

PROGRAMARE ORIENTATĂ OBIECT

Curs 4

Programarea orientată pe obiecte în C++

- Constructor de copiere
- Constructor explicit
- Funcții membre inline
- Membri statici
- This

- Operatori new și delete
- Funcții membre constante
- Prieteni

Constructorul de copiere

- ▶ Implicit obiectele pot fi copiate. În particular, un obiect poate fi inițializat cu copia unui alt obiect din aceeași clasă

Date d = today;

- ▶ Copierea implicită a obiectelor este o copiere membru cu membru.
- ▶ Constructorul de copiere este un constructor special al clasei
- ▶ Este utilizat pentru a copia un obiect existent de tipul clasei
- ▶ Are un singur argument, o referință către obiectul ce va fi copiat
- ▶ Forma generală este:

Date(const Date& obj);
Date(Date& obj);


- ▶ În cazul în care o clasă conține variabile de tip pointer,
- ▶ trebuie făcută alocare de memorie

Constructorul de copiere

```
class Data
{
    unsigned char zi, luna;
    unsigned int an;
public:
    void Afisare(void){ cout <<"Data:
                                "<<an<<"."<<luna<<"."<<zi<<endl;};
    Data(unsigned char z, unsigned char l, unsigned int a);
    Data(void);
    Data(Data &);
    ~Data();
};

Data::Data(Data & d)
{
    zi=d.zi;
    luna=d.luna;
    an=d.an;
}

int main(void)
{
    Data d3(d2);
    d3.afisare();
    return 0;
}
```



Constructorul de copiere

- ▶ Cazuri în care se apelează constructorul de copiere:

- ▶ La inițierea unui obiect nou creat cu un alt obiect

Data d2 = d1;

- ▶ La transferul parametrilor unei funcții prin valoare
- ▶ Funcție ce returnează un obiect

- ▶ În clasă se adaugă:

public:

Data Increment ();

- ▶ Definiția funcției:

```
Data Data::Increment (void)
{
    Data temp;
    temp.zi = zi + 1;
    temp.luna = luna + 1;
    temp.an = an;
    cout << "înainte de return"<<endl;
    return temp;
}
```

Constructorul de copiere

```
Data::Data(void)
{
    cout << "S-a apelat constr. fara argumente" << endl;
}

Data::Data(unsigned char z, unsigned char l, int a)
{
    cout << "S-a apelat constr. cu lista de argumente" << endl;
    zi = z;
    luna = l;
    an = a;
}

Data::Data(Data &d)
{
    cout << "S-a apelat constr. de copiere" << endl;
    zi=d.zi;
    luna=d.luna;
    an=d.an;
}
```



Constructorul de copiere

```
int main(void)
{
    cout<<"d1: ";
    Data d1 (9,1,2012);

    cout<<"d2: ";
    Data d2;
    d2 = d1.Increment();

    cout << "dupa apelul functiei increment" << endl;

    d2=d1;
    return 0;
}
```

d1: S-a apelat constr. cu lista de argumente

d2: S-a apelat constr. fara argumente

S-a apelat constr. fara argumente

inainte de return

S-a apelat constr. de copiere

▶ *dupa apelul functiei increment*

Constructorul de copiere

```
class Persoana
{
    char * nume;
    int varsta;
public:
    Persoana(void);
    Persoana (char *sName, int v);

    ~Persoana (void);
    void Afiseaza(void);
    Persoana SchimbaVarsta(int);
};
```



Constructorul de copiere

```
#include <iostream>
#include "header.h"
```

```
using namespace std;
```

```
Persoana::Persoana(void)
{
    nume = new char [1];
    varsta = 0;
}
```

```
Persoana::Persoana(char * n, int v)
{
    nume = new char [strlen (n) + 1];
    strcpy_s (nume, strlen (n) + 1, n);
    varsta = v;
}
```



Constructorul de copiere

```
Persoana::~~Persoana (void)
{
    if (nume)
        delete [] nume;
}
```

```
void Persoana::Afiseaza(void)
{
    cout << "Nume=" << nume << " varsta=" << varsta<< endl;
}
```

```
Persoana Persoana::SchimbaVarsta(int n)
{
    Persoana temp (nume, varsta);
    varsta = n;
    return temp;
}
```



Constructorul de copiere

```
#include <iostream>
#include "header.h"
using namespace std;

int main (void)
{
    Persoana p1 ("PaveL", 22);
    p1.Afiseaza();

    Persoana p2 ("Ana", 23);

    Persoana p3 = p2.SchimbaVarsta(25);

    p2.Afiseaza();

    p3.Afiseaza();
    return 0;
}
```

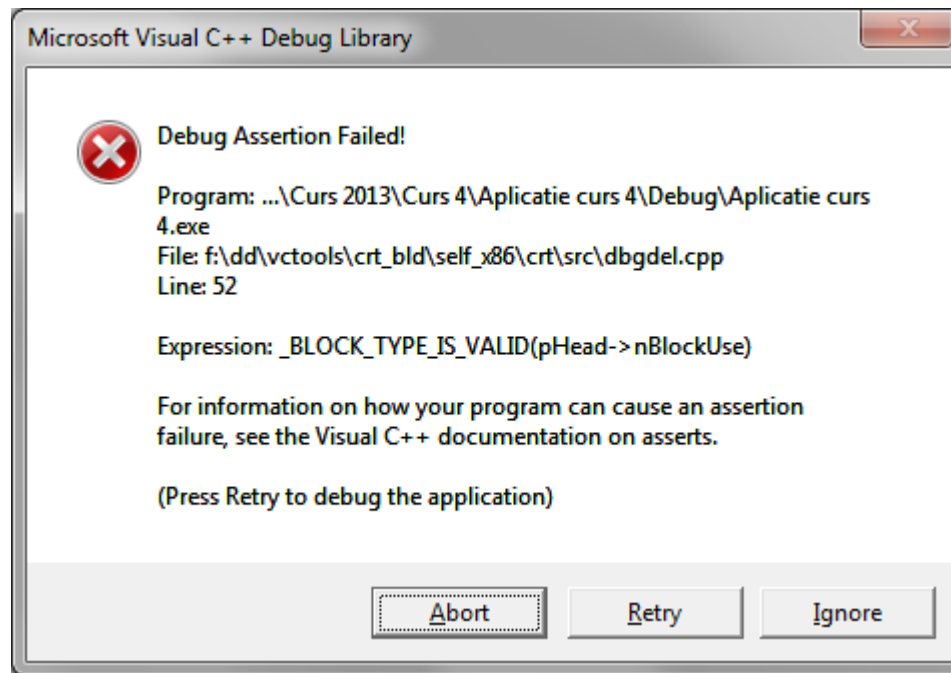


Constructorul de copiere

Nume=Pavel varsta=22

Nume=Ana varsta=25

Nume=TTTTTTTTTt2T varsta=23



Constructorul de copiere

- ▶ De ce ? Pentru ca lipsește constructorul de copiere!

- ▶ Se adauga în clasa un constructor de copiere:

Persoana (Persoana & p);

- ▶ Cu definiția

```
Persoana::Persoana (Persoana & p)  
{  
    nume = new char [strlen (p.nume) + 1];  
    strcpy_s (nume, strlen (p.nume) + 1, p.nume);  
    varsta = p.varsta;  
}
```



Constructor explicit

- ▶ Este constructorul ce nu poate lua parte la nici o conversie implicită
- ▶ Se declară folosind cuvântul cheie **explicit**
- ▶ Numai constructorii cu un singur argument pot lua parte într-o conversie implicită

```
class A
{
public:
    explicit A(int);
};

A::A(int)
{}

A a1 = 37;
```

error C2440: 'initializing' : cannot convert from 'int' to 'A'



Funcții inline

- Pentru funcții membru, dacă corpul funcției este definit în cadrul clasei, atunci acele funcții sunt implicit inline, cu condiția să respecte regulile de scriere a unor funcții inline

```
class Persoana
{
    char * nume;
    int varsta;
public:
    //...
    char*GetName(void)
    {
        return nume;
    }
    int GetVarsta(void);
    //...
};
```

```
inline int Persoana::GetVarsta(void){ return varsta;}
```

Memברי statici

- ▶ O variabilă care este membră a unei clase dar nu membră a unui obiect al clasei respective se numește membru *static*.
- ▶ Similar, o funcție care necesită accesul la membri unei clase, fără să fie apelată pentru un obiect oarecare, se numește funcție membră *statică*.



Membri statici

```
class Data
{
    unsigned char zi, luna;
    unsigned int an;
    static Data dataImplicita;
public:
    Data(unsigned char zz = 0, unsigned char ll = 0,
          unsigned int aa = 0);
    static void SetImplicitData(unsigned char,
                                unsigned char, unsigned int);
};

Data::Data(unsigned char zz, unsigned char ll, unsigned int aa)
{
    zi = zz ? zz : dataImplicita.zi;
    luna = ll ? ll : dataImplicita.luna;
    an = aa ? aa : dataImplicita.an;
}
```

Clasa – membri statici

- ▶ Membri statici, atât funcțiile cât și variabilele, trebuie să fie definiți (inițializați) cumva:

```
Data Data::dataImplicita(16, 12, 1770);
```

//sau

```
void Data::SetImplicitData(unsigned char zz, unsigned char ll,  
                           unsigned int aa)
```

```
{  
    Data::dataImplicita = Data(zz, ll, aa);  
}
```

```
int main(void)  
{  
    Data::SetImplicitData(16, 12, 1770);  
}
```



Pointerul *this*

- ▶ Pointerul *this* este o variabilă predefinită în C++ accesibilă în corpul oricărei metode *non-static* din cadrul unei clase
- ▶ Valoarea pointerului este dată de adresa obiectului pentru care s-a apelat o anumită metodă non-statică din clasă
- ▶ Este folosit:
 - ▶ Pentru a înlătura ambiguitățile dintre un parametru al unei funcții și o variabilă membră
 - ▶ În cazurile când este necesar un pointer către obiectul pentru care s-a apelat o anumită metodă
- ▶ Compilatorul C++ convertește apelul funcției non-statice apelate și pune ca prim parametru pointerul *this*.



Operatorii new, delete, new[] și delete[]

- ▶ Operatorii *new*, *delete*, *new[]* și *delete[]* sunt implementați folosind funcțiile:

```
void* operator new(size_t);  
void operator delete(void*);  
void *operator new[](size_t);  
void operator delete[](void*);
```

- ▶ La folosirea operatorului *new* pentru alocarea unui singur obiect sau variabile, este apelat automat constructorul implicit sau, dacă există constructori declarați, cel ce corespunde listei de argumente
- ▶ La folosirea operatorului *delete*, este apelat automat destructorul clasei



Operatorii new, delete

```
► int main()
{
    cout<<"d1: ";
    Data d1;

    cout<<"d2: ";
    Data * d2 = new Data;

    cout<<"d3: ";
    Data * d3 = new Data (1,2,2003);
    cout<<"d4: ";
    Data * d4 = new Data (*d3);
    cout<<"d5: ";
    Data * d5 = (Data*)malloc(sizeof(Data));
    cout<<endl;
    cout<<"final:";
    delete d2;
    delete d3;
    delete d4;
    free(d5);
    //delete d5;
    return 0;
}
►
```

Operatorii new, delete

d1: S-a apelat constr. fara argumente

d2: S-a apelat constr. fara argumente

d3: S-a apelat constr. cu lista de argumente

d4: S-a apelat constr. de copiere

d5:

final:s-a apelat destructorul

s-a apelat destructorul

s-a apelat destructorul



Alocarea si dealocarea tablourilor

- ▶ Pentru alocare de memorie unui tablou se folosește operatorul ***new[]***
 - ▶ Se apeleaza constructorul corespunzător pentru fiecare element (cel implicit sau cel fără argumente dacă este declarat)
- ▶ Pentru dealocare se foloseste operatorul ***delete []***
 - ▶ Se apelează destructorul pentru fiecare element

```
Date *dv;  
dv = new Data [10];  
  
delete [] dv;
```



Funcții membre constante

► Fie clasa: `class Data {`
 `unsigned char zi, luna;`
 `unsigned int an;`
 `public:`
 `unsigned char Zi(void) const { return zi; }`
 `unsigned char Luna(void) const { return luna; }`
 `unsigned int An(void) const;`
 `unsigned int AddAn(unsigned int aa) {an+=aa;};`
};

- Prin prezența cuvântului cheie **`const`** se specifică compilatorului că funcțiile respective nu pot modifica conținutul obiectului *Data*.

```
inline unsigned int Date::An(void) const
{
    return an++; //eroare: incercare de modificare a unei variabile
                //membre intr-o functie constanta
}
```



Funcții membre constante

- ▶ O funcție membră constantă definită în afara clasei necesită sufixul **const**.

```
inline unsigned int Date::An(void)const //corect
{
    return y;
}
inline unsigned int Date::year(void) //eroare
{
    return y;
}
```

- ▶ O funcție membră constantă poate fi apelată atât de obiecte **const** cât și de obiecte *non-const*.

```
void f(Data &d, const Data &cd){
    unsigned int i = d.An(); //ok
    d.AddAn(1);              //ok
    unsigned int j = cd.An(); //ok
    cd.AddAn(1);             //eroare: obiect constant
```

▶ }

Funcții membre constante

- ▶ Pentru un obiect declarat cu ***const*** dacă se apelează o metodă care nu este ***const***, compilatorul dă o eroare de compilare.



Prieteni

- ▶ Ascunderea membrilor unei clase este anulată pentru prietenii acelei clase, care pot fi funcții sau alte clase.
- ▶ O funcție prietenă are acces la toți membrii clasei, inclusiv la cei privați ori protejați.
- ▶ Regula este valabilă pentru toți membri unei clase prietene
- ▶ Pentru fiecare prieten al unei clase, există o declarație în interiorul clasei, *friend*.
 - ▶ *Funcții prietene care aparțin domeniului global*
 - ▶ *Funcții prietene care sunt membre ale unei clase*
 - ▶ *Clase prietene*



Funcții prietene globale

```
class Time
{
    int min, sec, hour;
public:
    Time(void) : min(0), sec(0), hour(0){};
    friend int min(Time& t);
    friend int sec(Time& t);
    friend int hour(Time& t);
}

inline int min(Time& t) { return t.min; }
inline int sec(Time& t) { return t.sec; }
inline int hour(Time& t) { return t.hour; }

Time t;
cout << "min: " << min(t) << ", sec: " << sec(t) << ", hour: " << hour(t) << endl;
```

- ▶ Funcțiile prietene sunt utile pentru supraîncărcarea operatorilor
-

Funcții membru prietene

- ▶ O funcție membru a unei clase poate fi funcție prietenă pentru altă clasă

```
class XXX {  
public:  
    void f(void);  
};  
  
class YYY {  
    int nr;  
public:  
    YYY(int n): nr(n){}  
    friend void XXX::f(void);  
};  
  
void XXX::f(void) {  
    YYY ob(10);  
    cout << ob.nr << endl;  
}
```



Clase prietene

- Declarând o clasă prietenă în cadrul unei alte clase, se extinde calitatea de prieten la toate funcțiile membru

```
class YYY  
{  
    friend class XXX;  
}
```



Vă mulțumesc !

