușă.



Sotia

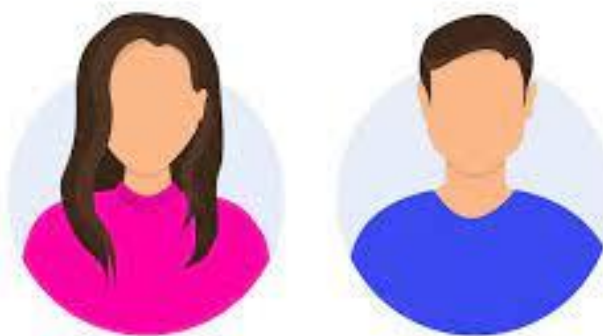


Sot

Exemplu.

Să zicem că la reședința unei familii se strică încuietoarea de la ușă.

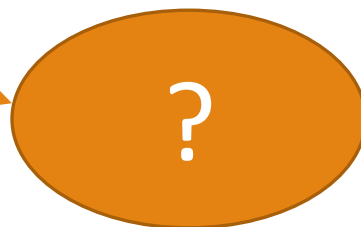
Soția nu are cunoștințe tehnice, așa că soluția ei să cheme un lăcătuș să schimbe încuietoarea.



Soția

Soț

Soțul, pe lângă cunoștințe, are și puțin orgoliu de bărbat și vrea să o repare personal.



Exemplu.

Să zicem că la reședința unei familii se strică încuietoarea de la ușă.

Soția nu are cunoștințe tehnice, așa că soluția ei să cheme un lăcătuș să schimbe încuietoarea.



Soția

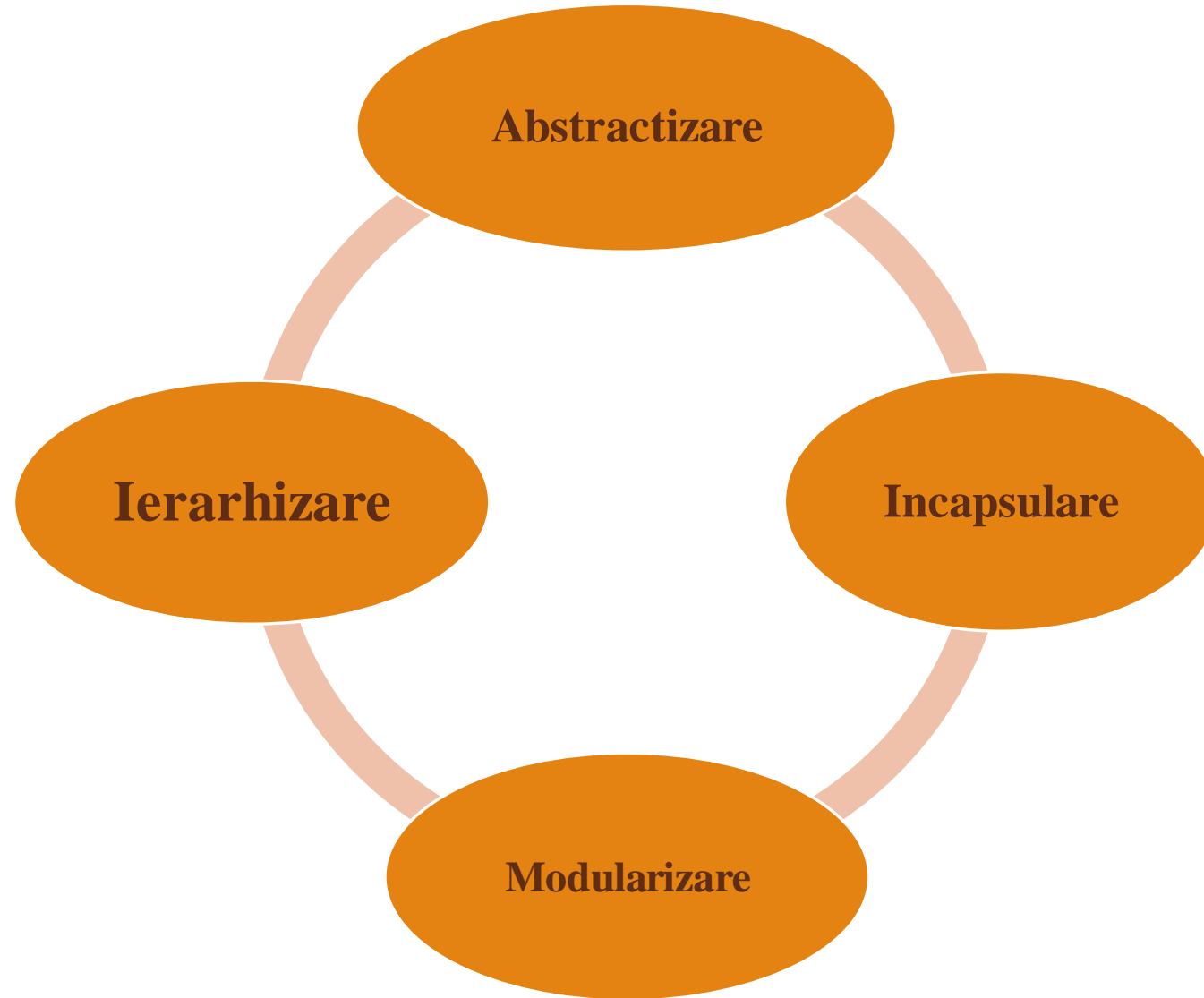


Soț

Soțul, pe lângă cunoștințe, are și puțin orgoliu de bărbat și vrea să o repare personal.

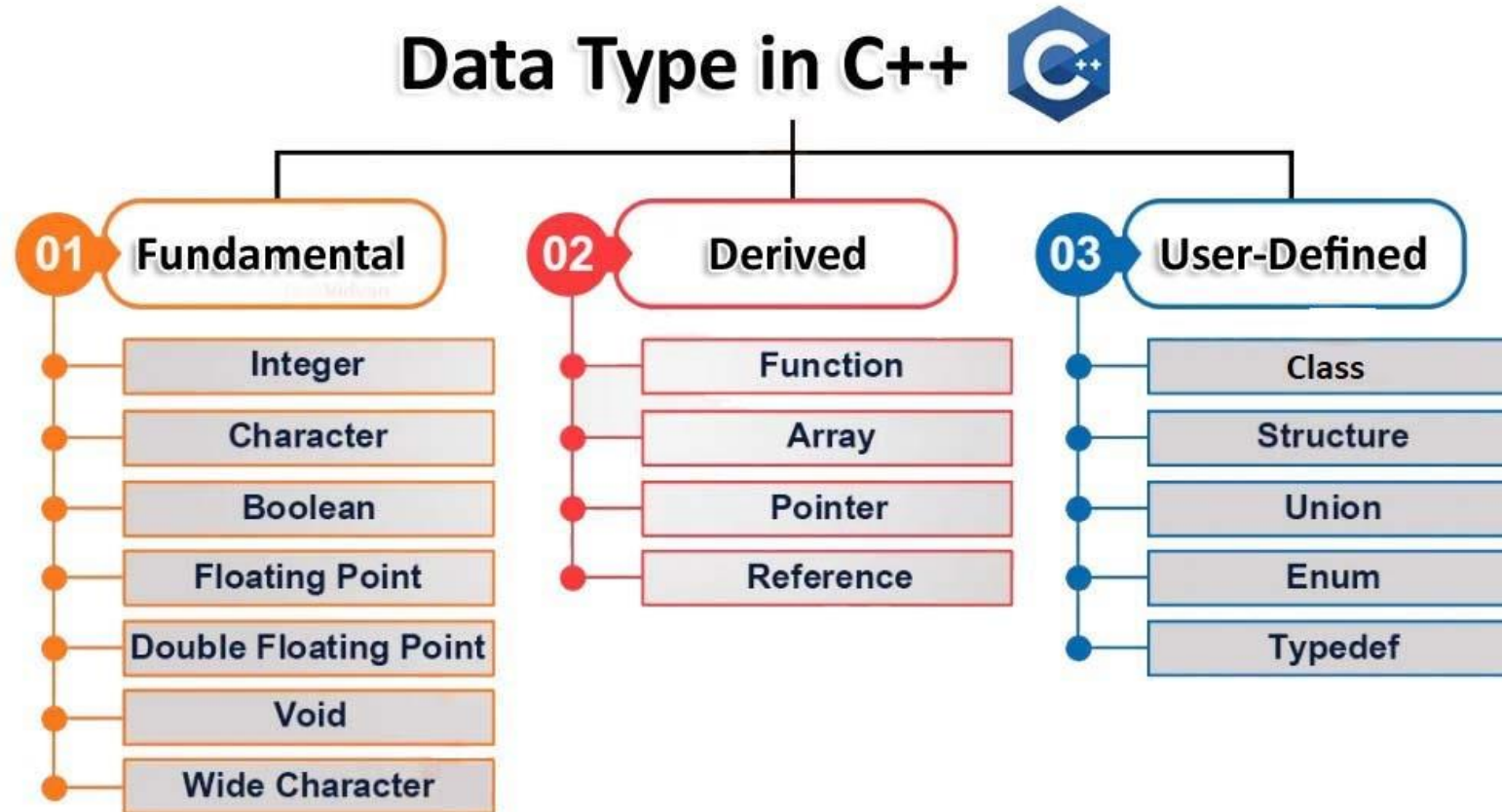
Ambele abordări conduc spre aceeași soluție. Soția și soțul acționează sub paradigme diferite.

Principiile programării orientate pe obiect



Tipuri de date

Tipurile de date în C++ reprezintă categorii de valori pe care o variabilă le poate deține. Acestea definesc modul în care sunt stocate datele și dimensiunea memoriei alocate pentru o anumită valoare.



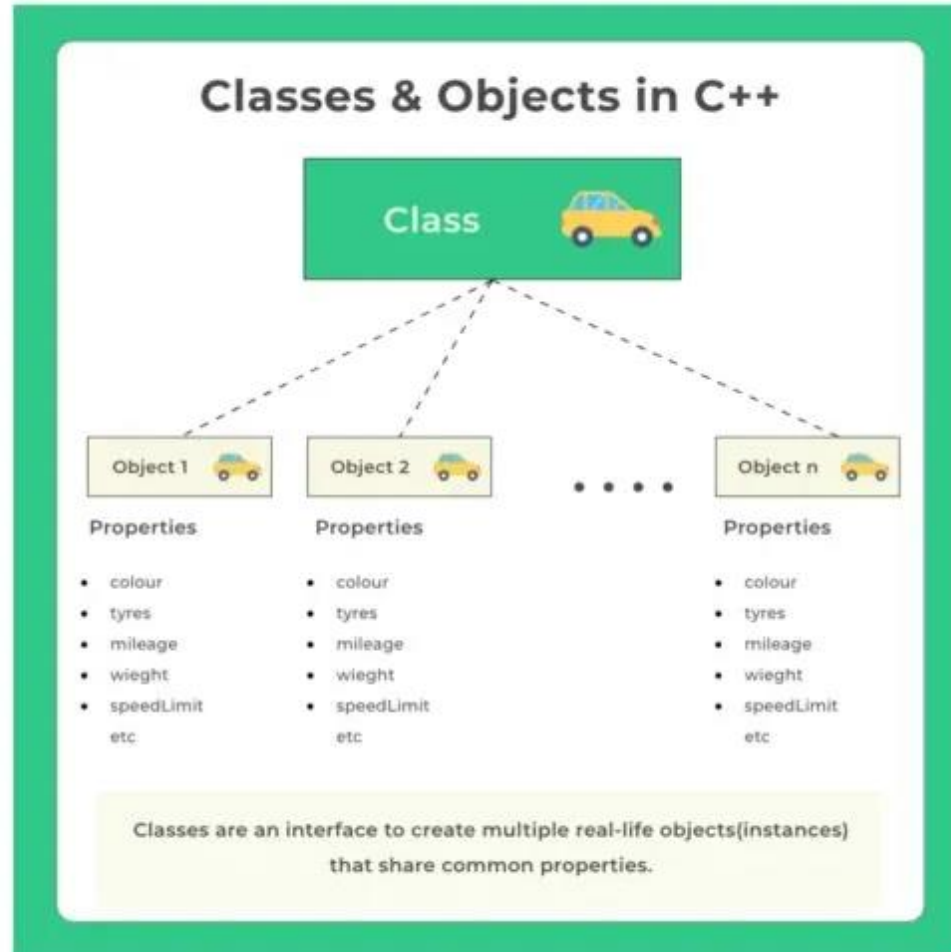
Clase și obiecte

În C++, clasele reprezintă structuri care modelează obiecte din lumea reală, combinând date și comportamente asociate. O **clasă** este o unitate logică care definește:

- **Attribute:** variabile care descriu starea unui obiect, cum ar fi numele, vârsta, sau prețul
- **Metode:** funcții care definesc comportamentul obiectelor, oferind funcționalități precum calculul, manipularea sau afișarea atributelor.

Un **obiect** este o instanță a unei clase, adică o reprezentare concretă a unui tip abstract. Clasele permit programatorilor să definească structuri mai complexe decât tipurile primitive, făcând posibilă modelarea entităților reale precum vehicule, produse sau conturi bancare.

Clase și obiecte



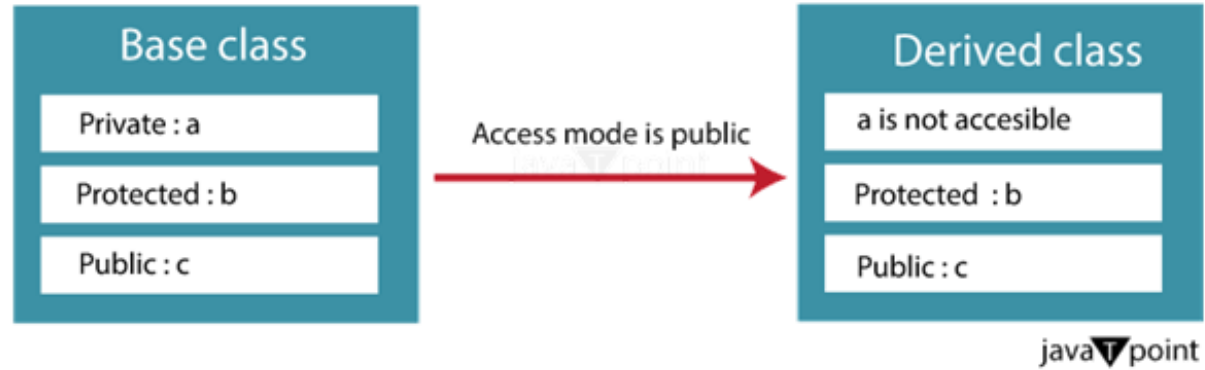
Modificatori de acces

Modificatorii de acces sunt elemente esențiale ale claselor, deoarece controlează vizibilitatea și accesul la atributele și metodele acestora. C++ oferă trei tipuri de modificatori de acces:

- **Private:** Atributele și metodele marcate cu acest modificador pot fi accesate doar din interiorul clasei în care sunt definite. Acestea nu sunt accesibile din exteriorul clasei, ceea ce protejează datele de modificări necontrolate.
- **Public:** Membrii declarați public sunt accesibili din orice parte a programului. Acesta este folosit în special pentru metode care oferă interfața de utilizare a clasei, cum ar fi funcțiile care permit interogarea și modificarea atributelor private.
- **Protected:** Membrii protejați sunt accesibili doar în cadrul clasei și al claselor derivate (prin moștenire). Acest modificador este util în relațiile de moștenire între clase, unde anumite atribute trebuie să fie accesibile pentru clasele derivate, dar să rămână ascunse în afara acestora.

Modificatorii de acces asigură implementarea **încapsulării**, un principiu fundamental al programării orientate pe obiecte, care garantează că datele interne ale unui obiect nu pot fi modificate sau accesate direct fără controlul clasei.

Modificatori de acces



Specifiers	Same Class	Derived Class	Outside Class
<code>public</code>	Yes	Yes	Yes
<code>private</code>	Yes	No	No
<code>protected</code>	Yes	Yes	No

Modificatori de acces

```
class Car {
public:
    // Câmpuri publice
    string make; // Marca mașinii (ex: "Toyota")
    string model; // Modelul mașinii (ex: "Corolla")

    // Constructor pentru a initializa marca și modelul
    Car(string carMake, string carModel) {
        make = carMake;
        model = carModel;
        fuelLevel = 100; // Nivelul de combustibil este setat la 100% implicit
    }

    // Metodă publică pentru a afișa informațiile despre mașină
    void displayInfo() {
        cout << "Marca: " << make << ", Model: " << model << endl;
        cout << "Nivel combustibil: " << fuelLevel << "%" << endl;
    }

    // Metodă publică pentru a conduce mașina (scade combustibilul)
    void drive() {
        if (fuelLevel > 0) {
            fuelLevel -= 10; // Consumă 10% combustibil
            cout << "Conduci la 60 km/h. Nivel combustibil: " << fuelLevel << "%" << endl;
        } else {
            cout << "Combustibil epuizat! Trebuie să alimentezi." << endl;
        }
    }

private:
    // Câmpuri private
    int fuelLevel; // Nivelul de combustibil (0-100%)
};
```

```
int main() {
    // Creăm un obiect Car
    Car myCar(carMake: "Toyota", carModel: "Corolla");

    // Afișăm informațiile despre mașină
    myCar.displayInfo();

    // Conducem mașina
    myCar.drive();

    // Afișăm din nou informațiile după ce am condus
    myCar.displayInfo();

    return 0;
}
```

Constructori

Constructorii sunt funcții speciale ale unei clase care sunt utilizate pentru inițializarea obiectelor. Aceștia sunt invocați automat atunci când un obiect este creat și pot primi parametri pentru a seta valorile inițiale ale atributelor obiectului. Există mai multe tipuri de constructori:

- **Constructor implicit:** nu primește parametri și este utilizat pentru a crea obiecte cu valori predefinite.
- **Constructor parametrizat:** permite inițializarea obiectelor cu valori specifice furnizate la momentul creării.
- **Constructor de copiere:** creează un nou obiect prin copierea valorilor atributelor dintr-un alt obiect existent.

Constructori

```
class Rectangle {  
    public:  
    float length, breadth;  
    Rectangle() {                //Default Constructor  
        length = 10;  
        breadth = 6.5;  
    }  
};
```

```
class Square {  
    public:  
    float side;  
    Square(float s) {  
        side = s;  
    }  
    Square(const Square &obj) {    //Copy Constructor  
        side = obj.side;  
    }  
};
```

```
class Rectangle {  
    private:  
    float length, breadth;  
  
    public:  
    Rectangle(float l, float b) {    //Parameterized Constructor  
        length = l;  
        breadth = b;  
    }  
    float Area(){  
        return length*breadth;  
    }  
};
```

Destructori

Destructorii sunt funcții speciale care sunt apelate automat atunci când un obiect este distrus (eliberat din memorie). Aceștia sunt utilizați pentru eliberarea resurselor alocate de obiecte, precum memoria dinamică sau fișierele deschise. În general, destructori sunt folosiți pentru a curăța resursele și a preveni scurgerile de memorie. Destructorii nu primesc parametri și nu returnează valori.

Constructorii și destructorii sunt fundamentali pentru gestionarea ciclului de viață al obiectelor și pentru prevenirea erorilor legate de alocarea sau eliberarea resurselor.

Destructor



Destructor

```
class MyClass {  
public:  
    // Constructor  
    MyClass() {  
        // Initialization code  
    }  
  
    // Destructor  
    ~MyClass() {  
        // Cleanup code  
    }  
};
```

```
class Test{  
public:  
    Test(){  
        cout << "Test Object Created" << endl;  
    }  
    ~Test(){  
        cout << "Test Object Destroyed" << endl;  
    }  
};
```

Constructor și destructori

Difference Between Constructor & Destructor In C++

Constructor

- Called when the object is created.
- Constructor accepts parameters
- Can be overloaded
- Can be multiple constructors

Destructor

- Called when the object is destroyed or deleted.
- Doesn't accept parameters
- Can't be overloaded
- There's only a single destructor