

# **Programare orientata pe obiecte**

## **Cursul 2 - Deosebiri intre C si C++**

# C2: Deosebiri intre C si C++

## 0. Introducere

- **\*.c**      =>      **\*.cpp**

d.p.d.v. al fisierelor

- **c**                      **cplusplus**

d.p.d.v. al compilatorului



*Exista medii de dezvoltare care au un compilator unic pentru programele scrise in C si C++ (Borland C) si altele care au compilatoare distincte si trebuie informate pe care sa il foloseasca (Dev-C++)!*

## C2: Deosebiri intre C si C++

### 1. Operatii de intrare/iesire (citire/scriere) de la/pe consola in C++ :

- functiile printf, scanf ⇔ cout, cin; <<, >> (obiecte si operatori de insertie si extractie)
- trebuie inclusa biblioteca: iostream si folosit namespace-ul std

**cin >> variabila;** // de la obiectul consola (cin) se citeste (>>) valoarea variabilei

**cout << variabila;** // pe obiectul consola (cout) se afisaza (<<) valoarea variabilei

- variabila poate sa fie de orice tip de date de baza sau sir de caractere

#### Exemple:

```
int x, a; double b;  
char sir[20];
```

```
cin >> x;
```

```
cin >> a >> b >> sir;
```

```
cout << "textul: " << sir;  
cout << "\n";
```

```
cout << "x=" << x << endl;
```

```
cout << "a=" << a << endl  
<< "b=" << b;
```

# C2: Deosebiri intre C si C++

## 2. Comentarii

**in C :**

```
/* citire data */  
getdate(&data);  
sunet();  
cout<<" Azi suntem in:"<<data.zi<<" " <<data.luna<< " " <<data.an;  
/* deschidere fisier  
fhandler=fopen("C:\\DATE\\REMEMBER.DAT","a");  
fclose(fhandler);*/
```

**in C++ :**

```
int stare_Panel = 0;  
// butoane (8): Scenariu, Salvare, Configurare, Declansare, Analiza,  
// Adnotari, Ajutor, Iesire  
// stari (4): Initiala, Declansare, Analiza, Simulare  
int StariButoane[4][8] =  
{{0,1,1,0,0,0,0,0},{1,1,0,1,0,0,0,0},{0,1,0,1,1,0,0,0},{1,1,1,1,1,1,1,1}};  
// int StariButoane[2][3] = {{1,2,3},{4,5,6}};  
// = {{0,1,1,0,0,0,0,0},{1,1,0,1,0,0,0,0},{1,1,0,1,0,0,0,0}};
```



*Daca linia este prea lunga si se trece automat pe linia urmatoare si comentariul "nu mai are efect" => se numesc comentarii de linie!*

## C2: Deosebiri intre C si C++

### 3. Pozitia declaratiilor in program

**in C :**

la inceput

**in C++ :**

oriunde in program



*Dar mereu inainte de a fi folosite!*

**in C :**

la inceput

**in C++ :**

oriunde in program

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

```
int x=3;
```

```
int f(int x)
{
int rezultat=0;
int dublu=0;
rezultat=x*x;
dublu=rezultat*2;
return dublu;
}
```

```
int main(int argc, char *argv[])
{
int rez=f(x);
cout<<x;

return 0;
}
```

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

```
int f(int x)
{
int rezultat=0;
rezultat=x*x;
int dublu=rezultat*2;
return dublu;
}
```

```
int x=3;
```

```
int main(int argc, char *argv[])
{
int rez=f(x);
cout<<x;

return 0;
}
```

*Avantaj: se face declararea acolo unde este nevoie de variabila -> readability*

# in C++ :

oriunde in program

```
float sum(float *v, int n)
{
    float s=0;
    for (int i=0; i<n; i++)
        s+=v[i];
    return s;
}
```



*Atentie la durata de viata ("scope") a variabilei respective:*

// vizibilitate 1

```
float sum(float *v, int n)
{
    float s=0;
    for (int i=0; i<n; i++)
        s+=v[i];
    cout<<i++; //eroare; variabila nu
return s;      //mai exista
}
```

// vizibilitate 2

```
if(m>23.2)
{
    double dv=-1;//...
}
else
{ //.....
    dv = 3.2; //eroare; variabila nu
    //..... //mai exista
}
```

## C2: Deosebiri intre C si C++

### 4. Structurile si uniunile

**C:**

```
typedef struct {  
    int x;  
    int y;  
} point;
```

**C++:**

```
struct point {  
    int x;  
    int y;  
};
```

*Atat structurile cat si uniunile pot fi declarate fara typedef =>*

*=> sunt tipuri de date recunoscute automat*



*Campurile sunt implicit **publice** atat in C cat si in C++*



**Observatii:** Cand se folosesc uniuni se presupune ca, la un moment dat, doar un camp este folosit -> toate campurile declarate ocupa acelasi spatiu.

Uniunile sunt utile pentru a stoca ceva ce poate sa fie de mai multe tipuri.

O structura are campurile stocate in locatii separate de memorie si toate campurile pot sa fie folosite in acelasi timp.

```
union uniune { int a; char b; }; //sizeof(uniune) = max sizeof(tip_camp)
```

```
struct structura { int a; char b; }; //sizeof(structura) =  $\sum$ sizeof(tip_campi)
```

```
uniune x;
```

```
x.a = 3; // sau x.b = 'c';
```

```
// NU pot sa le stochez simultan; valoarea lui x.b s-ar suprascrie in spatiul ocupat de x.a
```

```
structura y;
```

```
y.a = 3;
```

```
y.b = 'c';
```

Cum declar o variabila de tip point cu campurile x si y – coordonatele punctului?  
Cum fac modificarea si afisarea ei folosind functii? Cum arata in memorie?

```

#include <iostream>
using namespace std;

struct point{
    int x,y;
};

void modificare(point *p, int xx, int yy){ //se da ca param. un pointer catre var. de modificat
//    p->x=xx;
//    p->y=yy; //sau
    (*p).x=xx; // se lucreaza direct la adresa variabilei date ca param.
    (*p).y=yy; //si modificarile se pastreaza
}

void afisare(point p){
    cout<<p.x<<" "<<p.y;
}

int main(int argc, char** argv) {
    point p;
    modificare(&p,2,2);
    afisare (p);
    return 0;
}

```

# C2: Deosebiri intre C si C++

## 5. Functii ca parti ale unor structuri (functii membre)

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

struct point {
    int x; // x, y implicit publice
    int y; //<=>accesibile de oriunde
           //din program
    void modificare(int xx,int yy)
    {
        x=xx;//x si y ale oricarei variabile
        y=yy;//de tip point care apeleaza
    }           //aceasta functie

    void afisare()
    {
        cout<<x<<" "<<y<<endl;
    }
};
```

```
int main(int argc, char *argv[])
{
    point p={ 1,2};
    p.afisare(); // cout<<p.x<<" "<<p.y<<endl;
    p.modificare(3,3); // p.x=3;p.y=3;
    p.afisare();

    // in momentul apelului, aceste functii primesc
    // adresa variabilei (p) si au acces la campurile
    // acesteia

    //similar cu ce se intampla in exemplul
    //precedent, numai ca variabila cu care se
    //lucreaza nu e data ca param, ci stie sa
    //foloseasca o functie asociata tipului ei

    return 0;
}
```



- denumite si **functii membre** sau **metode**; implicit **publice**
- functiile membre **nu** cresc dimensiunea tipului de date
- **acces direct la campurile structurii (chiar daca sunt declarate private)**
- **utilitatea acestor functii se va vedea in cazul programarii obiectuale.**

```

#include <cstdlib>
#include <iostream>
using namespace std;

struct point {
    private:    int x; //tot ce urmeaza dupa
                int y; //private nu e vizibil
                //decat intre { }

    public: //se schimba vizibilitatea
    // functiile vor fi vizibile de oriunde
    void modificare(int xx, int yy)
    {
        x=xx;
        y=yy;
    }

    void afisare()
    {
        cout<<x<<" "<<y<<endl;
    }
} ;//pana aici sunt campurile x si y accesibile

```

```

int main(int argc, char *argv[])
{
    // point p={1,2}; //nu mai functioneaza
    //atribuirea directa de valori pentru x si y
    //deoarece nu sunt accesibile din aceasta zona
    //de cod

    point p;
    p.modificare(1,2);
    p.afisare();

    // cout<<p.x<<" "<<p.y<<endl; //nu mai am
    //acces la x si y

    p.modificare(3,3);

    // p.x=3;p.y=3; ; //nu mai am acces la x si y

    p.afisare();

    return 0;
}

```

De ce/cand as folosi private?

Ce se intampla daca functiile nu erau declarate public?

# C2: Deosebiri intre C si C++

## 6. Operatorul de rezolutie “::”

*Permite accesul la un indicator cu domeniu fisier, dintr-un bloc in care acesta nu e vizibil.*

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

```
int x=0; // x global
void funct(int val)
{
    int x = 0; // x local
    ::x = val; // ::x – referire la variabila globala x acoperita ca
    cout<<x; // vizibilitate (shadowed) de cea locala cu acelasi nume
    x = ::x;
    cout<<x; // aici :: se refera la zona de cod (domeniul) fara
    } // nume, imediat in afara functiei funct
```

```
int main(int argc, char *argv[])
{
    cout<<x;
    funct(4);
    cout<<x;

    return 0;
}
```

**//ce se afisaza?**



*De fapt acest operator a fost introdus pentru alt scop!*

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

```
struct point {
private:  int x;//altfel ar fi publice
         int y;

//definesc functiile (specific nume,
//tip returnat, semnatura)
public:  void modificare(int ,int );
         void afisare();
};
```

```
//implementez functiile
void point::modificare(int xx,int yy)
{
    x=xx;
    y=yy;
}
```

```
void point::afisare() //functia afisare din
// domeniul fisier cu numele point
```

```
{
    cout<<x<<" "<<y<<endl;
}
```

```
int main(int argc, char *argv[])
{
    point p;
    p.afisare();
    p.modificare(3,3);
    p.afisare();

    return 0;
}
```



- *Cineva a facut definirea stucturii cu campurile si functiile necesare; altcineva implementarea.*
- *Se doreste separarea codului in: zona de definitie si cea de implementare efectiva - astfel e mai usor de implementat, modificat si citit.*

## C2: Deosebiri intre C si C++

### 7. Functii *inline*

*Inline are efect doar asupra codului generat de compilator.*

*La fiecare apel - nu se apeleaza propriu-zis functia – ci se substituie apelul cu codul ei => executie mai rapida.*

**in C++ :**

```
inline int max(int a, int b)
{
    return a>b ? a:b;
}
// utilizare:
int z = max(x, y);
/*...*/
int z1 = max(x, y);
```



*La cursul 1 am precizat ca:*

- *in C++ e descurajata utilizarea directivelor preprocesor; aceasta este alternativa.*
- *functia main nu poate fi inline*

### **in C am intalnit ceva similar:**

```
# define max(A, B) ((A) > (B) ? (A):(B))
```

*Desi similar, apare deosebirea importanta ca directiva preprocesor este substituita in codul sursa inainte de compilare, iar functia inline se substituie in textul modulului obiect – la compilare (depistarea erorilor e mai usoara)*



## C2: Deosebiri intre C si C++

### 8. Argumente de functii cu valori implicite



*In C++ se pot apela functii cu mai putine argumente decat cele din declaratie (daca au parametri cu valori implicite).*

```
int f(int k = 0) // prototip pentru functia f
{
    return k*k;
}
```

// utilizare:

```
cout<<f()<<" "<< f(5); // va afisa 0 (folosind valoarea implicita) si 25
```



```
void printvect(int *v, int n, char*text = “ ”)
{
    cout<< text<<endl;
    for (int i = 0; i < n; i++)
        cout<<v[i]<<“ ”;
}
```

// utilizare:

```
printvect(x, nx, “vectorul x”);
```

```
printvect(y, ny);
```

*De evitat situatii sensibile de tipul impartirii la 0.*





*Parametrii impliciti trebuie sa se gaseasca la finalul listei de parametri!*

```
void funct(int val=0,int x)
{
cout<<val<<" "<<x;
}
```

Eroarea generata de compilator la apelul `funct(2,4)`:  
**default argument missing for parameter 2 of `void funct(int, int)`**

**Dar?**

```
void funct(int val=0,int x=0)
{
cout<<val<<" "<<x;
}
//se poate apela: funct(2,2);
//sau funct(2);
//sau funct();
```



La apelul: `funct(2)` se afisaza: 2 0

**! DAR** Nu o sa am cum sa apelez implicit parametrul `val` si explicit parametrul `x`

## C2: Deosebiri intre C si C++

### 9. Supradefinirea functiilor (overload)

In C++ pot exista mai multe functii cu acelasi nume, dar **semnatura** diferita in acelasi program (functii supradefinite).

**Semnatura functiei** = numarul, ordinea si tipul parametrilor

**Exemplu (aria unui dreptunghi):**

```
int arie_drept(int lun, int lat)
{
    return lun*lat;
}
```

```
float arie_drept(float lun, float lat)
{
    return lun*lat;
}
```

```
float arie_drept(float lun) //arie patrat - caz special- un parametru
{
    return lun*lun;
}
```

```
float arie_drept(int lun)
{
    return lun*lun;
}
```

// utilizare:

```
cout<<arie_drept(2, 4);
cout<< arie_drept(2.5, 2.2);
cout<< arie_drept(2.5);
cout<< arie_drept(2);
```



*In C++ se pot supradefini chiar si operatori: +, -, ++, --, /, <<, etc.).*

Dati exemple de cazuri cand este util sa supradefinim functii!

# C2: Deosebiri intre C si C++

## 10. Tipul referinta

**Studiu de caz pentru construirea unei functii care interschimba doua variabile:**

```
int a,b; // variabile globale

a) void shimba_1() // functie fara parametri
{
    int temp=a;
    a = b;
    b = temp;
}

// main...
a = 3; b = 4;
cout<< a <<" "<<b; //3 4
schimba_1();
cout<< a <<" "<<b; //4 3
```



- functia interschimba doar valorile unor variabile globale – solutie limitata*
- are o viteza scazuta*

b1)

```
void schimba_2(int ca, int cb)
{
    int temp = ca;
    ca = cb;
    cb = temp;
}
```

```
// main...
```

```
int a = 3, b = 4;
```

```
cout<< a <<" "<<b;
```

```
schimba_2(a, b);
```

```
cout<< a <<" "<<b;
```

// ce rezultat – de ce?



**// valorile lui a si b nu sunt interschimbate, s-a lucrat asupra copiilor lor**

## c) cu pointeri:

```
void schimba_3(int *pa, int *pb)
{
    int temp = *pa;
    *pa = *pb;
    *pb = temp;
}
```

```
//...main()
int a,b; // variabile locale
a = 3; b = 4;
cout<< a <<" "<< b;
schimba_3(&a, &b); //transmit adresele lui a si b
cout<< a <<" "<< b; // rezultat? – de ce?
```

•codul este functional (varianta C)

•necesita atentie sporita deoarece e predisusa la erori logice - greu de detectat.

**&** - operator de referentiere (adresare)

**\*** - operatorul de dereferentiere (indirectare)



d) cu referinta (specific C++):

$\&x$  – referinta catre o variabila x (un nou tip de date)



## Transmiterea parametrilor prin referinta

```
void schimba_4(int &ra, int &rb)
{
    int temp = ra;           // pass-by-reference
    ra = rb;                 // nu se mai fac copii ale parametrilor
    rb = temp;               // se opereaza direct la adresele lor
}                             // la iesirea din functie au valorile modificate

// main...
int a=3,b=5;
schimba_4(a, b);
cout<<a<<" "<<b; //5 3
```



## Alt exemplu (functie care majoreaza cu o valoare o variabila):

```
// cu pointer
void majoreaza(int *var, int val)
{
    *var += val;
}
// main...
int x = 0;
majoreaza(&x, 5);
// adresa lui x e trimisa ca argument
```

```
// cu referinta
void majoreaza(int &var, int val)
{
    var += val;
}
// main
int x = 0;
majoreaza(x, 5);
// referinta la x e primita ca argument
```

*Nu se face o copie a lui x, ci se lucreaza direct la adresa acelei variabile*

*pass by reference*

cup = 

fillCup( )

*pass by value*

cup = 

fillCup( )

www.penjee.com



- *tipul referinta este folosit ca parametru formal in semnatura functiilor*
- *o functie nu poate returna un pointer la o variabila locala (de ce?)*
- *tot asa nu poate returna o referinta la o variabila locala;*

## Exemplu (incorect):

```
struct cmplx
{
    float re, im;
};
cmplx & add(cmplx &c1, cmplx &c2) //de ce as vrea sa transmit parametrii prin referinta?
{
    cmplx c3;
    c3.re = c1.re + c2.re;
    c3.im = c1.im + c2.im;
    return c3;
}
/* utilizare*/    cmplx a,b; //init cu val
                  cmplx z = add(a, b);
```

- *o referinta nu poate exista singura (int &ref) –se refera la ceva/cineva-> trebuie initializata!*

```
EX:  int x=3;  
      int &ref=x;  
      cout<<ref;  //3
```

- *cand operatorul & este utilizat impreuna cu o variabila, expresia inseamna “adresa variabilei indicate de referinta”;*

```
int &ref;  ERROR: `ref' declared as reference but not initialized
```

- OBS: 1.** - *un pointer (int \*pint) poate exista separat si poate fi “refolosit”*
- *el e o variabila de sine statatoare*
  - *adresa lui e diferita de adresa unei variabile catre care acesta indica.*

- 2.** - *o referinta poate fi vazuta ca un alias al unei variabile*
- *nu poate fi refolosita pentru o alta variabila;*

*//daca declar si o variabila y si fac atribuirea:*

```
int y=5; ref=y;  //pur si simplu se modifica valoarea de la adresa  
                //unde este stocata variabila x;
```

## C2: Deosebiri intre C si C++

### 11. Cuvantul rezervat “const”

**Primul sens – in contextul declararii de constante**

In C:                    #define PI 3.14159//constanta defnita

In C++:                double const PI = 3.14159;// constanta declarata

- *Cuvantul **const** este un modifcator de tip – declara o constanta de un anumit tip – specificat (int, float, double, tipuri defnite de utilizator etc.)*
- *Modificarea ulterioara a valorii acestor variabile este interzisa.*

```
const int ival = 3;                    // o constanta de tip int  
                                      // initializata cu valoarea 3
```

```
ival =4;                                // atribuire ce duce  
                                      // la un mesaj de eroare
```

# Pointer constant vs. pointer la o constanta

**Pointer constant** -nu pot schimba adresa catre care pointeaza, dar pot modifica ce se gaseste la acea adresa: **T \* const ptr;**

```
#include<stdio.h>
#include <conio.h>
#include <iostream>

int main(){
    int a[] = {10,11}, b=3;
    int* const ptr = a;
    std::cout << *ptr << std::endl; //10
    std::cout << ptr << std::endl; //adresa

    *ptr = 11; //pot modif. ce se gaseste la adresa ptr
    std::cout << *ptr << std::endl; //11
    std::cout << ptr << std::endl; //adresa

    ptr = &b;

    return 0;
}
```

[Error] assignment of read-only variable 'ptr' }

**Pointer la o constanta** – nu pot schimba valoarea de la adresa catre care pointeaza, dar pot modifica adresa catre pointeaza:

**const T\* ptr;**

```
#include<stdio.h>
#include <conio.h>
#include <iostream>

int main(){
    int a = 10, b;
    const int* ptr = &b;

    ptr = &a; //pot modifica adresa

    std::cout << *ptr << std::endl; //10
    std::cout << ptr << std::endl; //adresa

    *ptr = 11;

    return 0;
}
```

[Error] assignment of read-only location '\*ptr'

## Al doilea sens – parametrii de tip referinta const transmisi intr-o functie :

### Ce se intampla?

...

```
void sum(const int &xx, int &yy)
```

```
{
```

```
    xx+=yy;
```

```
}
```

**ERROR: assignment of read-only reference `xx'**

```
int main(int argc, char *argv[])
```

```
{
```

```
    int x=2,y=3;
```

```
    sum(x,y);
```

```
    return 0;
```

```
}
```

*Functia care primeste argumentul prin referinta constanta e informata ca nu are voie sa modifice valoarea acestuia, poate doar sa o foloseasca.*

*Asigura protectie la erori logice.*



## Al treilea sens - functii membre de tip const :

*Cuvantul cheie (rezervat) **const** apare uneori dupa lista de argumente a unor metode (functii membre).*

*Este utilizat pentru a specifica faptul ca functia respectiva **nu modifica (nu altereaza)** campurile de date ale variabilei de tip structura care o va folosi, ci, doar le citeste sau utilizeza.*

*Va fi mai clara necesitatea cand vom ajunge la clase; vor aparea si alte discutii despre utilizarea si utilitatea lui const.*

Exemplu:

```
struct dreptunghi
{
    private:                // nu este necesar
        int lun;
        int lat;
    public:
        int get_lun(void) const
        {return lun;}
        int get_lat(void) const
        {return lat;}
};
```



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

**Cineva definește structura  
după anumite specificații  
Și altcineva face  
implementarea funcțiilor**

```
struct dreptunghi
```

```
{
    private:
        int lun;
        int lat;
    public:
        void modif(int , int );
        int get_lun() const;
        int get_lat() const;
};
```

```
void dreptunghi::modif(int l1, int l2)
```

```
{
    lun=l1; lat=l2;
}
```

```
int dreptunghi::get_lun() const
```

```
{
    lun++;
    return lun;
}
```

**ERROR: increment of data-member `dreptunghi::lun' in read-only structure**

```
int dreptunghi::get_lat() const
{ if (lat<0) return 0;
  else return lat;
}
```

```
int main()
{
    dreptunghi p[3];
    p[0].modif(10,10);
    cout<<p[0].get_lun();

    return 0;
}
```

Cum aloc spațiu pentru un  
vector cu elemente de tip  
dreptunghi?

## C2: Deosebiri intre C si C++

### 12. Managementul memoriei - operatorii **new** si **delete**:

- *Utilizati pentru a aloca si elibera spatiul de memorie - **dinamic**.*
  - ***new** – aloca spatiu*
  - ***delete** – elibereaza spatiu*
- ***new** si **delete** – sunt operatori si nu necesita paranteze “()” ca in cazul functiilor **malloc()**, **calloc()**, **realloc()** si **free()***
- ***new** – returneaza un pointer de tipul cerut (**int\***, **point\***, etc) catre o zona de memorie*
- ***delete** – returneaza void*

1. Alocarea si eliberarea spatiului pentru pointeri la tipuri de date de baza

```
int *ip,*ip1;
```

```
ip = new int;    //se aloca spatiu pentru un intreg ; fara initializare
```

```
ip1=new int(3); //se aloca spatiu pentru un intreg ; cu initializare cu valoarea 3
```

```
delete ip;      ip=NULL;
```

```
delete ip1;     ip1=NULL;
```

## 2. Alocarea si eliberarea spatiului pentru tipuri de date definite de utilizator

```
dreptunghi *dp;  
dp = new dreptunghi;//se alocu spatiu pentru un  
                //element de tip dreptunghi  
dp->modific(1,1);  
delete dp; dp=NULL;
```

## 3. Alocarea si eliberarea spatiului pentru tablouri (cu elemente de orice tip)

```
int* int_tab;  
int_tab = new int[20]; // alocu spatiu pt. 20 de intregi  
cout<<int_tab[3];     //afisaza al 4-lea element  
delete [] int_tab;    // elibereaza spatiul de memorie
```

```
dreptunghi *dr_tab = new dreptunghi[20];  
delete [] dr_tab;
```



• *Prezenta parantezelor patrate “[ ]” in alocare (new) necesita prezenta sa la eliberare (delete).*

• *Lipsa in zona eliberarii a “[ ]” duce la eliberarea incompleta a memoriei (doar prima locatie – cea care corespunde elementului de pe pozitia 0<sub>35</sub> este eliberata) - > memory leak*

## C2: Deosebiri intre C si C++

### 13. Tipul nou de date bool

*O variabila de tip bool poate lua urmatoarele doua valori :*

- *true (1)*
- *false (0)*

```
bool sw;
```

```
if (3<4) sw=true; //sw=1;
```

```
    else sw=false; //sw=0;
```

```
cout<<sw;
```

```
//afisaza 1
```

## Problema 1

Care din urmatoarele functii sunt o supradefinire a functiei:

`void f(double t, int y);`

1. `void f(int t, double y);`

*Da - ordine diferita a parametrilor in semnatura*

2. `int f(double x, int y);`

*Nu - aceeași semnatura; tipuri diferite returnate nu elucideaza compilatorul*

3. `void f(double t, int y=0);`

*Nu - aceeași semnatura; se poate face apelul: f(2.0,3)- identic cu cel pt fctia exemplu*

4. `void f(dreptunghi t, dreptunghi y);`

*Da - tipuri diferite de parametrilor in semnatura*

5. `void f(double t, int j, int k=0);`

*Nu- ca la pct 3*

6. `int f(int t);`

*Da-numar diferit de parametrilor*

## Problema 2

Pot cele 3 functii sa faca parte din acelasi program? De ce?

```
void funct(dreptunghi val) //pp ca a fost implementata pt var de tip struct dreptunghi
{
cout<<val.lun*val.lat;
}
```

```
void funct(int val)
{
cout<<val;
}
```

```
void funct(int val, int x=0)
{
cout<<val<<x;
}
```

**NU**

Considerati urmatorul apel:

```
funct(3);
```

**ERROR: call of overloaded `funct(int)' is ambiguous**

## Problema 3

Pot cele 2 functii sa faca parte din acelasi program? De ce?

```
double square(double d);  
double square(double &d);
```

Se apeleaza: **square(z);**

**NU!** Argumentul z se potriveste si tipului double si tipului double &d.

DAR:

```
int f(char *s);           //pointer catre un sir de caractere  
int f(const char *s);    //pointer catre un sir de caractere constant
```

**DA!** Compilatorul face diferenta intre variabile const si ne-const.

## Problema 4

1. Creați o structură pentru numere complexe (campurile real și imaginare sunt private) și implementați o funcție membră de afișare pentru această structură

```
struct complex
{
    private: double re;
            double im;
    public: void afis() const
        {
            cout<<re<<" +j*"<<im<<endl;
        }
};
```

2. În main alocați dinamic spațiu pentru un vector de dimensiune 3 care conține nr. complexe

```
complex *vect=new complex[3];
```

3. Parcurgeți-l afișând conținutul fiecărui element și apoi eliberați spațiul ocupat.

```
for (int i=0;i<3;i++)
    vect[i].afis();
delete [] vect;
```



## Problema 5

Alocati dinamic spatiu pentru o matrice de dimensiune lin x col cu elemente de tip T.

```
T **mat=new T *[lin];
```

```
for (int i=0;i<lin;i++)
```

```
    mat[i]=new T[col];
```

Parcurgeti si afisati elementele din ultima coloana (considerati ca T e tip de date de baza)

```
for (int i=0;i<lin;i++)
```

```
    cout<<mat[i][col-1];
```

Ce se intampla daca nu era un tip de baza?

Ce trebuia sa implementam?

Eliberati spatiul ocupat de matrice (unde se gasea?)

```
for (int i=0;i<lin;i++)
```

```
    delete [] mat[i];
```

```
delete []mat;
```

Ce se intampla daca incerc dupa eliberare ceva de genul:

```
cout<<mat[1][1];
```

```
//se afisaza ce se gaseste la acea adresa
```

Cum as putea sa nu mai las loc de astfel posibile surse de erori?

```
mat=NULL;
```

## Problema 6

Ce se afiseaza?

```
int y=3;
```

```
int &x=y;
```

```
cout<<x<<" "; 3
```

```
cout<<&x<<" "; 0x28ff44
```

```
cout<<&y<<" "; 0x28ff44
```

```
x=5;
```

```
cout<<y<<" "; 5
```

```
int z=2;
```

```
x=z;
```

```
z++;
```

```
cout<<x<<y; 2 2
```

**using** - cuvântul cheie - se poate folosi ca directivă pentru introducerea în zona de lucru a unui întreg namespace (grupare de variabile/obiecte și funcții):

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

```
namespace first {
int x = 5;
int y = 10;
}
```

```
namespace second {
double x = 3.1;
double y = 2.7;
}
```

```
int main () {
using namespace first;
cout << x << endl;
cout << y << endl;
cout << second::x << endl; //folosind operatorul de rezoluție precizez că mă refer la
cout << second::y << endl; //variabilele x și y din namespaceul second
```

```
return 0;
}
```

**using si using namespace** - sunt valide doar in blocul in care au fost mentionate sau in intregul cod daca au fost folosite in maniera globala.

Exemplu de utilizare a variabilelor dintr-un namespace si apoi a variabilelor din alt namespace:

```
#include <iostream>
using namespace std;
namespace first {
int x = 5;
}
namespace second { //intr-un namespace pot sa fie implementate si functii
double x = 3.1416;
}
int main () {
{
using namespace first;
cout << x << endl;    //5
}
{
using namespace second;
cout << x << endl;    //3.1416
}
return 0;
}
```

## Namespace alias

Putem declara nume alternative pentru namespaceuri existente astfel:

```
namespace new_name = current_name;
```

## Namespace std

Toate fisierele din biblioteca standard C++ isi au declarate variabilele/obiectele necesare in namespace-ul std.

De aceea includem biblioteca <iostream> si apoi precizam ca avem nevoie de variabile/obiectele declarate in std (using namespace std), aici fiind declarate obiectele cin, cout, endl.

Putem sa le utilizam si asa: std::cin std::cout std::endl, iar namespaceul std nu mai trebuie inclus in intregime.