

PROGRAMARE ORIENTATĂ PE OBIECTE

Curs 2

Facilități ale limbajului C++ utilizate în POO

Domeniu

Namespace

Tipul referință

Argumente cu valori predefinite

Supraîncărcarea numelor de funcții

Constante, pointeri și constante

Domeniu (scope)

- ▶ O declarație introduce un nume (*identificator*) într-un domeniu astfel încât acel identificator poate fi folosit doar într-o parte specifică a unui program.
 - ▶ **Domeniu local** – un bloc este o secvență de cod delimitată de `{}`. Orice nume declarat în interiorul unui bloc este numit **nume local**;
 - ▶ **Domeniu clasă, structură** – un nume dacă este declarat în interiorul unei clase/structuri este numit nume membru. Domeniul de valabilitate începe de la deschiderea cu `{` a declarației clasei și până la încheierea declarației clasei;
 - ▶ **Domeniu *namespace*** – un nume este numit nume membru al *namespace* dacă este definit într-un namespace în afara oricărei funcții, clase etc membre a spațiului;
 - ▶ **Domeniu global** – un nume este global dacă este definit în afara oricărei funcții, clase, namespace etc. Domeniul de valabilitate din momentul declarației și până la sfârșitul fișierului;



Domeniu (scope)

- ▶ **Domeniu instrucțiune** (statement scope) – un nume este în domeniu instrucțiune dacă este definit în interiorul **()** unei instrucțiuni **for**, **while**, **if** sau **switch**. Domeniul de valabilitate este limitat doar la instrucțiune. Este considerat nume local;
- ▶ **Domeniu funcție** – un nume declarat este valabil tot corpul funcției.
- ▶ O declarație a unui nume într-un bloc poate ascunde o declarație din blocul părinte sau una globală. Astfel un nume poate fi redefinit (*poate referi un alt tip*) în interiorul oricărui bloc existent. După ieșirea din blocul respectiv se poate reveni la definiția anterioară.



Domeniu (scope)

```
int x; // variabilă globală x
```

```
void f(void)
{
```

```
    int x; // variabilă locală x ascunde variabila globală x
```

```
    x = 1; // atribuire variabilei locale x
```

```
    {
```

```
        int x; // ascunde prima declarație locală a variabile x
```

```
        x = 2; // atribuire variabilei locale secundare x
```

```
    }
```

```
    x = 3; // atribuire primei variabile locale x
```

```
}
```

```
int *p = &x; //inițializare cu adresa variabilei globale x
```

- ▶ Ascunderea (*umbrirea*) numelor este inevitabilă în programele mari



Domeniu (scope)

- ▶ Un nume global ascuns poate fi accesat cu ajutorul operatorului de rezoluție de domeniu `::`.

```
int x;  
void f2(void)  
{  
    int x = 1; // ascunde variabila globală x  
    ::x = 2;   // atribuire variabile globale x  
    x = 2;     // atribuire variabilei locale x  
}
```

- ▶ O variabilă locală ascunsă nu poate fi accesată.



Domeniu (scope)

- ▶ Domeniul unui nume care nu este membru al unei clase începe din momentul declarației complete înainte de inițializare:

```
int x = 97;  
void f3()  
{  
    int x = x; // fără sens: inițializarea variabilei x cu  
               //valoarea sa neinițializată  
}
```



Domeniu (scope)

- ▶ Este posibilă utilizarea unui singur nume pentru a referi două obiecte diferite:

```
int x = 11;
void f4(void) // fără sens: utilizarea a două nume într-un
              // singur domeniu, ambele numite x
{
    int y = x; // utilizarea variabilei globale x: y = 11
    int x = 22;
    y = x;     // utilizarea variabile locale x: y = 22
}
```



Namespace (continuare)

- ▶ Orice program constă din mai multe părți separate.
- ▶ Ex.
 - ▶ Programul *"Hello, world!"* implică cel puțin două părți: cererea utilizatorului de a tipări *"Hello, world!"* și sistemul I/O care va realiza tipărirea
- ▶ *Namespace* permite gruparea unor entități cum ar fi clase, obiecte și funcții sub același nume. Astfel, domeniul global poate fi divizat în subdomenii, fiecare cu numele său.
- ▶ Formatul este:

```
namespace identificador
{
    entitati
}
```
- ▶ Unde *identificador* este un nume de identificare valid iar *entitatile* sunt clase, obiecte, variabile și funcții care sunt incluse în acel namespace



Namespace (continuare)

- ▶ Exemplu: *namespace myFirstNamespace*
 {

int a = 15;

int b = 17;

 }

//accesarea variabilelor

myFirstNamespace::a;

myFirstNamespace::b;

- ▶ În exemplul de mai sus *a* și *b* sunt variabile declarate în namespace-ul *myFirstNamespace*. Pentru a accesa variabilele respective din afara *myFirstNamespace* se va folosi operatorul de domeniu.



Namespace (continuare)

- Funcționalitatea namespace-urilor este utilă atunci când există posibilitatea ca un obiect global sau funcție să aibă aceeași denumire cauzând erori de redefinire.

```
namespace mySecondNamespace
{
    int a = 25;
    int b = 27;
}
```

```
int main (void)
{
    cout << myFirstNamespace::a << endl;
    cout << mySecondNamespace::a << endl;

    return 0;
}
```



Namespace (continuare)

- ▶ Cuvântul cheie **using** este folosit pentru a introduce un nume/entitate dintr-un namespace în regiunea curentă de declarații

```
int main (void)
{
    using myFirstNamespace::a;
    using mySecondNamespace::b;

    cout << a << endl;
    cout << b << endl;
    cout << myFirstNamespace::b << endl;
    cout << mySecondNamespace::a << endl;

    return 0;
}
```



Namespace (continuare)

- ▶ Cuvântul cheie **using** poate fi folosit pentru a introduce un întreg namespace

```
int main (void)
{
    using namespace myFirstNamespace;

    cout << a << endl;
    cout << b << endl;
    cout << mySecondNamespace::a << endl;
    cout << mySecondNamespace::b << endl;

    return 0;
}
```



Namespace (continuare)

- ▶ Se pot utiliza mai multe namespace-uri dacă se ține cont de domeniu

```
int main (void)
{
    {
        using namespace first;
        cout << x << endl;
    }
    {
        using namespace second;
        cout << x << endl;
    }
    return 0;
}
```

- ▶ Se pot declara nume alternative pentru namespace-uri existente astfel:

```
namespace new_name = current_name;
```



Tipul referință în C++

- ▶ Referința este un alias pentru o anumite variabilă. Dacă în C, transmiterea parametrilor unei funcții se face prin valoare (inclusiv și pentru pointeri), în C++ se adaugă și transmiterea parametrilor prin referință.
- ▶ Dacă tipul pointer se introduce prin construcția: *tip **, tipul referință se introduce prin *tip &*.
- ▶ O variabilă referință trebuie să fie inițializată la definirea sau declararea ei cu numele unei alte variabile.

```
void f(void)
{
    int i = 1;
    int &r = i;    //r si i refera același i
    int x = r;    //x=1
    r = 2;        //i=2
}
```

Tipul referință în C++

- ▶ Fiecare referință trebuie inițializată

```
int i = 1;
int &r1 = i;    //ok
int &r2;        //error
extern int &r3;  //ok, r3 initializat in alta parte
```

- ▶ Inițializarea unei referințe este ceva diferit față de asignare (atribuire). În ciuda aparențelor nici un operator nu acționează asupra referințelor

```
void g(void)
{
    int ii = 0;
    int &rr = ii;
    rr++;          //ii este incrementat cu 1
    int *pp = &rr; //pp pointează către ii
}
```

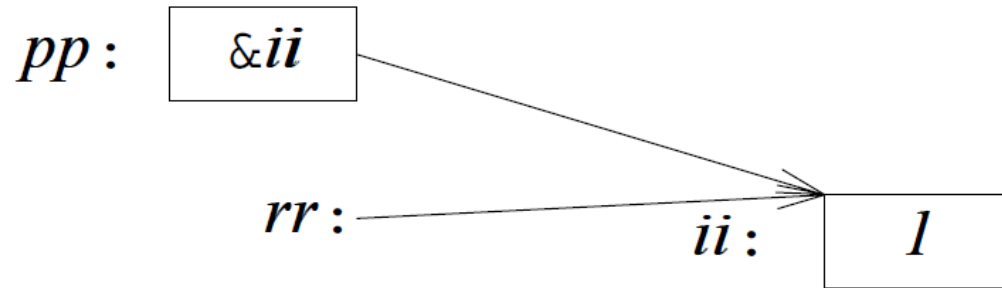


Tipul referință în C++

- ▶ `rr++` nu incrementează referința `rr`. Operatorul `++` este aplicat unui `int` care este `ii`.
- ▶ În consecință referința nu poate fi schimbată după inițializare. Întotdeauna va referi obiectul cu care a fost inițializată.
- ▶ Pentru a obține un pointer referit de `rr` se poate scrie `&rr`
- ▶ Implementarea evidentă a unei referințe este un pointer constant care este dereferențiat de fiecare dată când este folosit dacă se are în vedere că nu poate fi manipulat precum un pointer.



Tipul referință în C++



- ▶ O referință poate fi folosită ca argument într-o funcție astfel încât funcția să schimbe valoarea obiectului respectiv.

```
void increment(int &aa){ aa++; }  
void f(void)  
{  
    int x = 1;  
    increment(x); //x=2  
}
```

- ▶ La apelul funcției `increment` argumentul `aa` devine un alt nume al variabilei `x`.

Tipul referință în C++

- ▶ Totuși pentru a menține lizibilitatea unui program se recomandă a se evita utilizarea funcțiilor care își modifică argumentele. În schimb se pot folosi pointeri.

```
int next(int p) { return p + 1;}  
void incr(int*p) { (*p)++;}  
void g(void)  
{  
    int x = 1;  
    increment(x); //x=2  
    x = next(x);  //x=3  
    incr(&x);     //x=4  
}
```

- ▶ **increment(x)** nu furnizează nici un indiciu că variabila x este modificată



Argumente cu valori predefinite

- ▶ O îmbunătățire adusă limbajului C consta în posibilitatea specificării valorilor argumentelor unei funcții atunci când se declară prototipul ei. Aceste argumente sunt considerate a fi ***predefinite***.
- ▶ Fie redefinirea funcției de copiere a două șiruri:

```
char *MyStrcpy( char *dest, char *src, int sense = 0 );
```

- ▶ Următoarele apeluri sunt corecte:

```
MyStrcpy( sirDest, sirSrc );  
MyStrcpy( sirDest, sirSrc, 0 );  
MyStrcpy( sirDest, sirSrc, 1 );
```



Argumente cu valori predefinite

```
char *MyCopy( char *dest, char *src, int sense )
{
    size_t i; /*!< Index pentru parcurgere a sirului sursa*/
    size_t srcLength = strlen(src);

    for( i = 0; *(src + i); i++ )
    {
        /*(dest + i) = *(src + i); /*!< Copiere normala*/
        /*(dest + i) = *(src + srcLength - 1 - i); /*!< Copiere
                                                    inversa*/

        *(dest + i) = *(src + i + sense*(srcLength - 1 - 2 * i));
    }

    return dest;
}
```



Argumente cu valori predefinite

► Apel:

```
char src[]="Ana are mere";  
char dest[20]={0};
```

```
    ///copiere normala (stanga-dreapta)
```

```
MyCopy(dest, src);  
    cout << "Sirul sursa este: " << src << "\n";  
    cout << "Sirul destinatie este: " << dest << "\n";
```

```
    ///copiere inversa (dreapta-stanga)
```

```
MyCopy(dest, src,1);  
    cout << "Sirul sursa este: " << src << "\n";  
    cout << "Sirul destinatie este: " << dest << "\n";
```



Argumente cu valori predefinite

- ▶ Compilatorul recunoaște argumentele prin poziția lor.
- ▶ La apelul funcției pot fi lăsate nespecificate doar ultimele argumente, pentru care se iau valorile predefinite.
- ▶ Declararea unui argument ca având valoare implicită se face o singura dată.
- ▶ Concluzie:
 - ▶ La proiectarea unei funcții cu argumente predefinite, este bine să ordonăm argumentele în cadrul listei de argumente astfel: mai întâi lista parametrilor ficși (cei care se modifică cel mai des la apelul funcției), urmată de lista parametrilor cu valori implicite (cei care se modifică cel mai rar la apelul funcției)



Supraîncărcarea funcțiilor

- ▶ În C++ pot exista mai multe funcții cu același nume, dar cu liste diferite de argumente. Aceste funcții sunt ***supraîncărcate***.
- ▶ Supraîncărcarea se utilizează pentru a da același nume tuturor funcțiilor care fac același tip de operație.
- ▶ Numele unei funcții și ansamblul argumentelor sale, ca număr și tipuri, se numește ***semnătura*** acelei funcții. Se observă că funcțiile supraîncărcate au semnături diferite.



Supraîncărcarea funcțiilor cu un singur argument

► Ex.1:

```
short int MyAbs(short int x)
{
    return (x<0)? -x:x;
}
```

```
Long MyAbs(Long x)
{
    return (x<0)? -x:x;
}
```

```
double MyAbs(double x)
{
    return (x<0)? -x:x;
}
```

```
//Functii pentru valoare  
absoluta in C_ANSI  
int abs(int);  
Long Labs(Long);  
double fabs(double);
```

```
//Apel  
int a = -3;  
Long c = -5;  
double f = -23.2;
```

```
MyAbs(a) ;  
MyAbs(c) ;  
MyAbs(f) ;
```



Supraîncărcarea funcțiilor cu două argumente

► Ex. 2: Ridicarea la putere

//Declaratii

```
int MyPow( int n, int p );  
float MyPow( float n, int p );  
double MyPow( double n, int p );
```

//Definitii

```
int MyPow( int n, int p ){  
    return (p < 0)? 0:((p == 0)? 1: n * MyPow(n, p-1));  
}
```

```
float MyPow( float n, int p ){  
    return (p < 0)? 0:((p == 0)? 1: n * MyPow(n, p-1));  
}
```

```
double MyPow( double n, int p ){  
    return (p < 0)? 0:((p == 0)? 1: n * MyPow(n, p-1));  
}
```

//Apel

```
int x = 3, p = 3;  
float y = 2.0;  
double z = 3.3;
```

```
MyPow( x, p );  
MyPow( y, p );  
MyPow( z, p );
```

Supraîncărcarea funcțiilor


- ▶ Declaraarea funcțiilor supraincarcate respecta regulile de domeniu cunoscute. Astfel, daca se declara un prototip intr-un bloc, domeniul lui va fi acel bloc, ascunzand celelalte variante de prototipuri

```
//Declaratii  
int Fct(char);  
int Fct(char*);  
void Functie(void);
```



Supraîncărcarea funcțiilor

```
int Fct(char a) {
    cout << "int Fct(char) returneaza " << a << endl;
    return a;
}
int Fct(char *s){
    printf( "int Fct(char*s) returneaza adresa sir %s: -> %X\n", s, s);
    return 1;
}
void Fct(Long nr){
    cout << "void Fct(Long nr) afiseaza nr.Long=" << nr << endl;
}
void Functie(void) {
    void Fct(Long);
    cout << "Apel al functiei void Functie(void)" << endl;
    printf("\t"); Fct('a');
    printf("\t"); Fct(678);
}
//Apel
Fct('A');
Fct("Ana are mere");
Functie();
```



Supraîncărcarea funcțiilor cu argumente implicite

//Declaratii

```
void Function( void );  
void Function( int, int = 0 );  
void Function( char, char = 'a', int = 0 );
```

```
void Function( void )  
{  
    cout << "Apel void Function(void)" << endl;  
}
```

```
void Function( int a, int b )  
{  
    cout << "Apel void Function( int, int = 0 )" << endl;  
}
```

```
void Function( char a, char b, int c )  
{  
    cout << "Apel void Function( char, char = 'a', int = 0 )" << endl;  
}
```

//Apel

```
Function();  
Function(2);  
Function(2,-3);  
Function('a','b',5);  
Function('B');
```

Supraîncărcarea funcțiilor

- ▶ Compilatorul C++ selectează funcția corectă prin compararea tipurilor argumentelor din apel cu cele din declarație.
- ▶ Când facem supraîncărcarea funcțiilor, trebuie să avem grijă ca numărul și/sau tipul argumentelor versiunilor supraîncărcate să fie diferite. Nu se pot face supraîncărcări dacă listele de argumente sunt identice:

```
int Compute( int x );  
double Compute( int x );
```

- ▶ O astfel de supraîncărcare (numai cu valoarea returnată diferită) este ambiguă. Compilatorul nu are posibilitatea să discearnă care variantă este corectă și semnalează eroare.



Constante

- ▶ C++ oferă conceptul de constantă definită de utilizator, **const**, pentru a exprima noțiunea că o valoare nu se schimbă direct.
- ▶ Contextele în care sunt utile:
 - ▶ Variabile care nu își modifică valoarea după inițializare
 - ▶ Constantele simbolice permit o întreținere mai ușoară a programului
 - ▶ Majoritatea pointerilor sunt folosiți pentru a se citi o valoare(adresă) și nu pentru a fi scriși
 - ▶ Majoritatea parametrilor unei funcții sunt citiți nu scriși
- ▶ Cuvântul **const** poate fi adăugat unei declarații pentru a face obiectul respectiv constant și trebuie inițializat.

```
const int model = 90;           //model este const
const int v[] = { 1, 2, 3, 4 }; //v[i] este const
const int x;                   //error: nu există inițializare
```



Constante

- ▶ Declaraarea unui obiect **const** va asigura ca valoarea acestuia nu se modifică în cadrul blocului.

```
void f( void )  
{  
    model = 200;    //eroare  
    v[2]++;         //eroare  
}
```

- ▶ **Obs.** Cuvântul cheie **const** modifică un tip în sensul că restricționează modurile în care un obiect poate fi folosit.

```
void g(const X* p)  
{  
    //(*p) nu poate fi modificat  
}  
void h(void)  
{  
    X val; //val poate fi modificat  
    g(&val);  
}
```

Constante

- ▶ Funcție de compilator, se poate alocă spațiu pentru variabila respectivă sau nu.

```
const int c1 = 1;  
const int c2 = 2;  
const int c3 = my_f(3); //valoarea lui c3 nu este cunoscuta la compilare  
extern const int c4; //valoarea lui c4 nu este cunoscuta la compilare  
const int *p = &c2; //este necesar a se alocă spațiu pentru c2
```

- ▶ Pentru vectori se alocă memorie deoarece compilatorul nu poate ști ce element al vectorului este folosit într-o expresie.
- ▶ Constantele sunt utilizate în mod frecvent ca limite pentru vectori sau *case labels*:



Constante

```
const int a = 42;
const int b = 99;
const int max = 128;
int v[max];
void f(int i)
{
    switch(i)
    {
        case a:
            //...
        case b:
            //...
    }
}
```

- Enumeratori sunt o alternativă pentru astfel de cazuri



Pointeri și constante

- ▶ Prin folosirea unui pointer sunt implicate două obiecte: pointerul însuși și obiectul pointat
- ▶ Precedarea declarației unui pointer cu un *const* face obiectul (nu pointerul) constant.
- ▶ Declararea unui pointer constant presupune utilizarea **const* și nu doar ***.

```
void f1(char* p){  
    char s[] = "Gica";  
    const char* pc = s;           //pointer catre constant  
    pc[3] = 'g';                  //error: pc pointeaza catre constant  
    pc = p;                       //ok  
    char *const cp = s;           //pointer constant  
    cp[3] = 'a';                  //ok  
    cp = p;                       //error: cp este constant  
    const char *const cpc = s;    //pointer constant catre constant  
    cpc[3] = 'a';                 //error: cpc pointeaza catre constant  
    cpc = p;                      //error: cpc este constant  
}
```

Pointeri și constante

- ▶ Operatorul care face un pointer constant este **const*. Nu există declaratorul *const**. Dacă este întâlnit este considerat că face parte din tipul de bază.

```
char *const cp;    //pointer constant catre char  
char const* pc;    //pointer catre char constant  
const char* pc2;  //pointer catre char constant
```

- ▶ Modificatorul *const* se folosește frecvent la funcții, la declararea parametrilor formali de tip pointer sau referințe, pentru a interzice funcțiilor respective modificarea datelor spre care pointează parametrii respectivi.

```
char* strcpy(char* p, const char* q); //nu poate modifica (*q)
```

- ▶ Protecția datelor cu ajutorul *const* nu este totală pentru toate compilatoarele

