Tutoriat 1
Noțiuni introductive

1. Cum arată o clasă?

```
class NumeClasă {
        [modificatori de acces]:
        date;
        metode;
} [nume obiecte de tipul
NumeClasă];
```

```
class Person{
private:
   string name;
   double age;
public:
   double getAge(){
      return this->age;
   }
} p1, p2, p3;
```

2. Ce este un obiect?

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

   Ce este concret un obiect?
   Un obiect este o *variabilă* de tipul unei clase.

```
int main() {
   Person p1;
   Person *p2 = new Person();
}
```

3. Struct vs class
   a. struct (C)
      i. nu pot conține și metode
      ii. modificator de acces **public** *by default*
      iii. nu permite moștenirea
   b. struct (C++)
      i. modificator de acces **public** *by default*
      ii. permite moștenirea
   c. class (C++)
      i. modificator de acces **private** *by default*
      ii. permite moștenirea

C++

```
struct PersonStruct{
   string name;
   double age;

   string getName() {
      return name;
   }
}ps;

int main() {
   ps.name = "ana";
   ps.age = 18;
   cout<<ps.name<<" "<<ps.age<<endl;
//afiseaza: ana 18
   cout<<ps.getName(); // afiseaza: ana
}
```

4. Principiile POO
   - **Încapsularea** (Encapsulation)
   - Moștenirea (Inheritance)
   - Abstractizarea (Abstraction)

- Polimorfismul (Polymorphism)

5. Încapsularea
   a. Ce este?
      - Toate variabilele și funcțiile sunt înglobate într-o singură structură de date(clasă).
      - Accesul la anumiți membri ai unei clase poate fi controlat(folosim modificatorii de acces)

   b. Cum se face încapsularea?
      - **Modificatorii de acces din C++:**
        - **private** : datele și metodele NU pot fi accesate din afara clasei
        - **protected**: asemănător cu private, dar mai accesibil (to be continued..)
        - **public**: accesul este permis de oriunde
      - **Getters && setters:**
        - **getters** : metode **public** care întorc valoarea unei date membru **private** în afara clasei
        - **setters** : metode **public** care permit modificarea unei date membru **private** din afara clasei

   c. Exemplu de clasă care respectă principiul încapsulării

```
class Person{
   private:
      string name;
      double age;
   public:
      string getName() {
         return name;
      }

      void setName(string name) {
         this -> name = name;
      }

      double getAge() {
         return age;
      }

      void setAge(double age) {
```

```
        this -> age = age;
    }
};
```

PS: Din asta se pică cel mai ușor la colocviu ☹.

6. Constructori

Constructorul este o **metodă** specială **fără tip returnat** (de obicei este **public**), cu sau fără parametrii și poartă numele clasei, care este apelat în momentul creării unui obiect (adică la declarare).

```
class Person {
        ...
  public:
    Person(const string name, double age) {
      this->name = name;
      this->age = age;
    }
        ...
};
```

Constructorul de copiere(CC):

```
ClassName (const ClassName &obj);
```

Când este apelat CC?

- Când un obiect de tipul clasei este returnat prin valoare
- Când un obiect de tipul clasei este dat ca parametru prin valoare unei funcții
- Când un obiect este construit pe baza altui obiect (Person p, b(p))
- Când compilatorul generează un obiect temporal

**Compilatorul de C++ poate face optimizări și nu va apela mereu CC(mai multe despre copy elision, găsiți aici).*

```
class Person{
private:
```

```
    string name;
    int age;
public:
    Person(string name = "ana", int age = 20): name(name), age(age){}
    Person(const Person & ob){
        this -> name = ob.name;
        this -> age = ob.age;
        cout<<"Copy constructor called"<<endl;
    }

    const string &getName() const {
        return name;
    }

    void setName(const string &name) {
        Person::name = name;
    }

    int getAge() const {
        return age;
    }

    void setAge(int age) {
        Person::age = age;
    }
};
Person returnPerson(Person ob){ return ob;}

int main()
{
    Person p1; // nu afiseaza nimic
    Person p2(p1); // Copy constructor called
    cout<< p2.getName()<< " "<<p2.getAge()<< endl; // ana 20
    Person p3("ion", 21);  // nu afiseaza nimic
    Person p4(p3); //  Copy constructor called
    returnPerson(p3); // afiseaza de doua ori Copy constructor called
    cout<< p4.getName()<< " "<<p4.getAge()<< endl; // ion 21
    return 0;
}
```

7. Liste de inițializare

Listă de inițializare reprezintă o altă modalitate de a inițializa un câmp de date în constructor. Are și alte funcționalități care vor fi detaliate mai târziu.

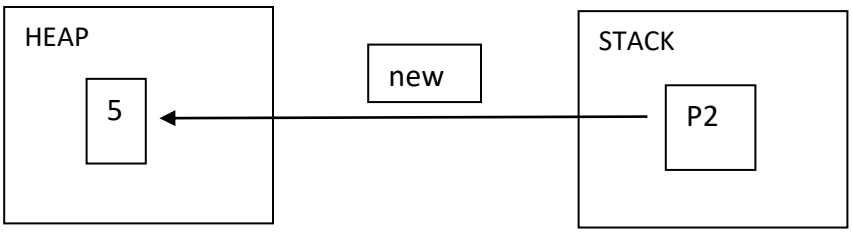
```
class Person {
```

```
        ...
  public:
    Person(const string name, double age) : name(name), age(age) {}
        ...
};
```

8. Pointeri și referințe

Ca să fie mai ușor, pot fi văzute ca tipul de date X* sau X& unde X poate să fie unul dintre tipurile deja definite (int, char, ..) sau un tip nou definit (Person, Animal, Dog..) .

a. Pointeri (*)




```
int main() {
  int *p1= new int;
  int *p2 = new int(5);
  cout<< *p1<< " " <<*p2; // valoarea_aleatoare 5
  delete p2;
  cout<< *p2; // valoare aleatoare
}
```

- Ce este **new**?
  - Keyword care face alocarea dinamică de spațiu pe HEAP.
  - Returnează adresa unde a alocat spațiul în memorie.
- Ce face delete?

  Dezalocă memoria de pe HEAP.

b. Referințe (*)

Extrag si rețin adresa de memorie a unei variabile deja existente.

Nu pot fi considerate variabile noi.

```
int main() {
  int a = 2;
  int &ref = a; // la adresa lui ref pun valoarea a in memorie
  int *p = &a; // in p retin adresa lui a
  int *pp = a; // eroare de compilare
  cout<< *p << endl; // 2
  cout<< ref<< endl; // 2
  cout<< *ref <<endl; // eroare de compilare
}
```

c. Atenție la tipuri

```
int main() {
   int a = 2;
   char c = 'c';
   int * p1 = &a; //2
   char * p2 = &c; // 'c'
   int *p3 = &c; // eroare de compilare
}
```

9. Funcțiile **friend**

Def: Sunt funcții care nu aparțin clasei, definite în afara acesteia, dar care pot accesa membrii privați sau protected ai clasei.

Supraîncarcarea(to be continued...) operatorilor >> și << se face folosind funcții friend, deoarece funcționalitatea acestora este deja definită în biblioteca standard. Pentru a citi sau scrie membrii clasei noastre, numim cele 2 funcții ca fiind friend și le redefinim comportamentul pentru obiectele de tipul clasei noastre.

```
class Person{
   .....
public:
   friend ostream& operator <<(ostream& os, A& ob);//pentru afisare
   friend istream& operator >> (istream& os, A& ob);//pentru citire
   ....
};

istream& operator >> (istream& os, Person& ob)
{
   .....
   return os;
}
ostream& operator <<(ostream& os, Person& ob)
{
   ....
   return os;
}
```

10. **Supraîncarcarea** metodelor în aceeași clasă (Overloading):

Definirea mai multor metode cu același nume în cadrul aceleiași clase. Se face “matching” cu parametrii de la apel(deci aceștia trebuie să difere pentru fiecare metodă în parte). Nu se poate face supraîncarcarea prin metode cu același nume, același tip de parametrii, dar tipuri returnate diferite.

```cpp
class Z{

private:
    int x;
public:
    Z(int x = 2): x(x){}

    void setX(int x){
        this -> x = x;
    }
    void setX(char x){
        this -> x = x;
    }

    int getX() const {
        return x;
    }
};


int main()
{
    Z z;
    z.setX(10);
    cout<<z.getX()<<endl; //10
    z.setX('a');
    cout<<z.getX()<<endl; //97
    return 0;
}
```