Tutoriat 1

Introducere

Programarea Orientată pe Obiecte (POO) reprezintă o paradigmă de programare imperativă care are în centru conceptul de ”obiect”. Ea este una dintre cele mai utilizate paradigme din momentul de față. Un cuvânt cheie pe care îl vom folosi de-a lungul lucrului cu obiecte este class.

Class vs Struct

Clasele reprezintă o extindere a structurilor de date. Acestea pot să conțină date, la fel ca structurile, dar în interiorul lor se pot afla și funcții. Noțiunea aceasta de funcții din interiorul unei structuri de date constituie diferența majoră dintre cele două modalități de reprezentare a datelor.

Mențiune: Aici vorbim de struct-ul din C standard. În C++, se pot adăuga funcții în interiorul unei structuri de date obișnuite și nu diferă atât de mult de o clasă.

Încapsularea (Encapsulation)

Primul principiu al programării orientate pe obiect pe care îl vom aborda momentan. Acest principiu este caracterizat de 2 idei principale:

- Toate variabilele și funcțiile sunt înglobate într-o singură structură de date și reprezintă caracteristici ale acesteia;
- Accesul la anumiți membri ai unei structuri de date poate fi controlat.

Având în vedere aceste 2 precizări, trebuie să discutăm despre acel ”acces” la anumite date. În C++, spre deosebire de C, s-a introduc un concept numit *”modificatori de acces” (access modifiers)*.

S-a plecat de la presupunerea că există anumite situații în care nu este indicată permiterea accesului la anumite date, considerate sensibile. De fapt, s-a ajuns la un fel de *”compromis”* în implementarea unei clase, care presupune că toate datele ar trebui ascunse de către cel care scrie clasa, iar anumite metode să rămână vizibile pentru cel care folosește acea structură de date. Astfel, este limitată posibilitatea de manipulare greșită a unor date și se limitează accesul la acestea.

- **Modificatorii de acces din C++:**
    - ***private*** = accesul este limitat la membri clasei (datele și funcțiile private nu pot fi accesate în afara clasei);
    - ***protected*** = accesul este limitat la membri clasei și la subclase (discutăm mai încolo) (datele și funcțiile protected nu pot fi accesate în afara clasei). Pentru moment, putem spune că protected se comportă ca private;
    - ***public*** = accesul este permis oriunde.

      Sintaxa pentru declararea unei clase:

```
class NumeClasa {
    [modificator_de_acces]:
        date;
        metode;
} [numeObiecte de tipul NumeClasa];
```

Modificatorii de acces pot să lipsească într-o clasă, fiindcă cel implicit este ***private***. În cazul unei structuri, modificatorul implicit este ***public***.

Aceasta este diferența majoră între clase și structuri în C++. Clasele mențin datele private, în timp ce în structuri nu se respectă principiul *încapsulării*.

De asemenea, se pot declara variabile de tipul clasei care vor fi globale, la fel ca la structuri, între acoladă și punct și virgulă *(nu vom face acest lucru în general)*.

Exemplu de clasă care respectă încapsularea:

```
class Student {
    private:
        int varsta;
        string nume;
    public:
        void afiseazaNume() {
            cout << nume;
        }
};
```

Astfel, această clasă menține câmpurile de date private și doar metodele rămânând publice. Apare în schimb o altă problemă. *Cum putem accesa datele private?*

- ## Getters and setters

Pentru a putea obține sau modifica datele dintr-o clasă, avem nevoie de 2 tipuri de metode importante pentru a respecta încapsularea: ***getters and setters***.

- ***Getters*** = metode publice care întorc valoarea unei date private pentru a putea avea acces la ea în afara clasei;
- ***Setters*** = metode publice care permit modificarea unei date private din afara clasei.

Exemplu:

```
class Student {
    private:
        int varsta;
        string nume;
    public:
        void afiseazaNume() {
            cout << nume;
```

```
        }

        void setVarsta(int x) {
            varsta = x;
        }

        int getVarsta() {
            return varsta;
        }
        // same for ”nume”
};
```

Până acum, am tot discutat despre accesul la date și cum putem să le menținem ascunse și tot să putem avea acces controlat la ele, dar am omis ilustrarea practică a folosirii unei clase.

- ## Obiect

Cel mai important concept din această paradigmă de programare a rămas mai pe final. Ce este de fapt un obiect?

Definiție formală: Un obiect este o instanță a unei clase.

Definiție informală: Un obiect este o variabilă de tipul unei clase.

Mai exact, în momentul în care creăm o variabilă care primește ca tip de date clasa noastră, atunci am creat un obiect de tipul clasei, adică o instanță.

La fel ca la structuri, pentru a putea accesa ceva din interiorul clasei, avem nevoie de o **instanță**.(acest cuvânt l-am ”bold-uit” pentru că este foarte important și o să vă tot întâlniți cu el).

Exemplu (pentru clasa Student de mai sus):

```
int main() {
    Student s;
    s.setVarsta(25);
    cout << s.getVarsta(); // 25
}
```

- ## Constructori

Acum că am clarificat ce este aceea o instanță, trebuie să observăm cum se creează mai exact aceasta.

În OOP, există o noțiune nouă și importantă: ***constructorii***.

Constructor = o metodă specială fără tip returnat, cu sau fără parametri și cu numele clasei, care este apelat în momentul creării unui obiect (adică la declarare).

Exemplu

```
class Student {
    private:
        int varsta;
        string nume;
    public:
        int getVarsta() {
            return varsta;
        }

        Student(int v) {
            varsta = v;
        }
};

int main() {
    Student s(3);
    cout << s.getVarsta(); // 3
}
```

- **Liste de inițializare**

  Am observat mai sus că, prin intermediul *constructorului*, putem inițializa datele membre ale unei clase.

  Acum o să introducem scurt și această noțiune de ***listă de inițializare*** care reprezintă o altă modalitate de a inițializa un câmp de date în constructor.

  Exemplu:

```
class Student {
    private:
        int varsta;
        string nume;
    public:
        Student(int v) : varsta(v) {}
        Student(string s) {
            nume = s;
        }
        // ambele metode sunt valide
}
```

  Vom observa în lecțiile următoare că această metodă de inițializare este foarte importantă în anumite situații, dar momentan tot ce trebuie să știm e că înlocuiește ce se află între acoladele constructorului.