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. Unde le folosim?
      Doar în **metode statice** și în **corpul constructorilor**.

      iii. Cum le accesăm?
         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.

iv. 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
}
```

v. 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
}
```