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.

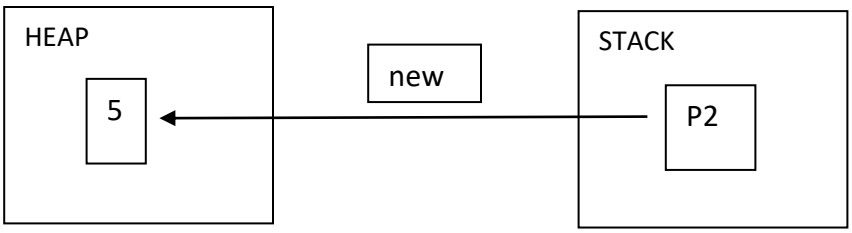
```
class Person {
```

```
        ...
  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;
}
```

Tutoriat 1
Supraîncarcarea operatorilor

1. Operatori unari: -, ++, --, !

```
class A {
private:
    int x;
    bool y;
public:
    A(int x = 2, bool y = true): x(x), y(y){}

    int getX() const {
        return x;
    }

    void setX(int x) {
        A::x = x;
    }

    bool isY() const {
        return y;
    }

    void setY(bool y) {
        A::y = y;
    }

    A operator - (){
        this -> x = - x;
        return A(this -> x);
    }

    A operator ! (){
        this -> y =  !y;
        return A(this -> y);
    }

    A operator ++ (){
        // prefix
        ++ this -> x;
        return A(this -> x, this -> y);
    }

    A operator ++ (int){
        // sufix
```

```
        A copy(this -> x, this -> y);
        this -> x ++;
        return copy;
    }

};
int main()
{
    A a(3, false);
    cout<<a.getX()<<" "<<a.isY()<<endl; // 3 0
    -a;
    cout<< a.getX()<<" "<<a.isY()<<endl; // -3 0
    !a;
    cout<< a.getX()<<" "<<a.isY()<<endl; // -3 1
    ++a;
    cout<< a.getX()<<" "<<a.isY()<<endl; // -2 1
    a++;
    cout<< a.getX()<<" "<<a.isY()<<endl; // -1 1
    return 0;
}
```

2. Operatori binari: +, -, /, *

```
class A {
private:
    int x;
public:
    A(int x = 2): x(x){}

    int getX() const {
        return x;
    }

    void setX(int x) {
        A::x = x;
    }

    // two objects of type A: this and ob
    A operator + (const A& ob) {
        A a;
        a.x = this -> x + ob.x;
        return a;
    }

};
int main()
{
    A a(3), b(5);
    cout<<a.getX()<<endl; // 3
```

```
    cout<<b.getX()<<endl; // 5
    A sum = a + b;
   cout<< sum.getX(); // 8
    return 0;
}
```

3. Operatori relationali: <, > , <=, >=, ==

```
class A {
private:
    int x;
public:
    A(int x = 2): x(x){}

    int getX() const {
        return x;
    }

    void setX(int x) {
        A::x = x;
    }

    // overloaded < operator
    bool operator <(const A& ob) {
        if(this -> x < ob.x){
            return true;
        }

        return false;
    }

};
int main()
{
    A a(3), b(5);
    cout<<a.getX()<<endl; // 3
    cout<<b.getX()<<endl; // 5

   cout<<  (a < b)<<endl; // 1
    //cout<<  (a > b); // eroare (trebuie supraincarcat si > )
    cout<<  (b < a); // 0
    return 0;
}
```

4. Operatorul de asignare =

```
class A {
private:
```

```
    int x;
public:
    A(int x = 2): x(x){}

    int getX() const {
        return x;
    }

    void setX(int x) {
        A::x = x;
    }

    A& operator = (const A& a){
        this -> x = a.x;
        return *this;
    }

};
int main()
{
    A a(3), b(5);
    cout<<a.getX()<<endl; // 3
    cout<<b.getX()<<endl; // 5

   a = b;
    cout<<a.getX()<<endl; // 5
    cout<<b.getX()<<endl; // 5
    return 0;
}
```

5. Input/Output (vezi Tutoriat 1 -> Noțiuni introductive -> funcții friend)

6. Functia call ()

```
class A {
private:
    int x;
public:
    A(int x = 2): x(x){}

    int getX() const {
        return x;
    }

    void setX(int x) {
        A::x = x;
    }
```

```
    // overload function call ()
    A operator()(int e, int f) {
        cout<< " Call () operator overloading"<<endl;
        A a; // x = 2
        cout<<a.x << " "<< e<< " "<< f<<endl;
        a.x =  a.x + e + f; //operatie random
        return a;
    }

};
int main()
{
    A a(3);
    cout<<a.getX()<<endl;

    A c = a(6,7);
    cout<<c.getX()<<endl; // Call () operator overloading
                          // 2  6 7
                          // 2 + 6 + 7 = 15
    return 0;
}
```

7. Operatorul []

```
const int n = 5;

class A {
private:
    int arr[n];

public:
    A() {
        for(int i = 0; i < n; i++) {
            arr[i] = i;
        }
    }

    int &operator[](int i) {
        if( i > n ) {
            cout << "Index out of bounds" <<endl;
            // return first element.
            return arr[0];
        }

        return arr[i];
    }
};

int main() {
```

```
    A a;

    cout << "Value of A[1] : " << a[1] <<endl;  // 1
    cout << "Value of A[3] : " << a[3]<<endl;  // 3
    cout << "Value of A[6] : " << a[6]<<endl;  // Index out of bounds

    return 0;
}
```

8. Opeartorul cast

```
class A
{
    int x;
public:

    A(int val=0): x(val){}

    // Note that conversion-type-id "int" is the implied return type.
    // Returns by value so "const" is a better fit in this case.
    operator int() const
    {
        return this -> x;
    }
};

int main()
{
    A a(10);
    cout << int(a); // 10
    return 0;
}
```

Tutoriat 2
Moștenire și compunere

1. Compunerea

```
class Student {
   ....
};
class Facultate {
   Student s[100]; // compunere
};
```

   a. Ce este?
   Compunerea reprezintă **declararea unui obiect** de tipul unei clase, ca dată membră a altei clase.

2. Moștenirea

   a. Ce este?
      - Al doilea principiu fundamental în POO.
      - Extinderea unei clase, prin crearea altor clase cu proprietăți comune.
      - Se realizează folosind **simbolul :** pus între numele clasei derivate și modificatorul de acces(opțional) alături de numele clasei de bază.

   b. Clasă de bază vs clasă derivată
      - Clasă de bază – cea **din care** se moștenește
      - Clasă derivată – clasa **care** moștenește
      - Obs: Tot ceea ce este **private** în clasa de **bază** devine ***inacesibil*** în clasa derivată.

```
class Animal {
   // clasa de bază
};
class Dog : Animal {
   // clasa derivată
};
```

   c. Tipuri de moșteniri
      i. Moștenire **private**
         - Totul din clasa de bază devine private în clasa derivată.
         - Datele și metodele care sunt deja private în clasa de bază devin inaccesibile în clasa derivată.
         - Tip de moștenire by default când nu este specificat modificatorul de acces la moștenire.
      ii. Moștenire **protected**
         - Tot ce nu e private în clasa de bază devine protected în clasa derivată.
      iii. Moștenire **public**
         - Aceasta este cea mai des folosită moștenire.
         - Totul rămâne la fel ca în clasa de bază.
         - Datele și metodele private tot inaccesibile rămân.

When the component is declared as:	When the class is inherited as:	The resulting access inside the subclass is:
public	public	Public
protected		protected
private		none
public	protected	protected
protected		protected
private		none
public	private	private
protected		private
private		none

d. Constructori la moștenire

- Prima dată, în constructorul din clasa derivată, este apelat constructorul din clasa de bază.
- Se pot apela anumiți constructori din clasa de bază folosind **lista de inițializare.**
- Obs: Este obligatoriu ca în clasa de bază să existe constructorul fără parametrii(dacă nu apelăm explicit alt constructor).

```
class Animal {
    public:
        Animal() {
            cout << "Animal() ";
        }
        Animal(int x) {
            cout << "Animal(x) " << x << " ";
        }
};
class Dog : public Animal {
    public:
        Dog() : Animal(4) {
            cout << "Dog ";
        }
};
int main() {
    Dog d; // Animal(x) 4 Dog
    return 0;
}
```

e. Destructori la moștenire

Destructorii se apelează în ordinea inversă a constructorilor.

```
class Animal {
    public:
        Animal() {
            cout << "C: animal ";
        }
        ~Animal() {
        cout << "D: animal ";
        }
};
class Dog : public Animal {
    public:
        Dog() {
            cout << "C: dog ";
        }
        ~Dog() {
        cout << "D: dog ";
        }
};
int main() {
    Dog d; //C: animal C: dog D: dog D: animal
    return 0;
}
```

f. Moștenirea multiplă

- O clasă poate moșteni mai multe clase în același timp.
- Obs: la moștenirea multiplă, constructorii sunt apelați **în ordinea în care sunt specificați la moștenire**(indifferent de ordinea din lista de inițialziare).

```
class Animal {
    public:
        Animal() {
            cout << "Animal ";
        }
};
class Reptile {
    public:
        Reptile() {
            cout << "Reptile ";
        }
};
class Snake : public Animal, public Reptile {
    public:
        Snake() {
            cout << "Snake ";
        }
};
```

```
int main() {
    Snake s; // Animal Reptile Snake
}
```

3. Mostenire vs compunere

    a. Compunerea: o clasă **are** un obiect de tipul altei clase.
    Ex: Facultatea are mai mulți studenți/profesori/cursuri…
        Firma are mai mulți ingineri/manageri/directori…
    b. Moștenirea: o clasă **este** de tipul altei clase(are câmpuri comune cu aceasta).
    Ex: Animal este câinele/pisica/pasărea....
        Forma geometrică este pătratul/rombul/dreptunghiul...

Tutoriat 3
Const și static

1. Keyword-ul **const**

   a. Ce este?
   Este un *flag(semnal)* dat către compilator care îi transmite să nu permită nicio modificare a valorii unei variabile. Orice **încercare de modificare** a unui const produce o eroare de compilare**.**

   b. La ce folosește?
   Utilizat când avem nevoie ca anumite date să nu poată să fie modificate.

   c. Cum îl folosim?
   O variabilă constantă trebuie **mereu inițializată**, altfel avem o eroare de compilare.

```
const int x = 3;
int const x = 3; // echivalente
```

   d. Pointeri constanți
   Exemplu: pointer constant către un întreg (pointerul nu se poate modifica - adresa de memorie către care pointează nu poate fi modificată, dar își poate schimba valoarea din căsuța de memorie.

```
int* const p = nullptr; // vrea sa fie initializat
int x = 3;
p = &x; // eroare
*p = 5; // ok
```

   e. Pointeri către constante
   Exemplu: pointer către un întreg constant(valoarea din căsuța de memorie **nu** se poate modifica, dar valoarea pointerului - adresa către care arată, se poate modifica).

```
const int* p;
int x = 3;
p = &x; // ok
*p = 5; // eroare
```

f. Diferența dintre cele două:

int * **const** p	**const int*** p

Tip: Te uiți ce tip este lângă keyword-ul **const** (int sau *). Dacă este int atunci întregul este constant, dacă este * atunci pointerul este constant.

g. Date membre constante
   i. Asupra unei date membru constante **NU** se poate aplica **operatorul =** (în afara declarării).
   ii. Se poate inițializa **la declarare** sau prin intermediul **listelor de inițializare**(listele de inițializare sunt *singurele modalități* prin care îi poate fi modificată, ulterior, valoarea unei date membru constante).

```
class A{
    const int x = 2;
    const int y;
public:
    A(int y): y(y){}
    A(int x, int y): x(x), y(y){}

    int getX() {
        return x;
    }

    int getY() {
        return y;
    }
};

int main() {
    A ob1(3);
    A ob2 (5, 6);
    cout<<" Pentru ob1: "<< ob1.getX()<< " "<< ob1.getY()<<endl; // 2 3
    cout<<" Pentru ob2: "<< ob2.getX()<< " "<< ob2.getY()<<endl; // 5 6
    return 0;
}
```

h. Metode constante
   O metodă constantă **NU** are voie să schimbe nimic la datele membre ale pointerului **this**. De aceea, cele mai întâlnite metode constante sunt getters.

   O metodă constantă se definește prin keyword-ul **const** pus între) și {.

   Fără cuvântul cheie **const**, compilatorul va considera metoda **neconstantă**, indiferent dacă modifică sau nu pointerul this.

```
class A{
    ....

   int getX() const { // metoda constanta
      return x;
   }

   int getY() const {
      this -> y = y * 2; // eroare
      return y;
   }

   void mesaj() { // metoda neconstanta (chiar daca nu modifica nimic)
       cout << "Acesta este un mesaj";
   }

};
```

La ce folosesc?
Metodele constante se folosesc pentru a lucra cu **obiecte constante**. Un obiect constant **NU** poate apela o **metodă neconstantă**(pentru că nu îi garantează nimic că nu va încerca să-l modifice în vreun fel), dar un **obiect neconstant**, poate să apeleze oricând o **metodă constantă**.

Adică: obiectul constant – doar metode constante
obiectul neconstant – metode constante și neconstante

```
class A{
    .....

    int getX() const { // metoda constanta
       return x;
    }

    int getY() { // metoda neconstanta (care nu modifica totusi this)
       return y;
    }
};

int main() {
    const A ob1(3);
    A ob2 (5, 6);
    cout<< ob1.getY()<<endl; // eroare: ob1 e const getY() nu este
    cout<<ob1.getX() << endl;  // ok: getX() este const
    cout<< ob2.getY()<< endl // ok: ob2 nu este const
```

```
    return 0;
}
```

i. Return

Valorile returnate de funcții/metode sunt văzute de compilator ca fiind **constante** (dacă **nu** sunt marcate cu **&** la tipul returnat).

```
class Test
{
public:
    Test(Test &) {} // ca sa compileze modificam in Test(const Test &){}
    Test() {}
};
Test fun()
{
    cout << "fun() Called\n";
    Test t;
    return t;
}
int main()
{
    Test t1;
    Test t2 = fun(); // eroare de compilare rezultatul lui fun() este considerat const
    return 0;
}
```

j. Referințe constante

Referința prin definiție presupune că obiectul dat ca parametru la o funcție poate fi modificat în interiorul ei, iar modificările se vor cunoaște și după terminarea funcției.

```
class A{
    int x;
public:
    A(int x): x(x){}
    void modify(A & ob){
        this -> x = ob.x;
       // ob.x = ob.x * 2; // ok
    }
    void notModify(const A & ob){
        this -> x = ob.x;
        //ob.x = ob.x * 2; // eroare
    }
};

int main() {
    A a(2),
```

```
    const A b(3);
    a.modify(b); //eroare
    a.notModify(b); //ok
    return 0;
}
```

Obs: Compilatorul **nu** verifică în interiorul metodei/funcției dacă obiectul chiar este modificat sau nu. El caută **cuvântul cheie** care să garanteze că va rămâne constant și dacă nu îl găsește, întoarce o eroare.

2. Keyword-ul **static**

   a. Ce este?
   O variabilă **statică** se comportă ca o variabilă globală a unei funcții.
   Variabila este inițializată doar **o singură dată** la pornirea programului.

```
void f() {
    static int nr = 0;
    nr++;
    cout << nr << '\n';
}
int main() {
    f(); // 1
    f(); // 2
    f(); // 3
// ...
}
```

   b. Date membre statice

      i. Ce sunt?
      Un membru static este primul inițializat într-o clasă și are aceeași valoare pentru orice instanță a clasei(pratic nu aparține de instanță, ci de întreaga clasă).

      ii. Cum le accesăm din afara clasei (dacă nu sunt private sau protected)?
         1. Prin **operatorul de rezoluție ::**
         2. Printr-o instanță a clasei (este posibil să fie accesate prin orice obiect al clasei declarat în exterior, dar nu pot fi accesate prin this) – nu se reocomandă în practică, dar nu este greșit.

      iii. Statice neconstante

         Cum le inițializăm?

În afara declarației clasei:

```
tip_de_date nume_clasă :: nume_variabilă = valoare_inițială;
```

cu precizarea că dacă **valoarea_inițială lipsește, atunci compilatorul va pune valoarea default. (ex: 0 pentru int)*

sau

În constructorul clasei (în corpul constructorului îi poate fi modificată valoarea, dar **nu și în lista de inițializare**).

Obs: Numai dacă încercăm să accesăm în program o variabilă **statică neinițializată**(adică care **nu** are linia de mai sus în program – cu sau fără valoare_inițială), vom primi o eroare de compilare .

```
class A{
public: // exemplu cu scop didactic
    static int x;
};
int A:: x = 3;
int main() {
    A ob;
    cout<< ob.x<<endl; // ok : 3
    cout<< A:: x << endl; // ok : 3
}
```

iv. Statice constante

1. Cum le inițializăm?
    a. Ca pe cele neconstante
    b. Ca pe o dată membră constantă(doar la declarare sau în corpul constructorului, dar **nu** se poate și prin lista de inițializare din constructor).

```
class A{
public: // exemplu cu scop didactic
    const static int x = 3;
    A(int x): x(x) {} // eroare
};
//const int A:: x = 3;  // ok
int main() {
    A ob;
    cout<< ob.x<<endl; // ok : 3
    cout<< A:: x << endl; // ok : 3
}
```

c. Metode statice

Corpul unei metode statice se poate afla atât în clasă cât și în afara ei.

Orice metodă statică are acces doar la **datele și metodele statice** ale clasei(practic **nu** are pointerul **this**). Orice încercare de a accesa pointerul **this**, în metodele statice, va produce o eroare de compilare.

Accesul se face la fel ca la datele membre statice.

```
class A{
        static int x;
public:
    static int getX() {
        return x;
    }
    static void setX(int x);
};
int A :: x = 3;

void A::setX(int x) {
    A::x = x;
}

int main() {
    A ob;
    cout<< ob.getX()<<endl; // ok : 3
     A:: setX(4) ; // ok
}
```

Tutoriat 4
Overloading și overriding

1. **Supraîncarcarea** (overloading)

   a. Cum se face?
   Prin declararea a 2 sau mai multe metode, în interiorul aceleiași clase, care au **nume identic**, dar antet diferit(**tipul parametrilor diferă/numărul parametrilor diferă**).

   Obs:
   - *Tipul retrunat nu contează*
   - *Supraîncarcarea funcțiilor se poate realiza și în afara claselor(independent de acestea).*

```
class A {
public:
    int f() {
        cout<<"Functia int f()"<<endl;
    }
    void f() {
        cout<<"Functia void f()"<<endl;
    }
};
int main() {
    A a;
    a.f(); // eroare
    return 0;
}
```

```
class A {
public:
    int f() {
        cout<<"Functia int f()"<<endl;
    }
    int f(int x) {
        cout<<"Functia int f(int x)"<<endl;
    }

    int f(double x) {
        cout<<"Functia int f(double x)"<<endl;
    }
```

```
};
int main() {
    A a;
    a.f(); // Functia int f()
    a.f(2); //Functia int f(int x)
    a.f(3.4);//Functia int f(double x)
    return 0;
}
```

b. Ascunderea metodelor (Hiding)

i. Ce face?
Anulează procesul de supraîncarcare.

ii. Când se realizează?
Doar la **moștenirea** claselor.

Obs: Folosim operatorul de rezoluție(**::**) alături de numele clasei de **bază,** dacă dorim apelul metodei din clasa de bază, făcut printr-o instanță a clasei derivate.

```
class Baza {
public:
    void f() {
        cout<<"f() din clasa de baza"<<endl;
    }
};
class Derivata : public Baza {
public:
    void f(int x) {
        cout<<"f(int x) din clasa derivata"<<endl;
    }
};
int main() {
    Derivata d;
    d.f();//eroare
    d.Baza:: f(); //ok -> afiseaza: f() din clasa de baza
    d.f(2); // ok -> afiseaza: f(int x) din clasa derivata

    Baza b;
    b.f(); //ok -> afiseaza: f() din clasa de baza

    return 0;
}
```

c. Operatori care **NU** pot fi supraîncărcați
   - Operatorul * (pointer)
   - Operatorul :: (rezoluție)
   - Operatorul .
   - Operatorul ?:
   - Operatorul sizeof
   - Operatorul typeid

2. **Suprascrierea** (overriding)
   a. Când?
   Suprascrierea se realizează la **moștenirea** claselor.

   b. Ce face?
   Metoda suprascrisă este înlocuită complet de cea care suprascrie.

   c. Cum?
   Metodele clasei **au același nume și aceiași parametri**i.

   Obs: La fel ca la supraîncarcare, folosim operatorul de rezoluție(**::**) alături de numele clasei de **bază**, dacă dorim apelul metodei din clasa de bază, făcut printr-o instanță a clasei derivate.

```
class Baza {
public:
   void f() { cout << "Metoda f() din clasa de baza"<<endl; }
};
class Derivata : public Baza {
public:
   void f() { cout << "Metoda f() din clasa derivata"<<endl; }
};
int main() {
   Derivata d;
   d.Baza::f(); // Metoda f() din clasa de baza
   d.f(); // Metoda f() din clasa derivata
   return 0;
}
```

Tutoriat 4
Upcasting

1. Upcasting

a. Cum se face?

Orice referință/pointer către o clasă **derivată** poate fi convertită într-o referință/pointer către clasa de **bază**.

Adică?

Într-o referință/pointer de tipul clasei de bază rețin adresa către un obiect de tipul clasei derivate:

Clasa_de_bază pointer = new Clasa_derivată();* -> **upcasting dinamic** (cel mai recomandat și frecvent folosit)

```
class Shape {};
class Square : public Shape {};
class Circle : public Shape {};

int main() {
   Shape* s1 = new Square();
   Shape* s2 = new Circle();
}
```

b. Condiția pentru ca upcasting-ul să fie posibil:

Moștenirea să fie de tip **public**.

PS: Apare destul de frecvent un exercițiu cu o astfel de *capcană* la examen.

```
class B{};
class D: private B{};
class C: protected B{};
class A: public B{};
int main(){
   B *p1 = new D(); // error: 'B' is an inaccessible base of 'D'
   B *p2 = new C(); // error: 'B' is an inaccessible base of 'C'
   B *p3 = new A(); //ok
}
```

c. Unde se folosește?
Upcasting-ul este folositor când avem de lucrat cu **vectori de obiecte** care pot fi de mai multe tipuri.

Obs: În exemplu se folosește biblioteca *vector* din STL. Mai multe despre STL puteți afla aici.

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

class Shape {
   friend ostream& operator <<(ostream& os, Shape& ob);
   friend istream& operator >> (istream& os, Shape& ob);
};
istream& operator >> (istream& os, Shape& ob)
{
   cout<<"citeste datele obiectului"<<endl;
   return os;
}
ostream& operator <<(ostream& os, Shape& ob)
{
   cout<<"afiseaza datele obiectului"<<endl;
   return os;
}
class Square : public Shape {

};
class Circle : public Shape {};

int main() {
   int n;
   cout<<"Dati n= ";
   cin >> n;
   vector<Shape*> formeGeometrice; // vectorul din STL
   for (int i = 0; i < n; ++i) {
      char optiune;
      cout << "Patrat sau cerc (P / C): ";
      cin >> optiune;

      if(optiune == 'P'){
            Square* s = new Square();
            cin >> *s;
```

```
            formeGeometrice.push_back(s);
          }
        else {
            Circle* c = new Circle();
            cin >> *c;
            formeGeometrice.push_back(c);
          }
      }
   cout<<"S-a terminat citirea"<<endl;

   for (int i = 0; i < n; ++i) {
     cout<<*formeGeometrice[i]<<endl;
   }
   return 0;
}
```

d. Suprascriere și upcasting

Printr-o referință/pointer de tipul clasei de bază(nu contează daca conține un obiect de tipul clasei derivate --- upcasting sau de tipul clasei de bază) se accesează doar metoda suprascrisă din clasa de bază.

```
class B{
public:
   void print(){cout<<"Print from B::"<<endl;}
};
class D: public B{
public:
   void print(){cout<<"Print from D::"<<endl;}
};

int main(){

   B *upcast = new D();
   B *b = new B();
   D *d = new D();

   upcast->print();// Print from B::
   (*upcast).print(); // Print from B::
   upcast-> D:: print(); //eroare -> 'D' is not a base of 'B'
   b->print(); // Print from B::
   d->print(); // Print from D::
}
```

e. Upcasting static(nerecomandat în practică, dar există)

Clasa_de_bază ob = (Clasa_de_bază) ob_clasa_derivată;

```
class Shape {};
class Square : public Shape {};
class Circle : public Shape {};
int main() {
    Square s;
    Circle c;
    Shape s1 = (Shape) s;
    Shape s2 = (Shape) c;
    return 0;
}
```

Tutoriat 5
Try-catch

1. La ce folosește?

   Tratarea unor erori neprevăzute care apar în timpul execuției unui program. Acestea duc la oprirea neașteptată a acelui program, dacă nu sunt prinse într-un bloc **try-catch**.

   Programul compilează, însă aruncă o eroare la rulare(runtime) și se oprește din execuție la instrucțiunea care o produce.

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

int main(){
    int x = 10, y = 0;
    cout<<x/y;  //eroare la runtime -> impartire la 0
    return 0;
}
```

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

int main(){
    int x = 10, y = 0;
    try {
        if(y ==0) {
            throw runtime_error("0 division");
        }else{
            cout<<x/y;
        }
    }
    catch(...){
        cout<<"A aparut o eroare"<<endl;
    }
    cout<<"A trecut de impartire";
    return 0;
}
```

   Programul va afișa:

   A aparut o eroare
   A trecut de impartire

2. Pașii de execuție pentru un bloc **try-catch**

1. Se execută, pe rând, ce se află în blocul **try**.
2. Dacă se întâlnește o excepție sau este aruncată una(throw) se oprește execuția blocului **try**.
    2.1. Se caută un bloc **catch** care să prindă exact tipul aruncat(NU se pot produce cast-uri implicite ca: int -> double, char -> int etc..) din **try**.
    2.2. Daca acest tip se găsește, se va executa ce se află în interiorul acelui **catch** și se continuă execuția programului după blocul **try-catch**.
    2.3. Altfel, se oprește execuția programului.
3. Dacă nu se întâlnește nicio excepție în **try**, se sare peste blocul **catch.**

Obs: Instrucțiunea *catch(...){…}* prinde orice tip de eroare.

3. Tipuri de date pentru excepții

În C++, poate fi aruncat orice tip de date de la cele deja definite(int, float, string, const char* etc..), la clase definite în program/clase care moștenesc alte tipuri de excepții deja definite:

```
int main(){
    int x = 10, y = 0;
    try {
        if(y ==0) {
            throw "0 division error";
        }else{
            cout<<x/y;
        }
    }
    catch(const char * error){
        cout<<error<<endl;
    }
    return 0;
}
```

Obs: Când aruncați ceva de forma *"String literal"*, tipul de date corespondent este **const char*, NU string.**

throw "String literal" => catch (const char*)
throw string("String literal") => catch (string)

```
class E{
    string message;
public:
```

```
    E(string message): message(message){}

    const string &getMessage() const {
        return message;
    }

};
int main(){
    int x = 10, y = 0;
    try {
        if(y ==0) {
            throw E("0 divison");
        }else{
            cout<<x/y;
        }
    }
    catch(E& e){
        cout<<e.getMessage()<<endl;
    }
    return 0;
}
```

```
class Exception : public runtime_error {
public:
    // Defining constructor of class Exception
    // that passes a string message to the runtime_error class
    Exception()
          : runtime_error("0 division error")
    {
    }
};

int main(){
    int x = 10, y = 0;
    try {
        if(y ==0) {
            throw Exception();
        }else{
            cout<<x/y;
        }
    }
    catch(Exception& e){
        cout<<e.what()<<endl;
    }
    return 0;
}
```

4. Propagarea excepțiilor

Excepțiile se propagă de la o funcție la alta, în funcție de cum sunt apelate acestea.

*Obs: Propagarea prin funcții se oprește la întâlnirea primului bloc **catch** cu tipul potrivit pentru eroarea propagată.*

Programul de mai jos cere introducerea unei valori pentru y până când aceasta este diferită de 0, pentru a efectua împărțirea :

```
class Exception : public runtime_error {
    ....
};

void verify (int y) {
    if (y == 0) {
        throw Exception ();
    }
}
void readY (int& y) {
    cin >> y;
    verify(y); // (apel 2)
}
int main () {
    int x = 10, y;
    while (true) {
        try {
            readY(y); // (apel 1)
            break; // daca NU a aruncat exceptie se excuta break;
        } catch (Exception  e) {
            // a fost aruncata o exceptie, propagate in functii, prinsa in catch
            cout << e.what();
        }
    }
    cout << x / y;
    return 0;
}
```

Tutoriat 5
Virtual, moștenire diamant și downcasting

1. Virtual

a. Ce este?
Virtual este un **keyword** care a apărut pentru a rezolva multe din problemele din C++ legate de moștenire.

b. Când se folosește?
În fața unei metode din clasa de bază(dar și la mostenirea claselor – vezi moștenirea diamant) și înseamnă că dacă acea metoda va fi **suprascrisă(overriding)** într-o clasă derivată, în momentul realizării **upcasting-ului**, metoda apelată va fi **cea din clasa derivată**.

Obs:
- **Virtual** este utilizat, în general, în clasele de bază, pentru că el ajută la moștenire.
- Nu este recomandată folosirea lui într-o clasă care nu urmează să fie moștenită, dar totuși, nu este interzis acest lucru.

```
class A
{
public:
    void f1() { cout << "f1 normal din A"<<endl; }
    virtual void f2() { cout << "f2 virtual din A"<<endl; }
};
class B : public A
{
public:
    void f1() { cout << "f1 din B care suprascrie"<<endl; }
    void f2() { cout << "f2 din B care suprascrie virtual din A"<<endl; }
};
int main()
{
    A *a = new B; // upcasting
    a->f1(); // f1 normal din A
    a->f2(); // f2 din B care suprascrie virtual din A
    return 0;
}
```

c. **Destructor virtual** vs destructor obișnuit(doar la realizarea upcasting-ului)

În mod normal, la moștenire, destructorii sunt *apelați de la clasa derivată spre clasa de bază*. Însă, la **upcasting**, la distrugerea pointerului/referinței prin care se realizează upcating-ul, se va apela **doar** destructorul clasei de bază, cel din clasa derivată rămânând neapelat => *memory leaks*.

Dacă destructorul din clasa de bază este declarat ca fiind și virtual, la distrugerea pointerului/referinței prin care s-a reazlizat upcastingul se vor apela ambii destructori în oridinea corectă, evitând memory leaks.

```
// ------------------- Exemplu normal ------------------
class B {
public:
   ~B() {
      cout <<
          "~B()";
   }
};
class D : public B {
public:
   ~D() {
      cout <<
          "~D()";
   }
};
// ---------------------- Exemplu destructor virtual ------------
class BV {
public:
  virtual ~BV() {
      cout <<
          "~BV()";
   }
};
class DV : public BV {
public:
   ~DV() {
      cout <<
          "~DV()";
   }
};
int main() {
   B *p = new D(); // upcasting
   BV *pv = new DV();
   delete p; // ~B()
   cout<<endl;
   delete pv; // ~DV()~BV()
```

```
    return 0;
}
```

2. Moștenirea diamant

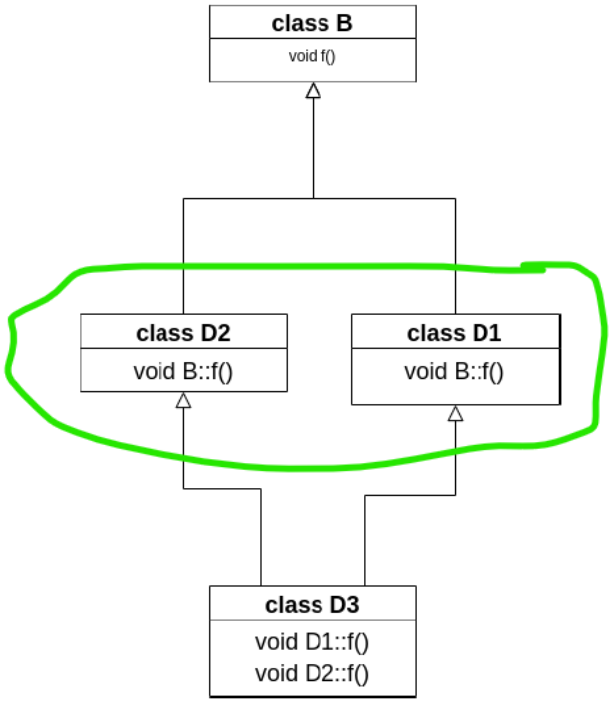
a. Moștenirea virtuală

Este folosită pentru un caz particular de moștenire și anume **moștenirea multiplă**.

Programul de mai jos **nu compliează**, deoarece nu știe care metodă *f()* să apeleze pentru instanța clasei D3(cea venită din D1/ cea venită din D2).

```
class B {
public:
    void f() { cout << "f() din B"; }
};
class D1 : public B {};
class D2 : public B {};
class D3 : public D1, public D2 {};
int main() {
    D3 d3;
    d3.f(); // eroare
    return 0;
}
```

Lanțul de moșteniri pentru programul de mai sus ar arăta așa:



b. Probleme la moștenirea multiplă
În exemplul de mai sus, în momentul moștenirii multiple, clasa D3 va avea acces la metoda *f()*, atât de pe ramura moștenirii clasei D1, cât și pe ramura clasei D2.

c. Soluții pentru rezolvarea problemei diamantului

i. Folosirea operatorului de rezoluție(::) împreună cu numele clasei din care folosim metoda f() moștenită, adăugate la apelul metodei (nu respectă principiile POO, deși este posibilă, **nu** este recomandată).

ii. Folosirea **moștenirii virtuale** (recomandată) : keyword-ul **virtual**(adăugat la moștenirea claselor încercuite cu verde în figura de mai sus) asigură faptul că nu se va copia decât o singură dată metoda *f()* din clasa de bază.

```
class B {
public:
    void f() { cout << "f() din B"; }
};
class D1 : virtual public B {}; //aici
class D2 : virtual public B {}; //aici
class D3 : public D1, public D2 {};
int main() {
    D3 d3;
    d3.f();  // f() din B
    return 0;
}
```

3. Downcasting

a. Ce este?
Reprezintă trecerea de la un pointer/referință de tipul clasei de bază la unul de tipul clasei derivate.

Adică?

Transform un obiect de tipul clasei de bază într-un obiect de tipul clasei derivate.

b. Cum se realizează?
   - Prin intermediul operatorului **dynamic_cast**:

clasa_derivată * pointer = dynamic_cast<clasa_derivată *>(obiect_clasă_de_bază)

!Obs:
- *Instrucțiunea returnează NULL dacă nu se poate face conversia cu succes.*
- *Este necesar ca în clasa de bază sa existe **cel puțin o metodă virtuală**, altfel utilizarea operatorului va duce la o **eroare de compilare**.*
- *Nu se poate realiza downcastingul(compilează, dar valoarea este NULL) dacă pointerul/referința prin care se face nu reprezintă un upcasting.*

```
class Animal {
public:
    void sleep() { cout << "Sleep"<<endl; }
    virtual ~Animal() {}
};
class Dog : public Animal {
public:
    void bark() { cout << "Bark"<<endl; }
};
class Cat : public Animal {
public:
    void meow() { cout << "Meow"<<endl; }
};
int main() {
    Animal* animals[2];
    Animal* a = new Animal();
    cout<< "Downcasting fara upcasting: "<<
dynamic_cast<Cat*>(a)<<endl; // afiseaza valoarea 0 (adica NULL)

    animals[0] = new Cat();
    animals[1] = new Dog();

    for (int i = 0; i < 2; i++) {
        if (Dog* d = dynamic_cast<Dog*>(animals[i])) {
            d->bark();
        } else if (Cat* c = dynamic_cast<Cat*>(animals[i])) {
            c->meow();
        }
    }
```

```
    return 0;
}
```

- Același lucru se poate realiza și prin intermediul **operatorului static_cast<>()**, însă este folosit doar în anumite situații. (mai multe puteți citi aici).

- Downcasting fără metode virtuale și fără operatorul dynamic_cast

  Este posibil doar în cazurile în care știm sigur ce tip de obiect se află pe fiecare poziție a vectorului.

```
class Animal {
public:
    void sleep() { cout << "Sleep"<<endl; }
};
class Dog : public Animal {
public:
    void bark() { cout << "Bark"<<endl; }
};
class Cat : public Animal {
public:
    void meow() { cout << "Meow"<<endl; }
};
int main() {
    Animal* animals[2];
    animals[0] = new Dog();
    animals[1] = new Cat();
    for (int i = 0; i < 2; ++i) {
        if (i == 0) {
            Dog* d = (Dog*)animals[i]; // downcast
            d->bark(); // correct
        } else {
            Cat* c = (Cat*)animals[i]; // downcast
            c->meow(); // correct
        }
    }
    return 0;
}
```

Tutoriat 6
Polimorfism

1. Ce este?
   Abilitatea unui obiect de a lua mai multe forme.

2. Cum se realizează?
   - **supraîncarcare/suprascriere** de metode
   - **supraîncarcare** de operatori
   - metode **virtuale**
   - **template**

3. Tipuri
   - Polimorfism la compilare (compile time polymorphism/**late binding**)
     Se realizează prin 2 modalități:
     - ✓ **supraîncarcare** de metode
     - ✓ **supraîncarcare** de operatori
     - ✓ **template**

     Obs:
     - ❖ Supraîncarcarea se poate realiza atât în clasă, cât și în afara acesteia.
     - ❖ Vezi operatori care nu pot fi supraîncarcați în Tutoriat 4- *Suprascriere și supraîncarcare.pdf*
     - ❖ Vezi supraîncarcarea operatorilor în Tutoriat 1 - *Supraîncarcarea operatorilor.pdf*

   - Polimorfism la rulare (run time polymorphism/**early binding**)
     Se realizează prin 2 modalități:
     - ✓ **suprascriere** de metode
     - ✓ metode **virtuale**

4. **RTTI**(Run Time Type Information)
   a. Ce este?
      Un mecanism care expune informații despre tipul unui obiect la execuție.

   b. Când se folosește?
      Este disponibil doar pentru clasele care conțin **cel puțin o metodă virtuală**.

   c. Exemplu

dynamic_cast: Deduce tipul real de la upcasting(dreapta egalului) al obiectului la execuție(are nevoie de cel puțin o metodă virtuală în clasa de bază).

Tutoriat 6
Abstractizare

1. Ce este?
   Ascunderea detaliilor implementării față de utilizatorul unei clase.

2. Exemple din practică
   Ex1: Când includem o bibliotecă, folosim funcțiile/metodele din acea bibliotecă, fără a fi nevoie să cunoaștem implementarea acestora.

   Ex2: Când scriem o clasă într-un fișier de tip header(.h), în programul principal doar apelăm metodele publice, fără a fi nevoie să cunoaștem implementarea acestora.

3. Metode pur virtuale
   Sunt metode virtuale fără implementare.

```
virtual void nume_metodă() = 0;
```

4. Clase abstracte
   Clase care conțin **cel puțin o metodă pur virtuală**.

   Obs:
   - **Nu** pot fi instanțiate.
   - Clasele lor derivate pot fi instanțiate doar dacă au fost **implementate toate metodele pur virtuale.**

5. Clasă abstractă vs interfață(în C++):
   Interfața are **doar metode pur virtuale**, spre deosebire de clasele abstracte care pot avea și alte tipuri de metode, pe lângă cele pur virtuale.

```
class MyInterface
{
public:
    // Empty virtual destructor for proper cleanup
    virtual ~MyInterface() {}

    virtual void Method1() = 0;
    virtual void Method2() = 0;
};


class MyAbstractClass
```

```
{
public:
    virtual ~MyAbstractClass();

    virtual void Method1();
    virtual void Method2();
    void Method3();

    virtual void Method4() = 0; // make MyAbstractClass not instantiable
};
```

6. Exemplu abstractizare

```
class Animal {
public:
    virtual void eat() = 0;
    virtual void sleep() = 0;
};
class Dog : public Animal {
public:
    void eat() {
        cout << "Dog::eat()";
    }
    void sleep() {
        cout << "Dog::sleep()";
    }
    void bark() {
        cout << "Dog::bark()";
    }
};
class Cat : public Animal {
public:
    void eat() {
        cout << "Cat::eat()";
    }
    void sleep() {
        cout << "Cat::sleep()";
    }
    void meow() {
        cout << "Cat::meow()";
    }
};
int main() {
    Animal a; // eroare de compilare => Animal este clasa abstracta=> nu poate fi instantiata
    Dog d; // corect
    Cat c; // corect
    Animal* a1 = new Dog; // corect (se declara un pointer de tipul clasei abstracte, dar se intasntiaza un obiect in memorie de tipul clasei neabstracte)
```

```
    Animal* a2 = new Cat; // corect
    return 0;
}
```

Tutoriat 6
Template

1. Ce este?
   Presupune scrierea unei singure clase/funcții a cărei comportament este asemănător și se modifică, doar dacă se modifică și un anumit tip de date.

   Obs:
   Template este o formă de **polimorfism** la compilare.

2. La ce se folosește?
   Template este un instrument prin care se poate evita rescrierea unor blocuri de cod.

3. Despre **typename**
   Poate fi înlocuit începând cu standardul C++17 cu **class(cel folosit în definirea template)** fără vreo diferență semnificativă.

4. Funcții template
   a. Pași la compilare:

      1. Când e întâlnită o funcție template, este compilată, fără a se ține cont de tipul de date necunoscut.
      2. În momentul în care se apelează o funcție template, compilatorul creează o nouă funcție obișnuită în care tipul de date necunoscut este înlocuit cu cel specificat în apel.

      f<int>(3) ===> void f (int x) {...}
      f<char>('c') ===> void f (char x) {...}

```
template <typename T>
void f (T x) {
    cout << "Functie template"<<endl;
}
void g (int x) {
    cout << "Functie obisnuita"<<endl;
}
int main () {
    f<int>(3); // Functie template
    f<char>('c'); // Functie template
    g(2); // Functie obisnuita
```

```
    return 0;
}
```

b. Specializarea funcțiilor template

```
template <typename T>
void f (T x) {
    cout << "Functie template"<<endl;
}
template <>
void f (int x) {
    cout << "Functie  specializata"<<endl;
}
int main () {
    f(3); // Functie  specializata
    f<int>(3); // Functie  specializata
    f('c'); // Functie template
}
```

c. Prioritatea la supraîncarcare(overloading)

Cum procedează compilatorul când caută o funcție care se potrivește cu un apel:

1. Se caută o **funcție normală** care să aibă parametrii potriviți.
2. Dacă nu s-a găsit la punctul 1, se caută o **specializare** cu parametrii potriviți.
3. Dacă nici punctul 2 nu a furnizat un rezultat, se caută **o funcție template** cu numărul de parametrii potriviți.
4. Dacă nici punctul 3 nu a furnizat un rezultat, se întoarce o **eroare la compilare.**

Mai pe scurt, o ordine de prioritate ar fi:

1. Funcțiile normale
2. Funcțiile specializate
3. Funcțiile template

```
template <typename T>
void f (T x) {
    cout << "Functie template"<<endl;
}
template <>
void f (int x) {
    cout << "Functie specializata"<<endl;
}
void f (char c) {
```

```
    cout << "Functie normala (char)"<<endl;
}
void f (int c) {
    cout << "Functie normala (int)"<<endl;
}
int main () {
    f(3); // Functie normala (int)
    f('c'); // Functie normala (char)
    f<int>(3); // Functie specializata
    f<char>('c'); // Functie template
}
```

5. Clase template
    a. Exemplu

Obs:
Spre deosebire de funcțiile template, aici este obligatorie specificarea tipului de date la declararea unui obiect (între <>).

```
template <typename T>
class A {
// clasa template
    T x;
    int y;
public:
    A(){cout<<"A"<<endl;}
};
class B {
// clasa obisnuita fara template
    char x;
    int y;
public:
    B(){cout<<"B"<<endl;}
};
int main () {
    A<int> a1; // A
    A<char> a2; // A
    B b; // B
    return 0;
}
```

    b. Specializarea claselor template

```
template <typename T>
class A {
    T x;
```

```
public:
    A() { cout << "A template"; }
};
template <>
class A<int> {
    int x;
public:
    A<int>() { cout << "A specializata "; }
};
int main () {
    A<int> a1; //A specializata
    A<char> a2; // A template
}
```

c. Metode template

```
template <typename T>
class A {
    T x;
public:
    T getX () const;
    void setX (T);
};
template <typename T>
T A<T>::getX () const {
    return x;
}
template <typename T>
void A<T>::setX (T _x) {
    x = _x;
}
int main(){
    A<int> object;
    object.setX(2);
    cout<<object.getX(); // 2
}
```

Tutoriat 6
Singleton

1. **Design patterns**
   Sunt soluții tipice pentru probleme comune în dezvoltarea de software. Acestea sunt independente de limbaj, pentru că se referă mai mult la proiectarea unor clase, decât la modul în care acestea sunt implementate. Adică, un design pattern din C++ este același și în Java, cu diferențe minore de implementare.

   Tipuri:
   - Creational Patterns
   - Structural Patterns
   - Behavioral Patterns

2. **Creational Patterns**
   Oferă diverse mecanisme de creare a obiectelor. Astfel, este mai ușoară întreținerea codului care poate fi refolosit mai târziu.

3. **Singleton**
   a. Ce este?
      - design pattern creațional
      - permite crearea unei clase care are în *permanență* o **singură instanță**

   b. La ce folosește în practică?
      - menține o singură conexiune activă la o bază de date
      - menține un singur flux deschis la un fișier comun mai multor procese
      - menținerea unei singure instanțe pentru **meniul** unui joc/unei aplicații

   c. Cum se implementează?
      1. Se creează un **constructor privat** pentru a împiedica crearea de noi instanțe prin keyword-ul *new.*
      2. Se creează un câmp de date **static** care va fi un **pointe**r către un obiect de **tipul clasei** și care va reprezenta instanța unică a acelei clase.
      3. Se creează o metoda **statică** care apelează **constructorul privat** dacă nu a fost creată deja o instanță a clasei.

```
class Singleton {
private:
   static Singleton* instance;
   Singleton() {
      cout << "Constructor called";
   }
public:
   static Singleton* getInstance() {
      if (instance == NULL) {
         instance = new Singleton;
      }
      return instance;
   }
};
Singleton* Singleton:: instance;

int main () {
   Singleton *s1;
   s1 = Singleton::getInstance(); // Constructor called
   Singleton *s2; //pointer, nu are o zona de memorie catre care arata la
linia asta
   s2 = Singleton::getInstance(); // nu intra in constructor
//    Singleton s3; // constructorul e privat => eroare
//    Singleton *s4 = new Singleton();  // constructorul e privat => eroare
}
```

d. Particularizare
O clasă care permite crearea a maxim 3 instanțe.

```
class Singleton3{
private:

   static Singleton3* _instance[3]; // declaring 3 pointers to Singleton3
   static int count; // the current number of instances created

   Singleton3(){cout<<"Constructor called"<<endl;}

public:
   static Singleton3* getInstance(){
      if (count < 3){
         _instance[count] = (Singleton3*) new Singleton3();
         count++;
         return (Singleton3*)(_instance[count - 1]);
      }
      else {
         return nullptr;
      }
```

```
    }

    ~Singleton3(){
        delete  [] *_instance;
    }
};
Singleton3* Singleton3 :: _instance[] = { nullptr,nullptr,nullptr };
int Singleton3 :: count = 0;

int main() {
    Singleton3* objArray[3];

    for (int i = 0; i < 3; i++){
        // Create an instance
        objArray[i] = Singleton3::getInstance();
    }

    // Attempt to create object no 4
    Singleton3* obj4 = Singleton3::getInstance();
    if (obj4 == nullptr) {
        cout << "Error of creating obj4." << endl;
    }

    return 0;
}
```