



Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2021 – 2022
Semestrul II
Seriile 13, 14, 15

Curs 3



Cuprinsul cursului

- Class, struct, union
- Functii si clase prieten
- Functii inline
- Constructori / destructor



Struct si class

- singura diferenta: struct are default membri ca public iar class ca private
- struct defineste o clasa (tip de date)
- putem avea in struct si functii
- pentru compatibilitate cu cod vechi
- extensibilitate
- **a nu se folosi struct pentru clase**



// Utilizarea unei structuri pentru a defini o clasa.

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

```
struct mystr {
    void buildstr(char *s)
        { if(!*s) *str = '\0';
          else strcat(str, s); }
    void showstr() { cout << str << "\n"; }
private:
    char str[255];
};
```

```
class mystr {
    char str[255];
public:
    void buildstr(char *s); //
public
    void showstr();
};
```



Union si class

- la fel ca struct
- toate elementele de tip data folosesc aceeași locație de memorie
- membrii sunt publici (by default)



Union si class

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

union swap_byte {
    void swap() { unsigned char t = c[0]; c[0] = c[1]; c[1] = t; }
    void set_byte(unsigned short i) { u = i; }
    void show_word() { cout << u; }

    unsigned short u;
    unsigned char c[2];
};

int main() {
    swap_byte b;
    b.set_byte(49034);
    b.swap();
    b.show_word();
    return 0;
}
```



Union ca o clasa

- union nu poate mosteni
- nu se poate mosteni din union
- nu poate avea functii virtuale (nu avem mostenire)
- nu avem variabile de instanta statice
- nu avem referinte in union
- nu avem obiecte care fac overload pe =
- obiecte cu (con/de)structor definiti nu pot fi membri in union



Union anonime

- nu au nume pentru tip
- nu se pot declara obiecte de tipul respectiv
- folosite pentru a spune compilatorului cum se aloc/procesez variabilele respective in memorie
 - folosesc aceeasi locatie de memorie
- variabilele din union sunt accesibile ca si cum ar fi declarate in blocul respectiv



Union anonime

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

```
int main()
{
```

```
    union {
        long l;
        double d;
        char s[4];
    };
```

```
    l = 100000;
    cout << l << " ";
    d = 123.2342;
    cout << d << " ";
    strcpy(s, "hi");
    cout << s;
    return 0;
```

```
}
```



Union anonime

- nu poate avea functii
- nu poate avea private sau protected (fara functii nu avem acces la altceva)
- union-uri anonime globale trebuiesc precizate ca statice



Funcții prieten

- Cuvantul cheie: **friend**
- pentru accesarea campurilor protected, private din alta clasa
- folosite la overload-area operatorilor, pentru unele functii de I/O, si portiuni interconectate (exemplu urmeaza)
- in rest nu se prea folosesc



Funcții prieten pentru o clasă

```
#include <iostream>
using namespace std;
class myclass {
    int a, b;
public:
    friend int sum(myclass x); // poate accesa direct a si b private
    void set_ab(int i, int j) { a = i; b = j; }
};

int sum(myclass x) { return x.a + x.b; }

int main() {
    myclass n;
    n.set_ab(3, 4);
    cout << sum(n);
    return 0;
}
```



Funcții prieten pentru mai multe clase

```
#include <iostream>
using namespace std;
class C2;
class C1 {    int x;
public:
    void set_x(int a){x = a;}
    friend void f(C1, C2);
};

class C2 {    int y;
public:
    void set_y(int b){y = b;}
    friend void f(C1, C2);
};
```

```
void f(C1 ob1, C2 ob2)
{ cout<<ob1.x + ob2.y<<"\n";}
```

```
int main()
{
    C1 A;
    C2 B;
    A.set_x(10);
    B.set_y(20);
    f(A,B);
}
```



Funcții prieten din alte obiecte

```
#include <iostream>
using namespace std;
class C2;
class C1 {    int x;
public:
    void set_x(int a){x = a;}
    void f(C2);
};

class C2 {    int y;
public:
    void set_y(int b){y = b;}
    friend void C1::f(C2);
};
```

```
void C1::f(C2 ob2)
{ cout<<this->x + ob2.y<<"\n";}
```

```
int main()
{
    C1 A;
    C2 B;
    A.set_x(10);
    B.set_y(20);
    A.f(B);
}
```



Clase prieten

- Declararea unei clase Y ca prieten al unei clase X, are ca efect ca toate funcțiile membre ale clasei Y au acces la membrii privati ai clasei X.

```
#include <iostream>
```

```
class C1 {    int x;
```

```
public:
```

```
    friend class C2;
```

```
};
```

```
class C2 {
```

```
public:
```

```
    void set_x(int a, C1& ob){ ob.x = a;}
```

```
    int get_x (C1 ob) {return ob.x;}
```

```
};
```

```
int main()
```

```
{
```

```
    C1 A;
```

```
    C2 B;
```

```
    B.set_x(10,A);
```

```
    std::cout<<B.get_x(A);
```

```
}
```



Funcții inline

- execuție rapidă
- este o sugestie/cerere pentru compilator
- pentru funcții foarte mici
- pot fi și membri ai unei clase
- foarte comune în clase
- două tipuri: explicit (**inline**) și implicit



Explicit inline

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

```
inline int max(int a, int b)
{
    return a>b ? a : b;
}
```

```
int main()
{
    cout << max(10, 20);
    cout << " " << max(99, 88);
    return 0;
}
```

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

```
int main()
{
    cout << (10>20 ? 10 : 20);
    cout << " " << (99>88 ? 99 : 88);
    return 0;
}
```



Explicit inline in clase

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

```
class myclass {
    int a, b;
public:
    void init(int i, int j);
    void show();
};
```

```
inline void myclass::init(int i, int j)
{ a = i; b = j; }
```

```
inline void myclass::show()
{ cout << a << " " << b << "\n"; }
```

```
int main() {
    myclass x;
    x.init(10, 20);
    x.show();
    return 0;
}
```



Definirea functiilor inline implicit (in clase)

```
#include <iostream>
```

```
using namespace std;
```

```
class myclass {
```

```
    int a, b;
```

```
public:
```

```
// automatic inline
```

```
    void init(int i, int j)
```

```
{
```

```
        a = i;
```

```
        b = j;
```

```
}
```

```
    void show() { cout << a << " " << b << "\n"; }
```

```
};
```

```
int main() {
```

```
    myclass x;
```

```
    x.init(10, 20);
```

```
    x.show();
```

```
    return 0;
```

```
}
```



Constructori/Destructor

- inițializare automată
- efectueaza operatii prealabile utilizarii obiectelor create
- obiectele nu sunt statice
- constructor: funcție specială, numele clasei
- constructorii nu pot întoarce valori (nu au tip de întoarcere)



Constructori/Destructorii

Caracteristici speciale:

- numele = numele clasei (~ numele clasei pentru destructori);
- la declarare nu se specifica tipul returnat;
- nu pot fi mosteniti, dar pot fi apelati de clasele derivate;
- nu se pot utiliza pointeri către functiile constructor / destructor;
- constructorii pot avea parametri (inclusiv impliciti) și se pot supradefini. Destructorul este unic și fără parametri.



Constructori/Destructor

Constructorii de copiere (discuții mai ample mai tarziu)

- creaza un obiect preluand valorile corespunzatoare altui obiect;
- exista implicit (copiata bit-cu-bit, deci trebuie redefinit la date alocate dinamic).



Constructori/Destructor

Orice clasa, are by default:

- un constructor de initializare
- un constructor de copiere
- un destructor
- un operator de atribuire.

Constructorii parametrizati:

- argumente la constructori
 - putem defini mai multe variante cu mai multe numere si tipuri de parametri
- overload de constructori (mai multe variante, cu numar mai mare și tipuri de parametri).



Constructori/Destructor

Orice clasa, are by default:

```
class A
{
    int x;
    float y;
    string z;
};

int main()
{
    A a;      // apel constructor de initializare - fara parametri
    A b = a;  // apel constructor de copiere
    A e(a);   // apel constructor de copiere
    A c;      // apel constructor de initializare
    c = a;    //operatorul de atribuire (=)
}
```




Constructori/Destructor

Exemplu – necesitate rescriere constructori

```
class A
{
    int *v;
public:
    A(){v = new int[10]; cout<<"C";}
    ~A(){delete[]v; cout<<"D";}
    void afis(){cout<<v[3];}
};

void functie (A ob) { ob.afis(); }

int main()
{
    A o1;
    afisare(o1);
    o1.afis();
}
```



Constructori/Destructorii

Exemplu - Constructori parametrizati

```
class A {  
    int x;  
    float y;  
    string z;  
public:  
    A() {x = -45;}  
    A(int x) {this->x = x; this->y = 5.67; z = "Seria 14";}  
    // this -> camp ⇔ camp simplu: this -> z echivalent cu z  
    A(int x, float y) {this->x = x; this->y = y; z = "Seria 13 25";}  
    A(int x, float y, string z) {this->x = x; this->y = y; this->z = z;}  
  
int main() {  
    A a;  
    A b(34);  
    return 0;  
}
```



Constructori/Destructor

Exemplu - Constructori parametrizati

```
class A
{
    int x;
    float y;
    string z;
public:
    A(int x = -45, float y = 5.67, string z = "Seria 13") // valori implicite
    { this->x = x; this->y = y; z = z; }
};

int main()
{
    A a;
    A b(34);
    return 0;
}
```



Constructori/Destructor

Functiile constructor cu un parametru – caz special (sursa H. Schildt)

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

class X {
    int a;
public:
    X(int j) { a = j; }
    int geta() { return a; }
};

int main()
{
    X ob = 99; // passes 99 to j
    cout << ob.geta(); // outputs 99
    return 0;
}
```

**Se creeaza o conversie
explicita de date!**



Constructorii/Destructorii

Tablouri de obiecte

Daca o clasa are constructori parametrizati, putem initializa diferit fiecare obiect din vector.

```
class X{int a,b,c; ... };  
X v[3] = {X(10,20) , X (1,2,3), X(0) };
```

Daca constr are un singur parametru, atunci se poate specifica direct valoarea.

```
class X {  
    int i;  
public:  
    X(int j) { i = j;}  
};
```

```
X v[3] = {10,15,20 };
```



Constructori/Destructor

Exemplu – Ordine apel

```
class A
{
    int x;
public:
    A(int x = 0)  {
        x = x;
        cout<<"Constructor"<<x<<endl;  }
    ~A()  {
        cout<<"Destructor"<<endl;  }
    A(const A&o)  {
        x = o.x;
        cout<<"Constructor de copiere"<<endl;  }
    void f_cu_referinta(A&ob3)  {
        A ob4(456);  }
    void f_fara_referinta(A ob6)  {
        A ob7(123);  }
} ob;
```

```
int main()
{
    A ob1(20), ob2(55);
    ob2.f_cu_referinta(ob1);
    ob1.f_fara_referinta(ob2);
    A ob5;
    return 0;
}
```



Constructori/Destructor

Exemplu – Ordine apel

- 1) In acelasi domeniu de vizibilitate, constructorii se apeleaza in ordinea declararii obiectelor, iar destructorii in sens invers.
- 2) Variabilele globale se declara inaintea celor locale, deci constructorii lor se declara primii.
- 3) Daca o functie are ca parametru un obiect, care nu e transmis cu referinta atunci, se activeaza constructorul de copiere si, implicit, la iesirea din functie, obiectul copie se distruge, deci se apeleaza destructor

```
/// Ordine:  
/// 1. Constructor ob;  
/// 2. Constructor ob1;  
/// 3. Constructor ob2;  
/// 4. Constructor ob4; - ob3 e referinta/alias-ul ob1, nu se creeaza obiect nou  
/// 5. Destructor ob4;  
/// 6. Constructor copiere ob6  
/// 7. Constructor ob7  
/// 8. Destructor ob7  
/// 9. Destructor ob6  
/// 10. Constructor ob5  
/// 11. Destructor ob5  
/// 12. Destructor ob2  
/// 13. Destructor ob1  
/// 14. Destructor ob
```



Constructori/Destructor

Exemplu – Ordine apel

```
class A {  
public:  
  A() {cout<<"Constr A"<<endl;}  
};
```

```
class B {  
public:  
  B() {cout<<"Constr B"<<endl;}  
private:  
  A ob;  
};
```

```
int main()  
{  
  B ob2;  
  /// Apel constructor obiect A si apoi constructorul propriu  
}
```




Constructori/Destructor

```
class A {  
    int x;  
public:  
    A(int x = 7){this->x = x; cout<<"Const "<<x<<endl;}  
    void set_x(int x){this->x = x;}  
    int get_x(){ return x;}  
    ~A(){cout<<"Dest " <<x<<endl;}  
};  
  
void afisare(A ob) {  
    ob.set_x(10);  
    cout<<ob.get_x()<<endl; }  
  
int main ( ) {  
    A o1;  
    cout<<o1.get_x()<<endl;  
    afisare(o1);  
    return 0;  
}
```

Exemplu – Ce se afiseaza?

Const 7 // obiect o1
7 // o1.get_x()
10 // in functie ob.get_x()
Dest 10 // ob
Dest 7 // o1



Constructori/Destructori

```
class cls { public:  
    cls() { cout << "Inside constructor 1" << endl; }  
    ~cls() { cout << "Inside destructor 1" << endl; } };
```

```
class cls {  
    cls xx;  
public:  
    cls() { cout << "Inside constructor 2" << endl; }  
    ~cls() { cout << "Inside destructor 2" << endl; } };
```

Exemplu – Ce se afiseaza?

```
class cls2 {  
    cls xx;  
    cls xxx;  
public:  
    cls2() { cout << "Inside constructor 3" << endl; }  
    ~cls2() { cout << "Inside destructor 3" << endl; } };
```

```
int main() {  
    cls2 s;  
}
```



Polimorfism pe constructori

- foarte comun sa fie supraincarcati
- de ce?
 - flexibilitate
 - pentru a putea defini obiecte initializate si neinitializate
 - constructori de copiere: copy constructors



overflow pe constructori: flexibilitate

- putem avea mai multe posibilitati pentru initializarea/construirea unui obiect
- definim constructori pentru toate modurile de initializare
- daca se incearca initializarea intr-un alt fel (decat cele definite): eroare la compilare

Facultatea de Matematică și Informatică

Universitatea din București



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

class date { int day, month, year;
public:
    date(char *d);
    date(int m, int d, int y);
    void show_date();
};
```

```
// Initialize using string.
date::date(char *d)
{ sscanf(d, "%d%*c%d%*c%d", &month, &day, &year); }
```

```
// Initialize using integers.
date::date(int m, int d, int y)
{ day = d; month = m; year = y; }
```

```
void date::show_date()
{
    cout << month << "/" << day;
    cout << "/" << year << "\n";
}
```

```
int main()
{
    date ob1(12, 4, 2003), ob2("10/22/2003");
    ob1.show_date(); Enter new date: ";
    ob2.show_date();
    return 0;
}
```

"%d%*c%d%*c%d"

citim din sir

*: ignoram ce citim

c: un singur caracter

citim 3 intregi sau
luna/zi/an



polimorfism de constructori: obiecte initializate si ne-initializate

- important pentru array-uri dinamice de obiecte
- nu se pot initializa obiectele dintr-o lista alocata dinamic
- asadar avem nevoie de posibilitatea de a crea obiecte neinitializate (din lista dinamica) si obiecte initializate (definite normal)



```
#include <iostream>
#include <new>
using namespace std;
class powers
{ int x;
public:
    // overload constructor two ways
```

- ofThree si lista p au nevoie de constructorul fara parametri

```
};

int main()
{
    powers ofTwo[] = {1, 2, 4, 8, 16}; // initialized
    powers ofThree[5]; // uninitialized
    powers *p;
    int i; // show powers of two
    cout << "Powers of two: ";
    for(i=0; i<5; i++) {
        cout << ofTwo[i].getx() << " ";
    }
    cout << "\n\n";
```

```
// set powers of three
ofThree[0].setx(1); ofThree[1].setx(3);
ofThree[2].setx(9); ofThree[3].setx(27);
ofThree[4].setx(81);

// show powers of three
cout << "Powers of three: ";
for(i=0; i<5; i++) { cout << ofThree[i].getx() << " ";}
cout << "\n\n";
```

```
return 1;}
```

```
// initialize dynamic array with powers of two
for(i=0; i<5; i++) { p[i].setx(ofTwo[i].getx());}

// show powers of two
cout << "Powers of two: ";
for(i=0; i<5; i++) { cout << p[i].getx() << " ";}
cout << "\n\n";
delete [] p;
return 0;
}
```



polimorfism de constructori: constructorul de copiere

- pot apărea probleme când un obiect initializează un alt obiect

`MyClass B = A;`

- aici se copiază toate câmpurile (starea) obiectului A în obiectul B
- problema apare la alocare dinamică de memorie: A și B folosesc aceeași zonă de memorie pentru că pointerii arată în același loc
- destructorul lui MyClass eliberează aceeași zonă de memorie de două ori (distruge A și B)



constructorul de copiere

- aceeași problema
 - apel de funcție cu obiect ca parametru
 - apel de funcție cu obiect ca variabilă de întoarcere
 - în aceste cazuri un obiect temporar este creat, se copiază prin constructorul de copiere în obiectul temporar, și apoi se continuă
 - deci vor fi din nou două distrugeri de obiecte din clasa respectivă (una pentru parametru, una pentru obiectul temporar)



constructorul de copiere

Cazuri de utilizare:

Initializare explicita:

```
MyClass B = A;  
MyClass B (A);
```

Apel de functie cu obiect ca parametru:

```
void f(MyClass X) {...}
```

Apel de functie cu obiect ca variabila de intoarcere:

```
MyClass f() {MyClass obiect; ... return obiect;}.  
MyClass x = f();
```

Copierea se poate face și prin operatorul = (*detalii mai târziu*).



putem redefini constructorul de copiere

```
classname (const classname &o) {  
    // body of constructor  
}
```

- *o* este obiectul din dreapta
- putem avea mai multi parametri (dar trebuie sa definim valori implicite pentru ei)
- **&** este apel prin referinta
- putem avea si atribuire (*o1=o2;*)
 - redefinim operatorii mai tarziu, putem redefini =
 - = diferit de initializare

Facultatea de Matematică și Informatică

Universitatea din București



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

class array {
    int *p;
    int size;
public:
    array(int sz) {
        try {
            p = new int[sz];
        } catch (bad_alloc xa) {
            cout << "Allocation Failure\n";
            exit(EXIT_FAILURE);
        }
        size = sz;
    }
    ~array() { delete [] p; }

    // copy constructor
    array(const array &a);
    void put(int i, int j) { if(i >= 0 && i < size) p[i] = j; }
    int get(int i) { return p[i]; }
};
```

```
// Copy Constructor
array::array(const array &a) {
    int i;
    try {
        p = new int[a.size];
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        exit(EXIT_FAILURE);
    }
    for(i=0; i<a.size; i++) p[i] = a.p[i];
}
```

```
int main()
{
    array num(10);
    int i;
    for(i=0; i<10; i++) num.put(i, i);
    for(i=9; i>=0; i--) cout << num.get(i);
    cout << "\n";

    // create another array and initialize with num
    array x(num); // invokes copy constructor
    for(i=0; i<10; i++) cout << x.get(i);
    return 0;
}
```



- Observatie: constructorul de copiere este folosit doar la initializari
- daca avem
 - array a(10);
 - array b(10);
 - b=a;
 - nu este initializare, este copiere de stare
 - este posibil sa trebuiasca redefinit si operatorul = (mai tarziu)



Perspective

Curs 4

- Static, clase locale
- Operatorul ::
- supraincercarea functiilor in C++
- supraincercarea operatorilor in C++