



Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2019 – 2020
Semestrul II
Seriile 13, 14 și 21

Curs 3

3-4/3/2020



Cuprinsul cursului

- Recapitularea discuțiilor din cursul anterior (Generalități despre curs, Reguli de comportament)
- Struct, Union si clase
- functii prieten
- Constructori/destructori



Organizatorice

- **Examen: 9 iunie 2020**
- **Laboratoare**
- **Seminar**



Ascunderea informației

foarte importantă

public, protected, private

Avem acces?	public	protected	private
Aceeași clasă	da	da	da
Clase derivate	da	da	nu
Alte clase	da	nu	nu



Alocare dinamica

C

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *a,*b;
    a = (int *)malloc(sizeof(int));
    b = (int *)malloc(4 * sizeof(int));
    b = (int *) realloc(b,7 * sizeof(int));
    free(a);
    free(b);
    return 0;
}
```

C++

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

int main()
{
    int *a,*b;
    a = (int *)malloc(sizeof(int));
    b = (int *)malloc(4 * sizeof(int));
    b = (int *)realloc(b,7 * sizeof(int));
    free(a);
    free(b);

    a = new int(); // valoare oarecare
    cout<<*a<<endl;
    delete a;
    a = new int(22);
    cout<<*a<<endl;
    delete a;
    b = new int[4];
    delete[] b;
    return 0;
}
```



Transmiterea parametrilor

C

```
void f(int x){ x = x *2;}

void g(int *x){ *x = *x + 30;}

int main()
{
    int x = 10;
    f(x);
    printf("x = %d\n",x);
    g(&x);
    printf("x = %d\n",x);
    return 0;
}
```

C++

```
#include <iostream>
using namespace std;
void f(int x){ x = x *2;} //prin valoare
void g(int *x){ *x = *x + 30;} // prin pointer
void h(int &x){ x = x + 50;} //prin referinta

int main()
{
    int x = 10;
    f(x);
    cout<<"x = "<<x<<endl;
    g(&x);
    cout<<"x = "<<x<<endl;
    h(x); cout<<"x = "<<x<<endl;
    return 0;
}
```



Transmiterea parametrilor

Observatii generale

- parametrii formali - sunt creati la intrarea intr-o functie si distrusi la retur;
- apel prin valoare - copiaza valoarea unui argument intr-un parametru formal \Rightarrow modificarile parametrului nu au efect asupra argumentului;
- apel prin referinta - in parametru este copiata adresa unui argument \Rightarrow modificarile parametrului au efect asupra argumentului.
- functiile, cu exceptia celor de tip void, pot fi folosite ca operand in orice expresie valida.



Transmiterea parametrilor

Regula generala: in C o functie nu poate fi tinta unei atribuirii.

In C++ - se accepta unele exceptii, permitand functiilor respective sa se gaseasca in membrul stang al unei atribuirii.

```
char s[10] = "Hello";  
char& f(int i) { return s[i];}
```

```
int main()  
{  
    f(2) = 'X';  
    cout<<s;  
    return 0;  
}
```




Transmiterea parametrilor

Cand tipul returnat de o functie nu este declarat explicit, i se atribuie automat int.

Tipul trebuie cunoscut inainte de apel.

```
f (double x)
{
    return x;
}
```

Prototipul unei functii: permite declararea in afara si a numarului de parametri / tipul lor:

```
void f(int); // antet / prototip
```

```
int main() { cout<< f(50); }
```

```
void f( int x)
{
    // corp functie;
}
```



Functii in structuri

C

```
#include <stdio.h>
#include <stdlib.h>
struct test
{
    int x;
    void afis()
    {
        printf("x= %d",x);
    }
}A;

int main()
{
    scanf("%d",&A.x);
    A.afis(); /* error ? struct test' has no
member called afis() */
    return 0;
}
```

C++

```
#include <iostream>
using namespace std;
struct test
{
    int x;
    void afis()
    {
        cout<<"x= "<<x;
    }
}A;

int main()
{
    cin>>A.x;
    A.afis();
    return 0;
}
```



Moștenire

- terminologie
 - clasă de bază, clasă derivată
 - superclasă subclasă
 - părinte, fiu
- mai târziu: funcții virtuale, identificare de tipuri în timpul rulării (RTTI)



Constructori/Destructor

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



// Aceste linii creează clasa stack.

```
class stack {  
    int stck[SIZE];  
    int tos;  
  
    public:  
        stack(); // constructor  
        void push(int i);  
        int pop();  
};
```

// constructorul clasei stack

```
stack::stack()  
{  
    tos = 0;  
    cout << "Stack Initialized\n";  
}
```



- constructorii/destructorii sunt chemați de fiecare dată o variabilă/obiect de acel tip este creată/distrusă. Declarații active nu pasive.
- Destructori: reversul, execută operații când obiectul nu mai este folositor
- memory leak

stack::~~stack()



// Creăm clasa stack.

```
class stack {  
    int stck[SIZE];  
    int tos;  
  
    public:  
        stack(); // constructor  
        ~stack(); // destructor  
        void push(int i);  
        int pop();  
};
```

// constructorul clasei stack

```
stack::stack()  
{  
    tos = 0;  
    cout << "Stack Initialized\n";  
}
```

// destructorul clasei stack

```
stack::~~stack()  
{  
    cout << "Stack Destroyed\n";  
}
```

// Using a constructor and destructor.

```
#include <iostream>
```

```
using namespace std;
```

```
#define SIZE 100
```

// This creates the class stack.

```
class stack {
```

```
    int stck[SIZE];
```

```
    int tos;
```

```
public:
```

```
    stack(); // constructor
```

```
    ~stack(); // destructor
```

```
    void push(int i);
```

```
    int pop();
```

```
};
```

// stack's constructor

```
stack::stack()
```

```
{
```

```
    tos = 0;
    cout << "Stack Initialized\n";
```

```
}
```

// stack's destructor

```
stack::~~stack()
```

```
{
```

```
    cout << "Stack Destroyed\n";
```

```
}
```

Stack Initialized
Stack Initialized

3 1 4 2

Stack Destroyed
Stack Destroyed

```
void stack::push(int i){
```

```
    if(tos==SIZE) {
```

```
        cout << "Stack is full.\n",
```

```
        return;
```

```
    }
```

```
    stck[tos] = i;
```

```
    tos++;
```

```
}
```

```
int stack::pop(){
```

```
    if(tos==0) {
```

```
        cout << "Stack underflow.\n";
```

```
        return 0;
```

```
    }
```

```
    tos--;
```

```
    return stck[tos];
```

```
}
```

```
int main(){
```

```
    stack a, b; // create two stack objects
```

```
    a.push(1);
```

```
    b.push(2);
```

```
    a.push(3);
```

```
    b.push(4);
```

```
    cout << a.pop() << " ";
```

```
    cout << a.pop() << " ";
```

```
    cout << b.pop() << " ";
```

```
    cout << b.pop() << "\n";
```

```
    return 0;
```

```
}
```





Clasele în C++

- cu “class”
- obiectele instanțiază clase
- similare cu struct-uri și union-uri
- au funcții
- specificatorii de acces: public, private, protected
- default: private
- protected: pentru moștenire, vorbim mai târziu



```
class nume_clasă {  
    private variabile și funcții membru  
    specificator_de_acces:  
        variabile și funcții membru  
    specificator_de_acces:  
        variabile și funcții membru  
    // ...  
    specificator_de_acces:  
        variabile și funcții membru  
} listă_obiecte;
```

- putem trece de la public la private și iar la public, etc.



```
class employee {  
    char name[80]; // private din oficiu  
public:  
    void putname(char *n); // acestea sunt publice  
    void getname(char *n);  
private:  
    double wage; // acum din nou private  
public:  
    void putwage(double w); // înapoi la public  
    double getwage();  
};
```

```
class employee {  
    char name[80];  
    double wage;  
public:  
    void putname(char *n);  
    void getname(char *n);  
    void putwage(double w);  
    double getwage();  
};
```

- se folosește mai mult a doua variantă
- un membru (ne-static) al clasei nu poate avea inițializare
- nu putem avea ca membri obiecte de tipul clasei (putem avea pointeri la tipul clasei)
- nu auto, extern, register



- variabilele de instanta (instance variables)
- membri de tip date ai clasei
 - in general private
 - pentru viteza se pot folosi “public” dar NU LA
ACEST CURS



Exemplu

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

class myclass {
public:
    int i, j, k; // accessible to entire program
};

int main()
{
    myclass a, b;
    a.i = 100; // access to i, j, and k is OK
    a.j = 4;
    a.k = a.i * a.j;
    b.k = 12; // remember, a.k and b.k are different
    cout << a.k << " " << b.k;
    return 0;
}
```



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



// Using a structure to define a class.

```
#include <iostream>
```

```
#include <cstring>
```

```
using namespace std;
```

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

```
void mystr::buildstr(char *s){  
    if(!*s) *str = '\0'; // initialize string  
    else strcat(str, s);}  
void mystr::showstr() {cout << str << "\n"; }  
int main() {  
    mystr s;  
    s.buildstr(""); // init  
    s.buildstr("Hello ");  
    s.buildstr("there!");  
    s.showstr();  
    return 0; }
```

```
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)



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

```
union swap_byte {
    void swap();
    void set_byte(unsigned short i);
    void show_word();
    unsigned short u;
    unsigned char c[2];
};
```

```
void swap_byte::swap()
{
```

```
    unsigned char t;
    t = c[0];
    c[0] = c[1];
    c[1] = t;
```

```
}
```

```
void swap_byte::show_word() { cout << u;}
```

```
void swap_byte::set_byte(unsigned short i)
{
    u = i;
}
```

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

35519



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



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

int main()
{
    // define anonymous union
    union {
        long l;
        double d;
        char s[4];
    };
    // now, reference union elements directly
    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



functii prieten

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



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

```
class myclass {
    int a, b;

public:
    friend int sum(myclass x);
    void set_ab(int i, int j);
};

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

// Note: sum() is not a member function of any class.

```
int sum(myclass x) {
    /* Because sum() is a friend of myclass, it can directly access a and b. */
    return x.a + x.b;
}

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



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

```
const int IDLE = 0;
const int INUSE = 1;
class C2; // forward declaration
```

```
class C1 {
    int status; // IDLE if off, INUSE if on screen // ...
public:
    void set_status(int state);
    friend int idle(C1 a, C2 b);
};
```

```
class C2 {
    int status; // IDLE if off, INUSE if on screen // ...
public:
    void set_status(int state);
    friend int idle(C1 a, C2 b);
};
```

```
void C1::set_status(int state)
{ status = state; }
```

```
void C2::set_status(int state)
{ status = state; }
```

```
int idle(C1 a, C2 b)
{
    if(a.status || b.status) return 0;
    else return 1;
}
```

```
int main()
{
    C1 x;
    C2 y;
    x.set_status(IDLE);
    y.set_status(IDLE);
    if(idle(x, y)) cout << "Screen can be used.\n";
    else cout << "In use.\n";
    x.set_status(INUSE);
    if(idle(x, y)) cout << "Screen can be used.\n";
    else cout << "In use.\n";
    return 0;
}
```




- functii prieten din alte obiecte

Facultatea de Matematică și Informatică

Universitatea din București



```
#include <iostream>
using namespace std;
const int IDLE = 0;
const int INUSE = 1;

class C2; // forward declaration

class C1 {
    int status; // IDLE if off, INUSE if on screen // ..}
public:
    void set_status(int state);
    int idle(C2 b); // now a member of C1
};

class C2 {
    int status; // IDLE if off, INUSE if on screen // ...
public:
    void set_status(int state);
    friend int C1::idle(C2 b);
};

void C1::set_status(int state)
{
    status = state;
}
```

```
void C2::set_status(int state)
{
    status = state;
}

// idle() is member of C1, but friend of C2
int C1::idle(C2 b)
{
    if(THIS->status || b.status) return 0;
    else return 1;
}

int main()
{
    C1 x;
    C2 y;
    x.set_status(IDLE);
    y.set_status(IDLE);
    if(x.idle(y)) cout << "Screen can be used.\n";
    else cout << "In use.\n";
    x.set_status(INUSE);
    if(x.idle(y)) cout << "Screen can be used.\n";
    else cout << "In use.\n";
    return 0;
}
```



clase prieten

- daca avem o clasa prieten, toate functiile membre ale clasei prieten au acces la membrii privati ai clasei

Facultatea de Matematică și Informatică

Universitatea din București



// Using a friend class.

```
#include <iostream>
```

```
using namespace std;
```

```
class TwoValues {  
    int a;  
    int b;  
public:  
    TwoValues(int i, int j) { a = i; b = j; }  
    friend class Min;  
};
```

```
class Min {  
public:  
    int min(TwoValues x);  
};
```

```
int Min::min(TwoValues x)  
{ return x.a < x.b ? x.a : x.b; }
```

```
int main() {  
    TwoValues ob(10, 20);  
    Min m;  
    cout << m.min(ob);  
    return 0; }
```



functii inline

- foarte comune in clase
- doua tipuri: explicit (**inline**) si implicit



Explicit

```
#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;
}
```



functii inline

- executie rapida
- este o sugestie/cerere pentru compiler
- pentru functii foarte mici
- pot fi si membri ai unei clase



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

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

// Create an inline function.
inline void myclass::init(int i, int j)
{ a = i; b = j; }

// Create another inline function.
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;
}
```

```
#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";
    }
};
```



Constructori parametrizati

- trimitem argumente la constructori
- putem defini mai multe variante cu mai multe numere si tipuri de parametrii
- overload de constructori



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

```
class myclass {
    int a, b;
public:
    myclass(int i, int j) {a=i; b=j;}
    void show() {cout << a << " " << b;}
};
```

```
int main()
{
    myclass ob(3, 5);
    ob.show();
    return 0;
}
```

```
myclass ob = myclass(3, 4);
```

- a doua forma implica “copy constructors”
- discutat mai tarziu



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

```
const int IN = 1;
const int CHECKED_OUT = 0;
```

```
class book {
    char author[40];
    char title[40];
    int status;

public:
    book(char *n, char *t, int s);
    int get_status() {return status;}
    void set_status(int s) {status = s;}
    void show();
};
```

```
book::book(char *n, char *t, int s)
{
    strcpy(author, n);
    strcpy(title, t);
    status = s;
}
```

```
void book::show()
{
    cout << title << " by " << author;
    cout << " is ";
    if(status==IN) cout << "in.\n";
    else cout << "out.\n";
}
```

```
int main()
{
    book b1("Twain", "Tom Sawyer", IN);
    book b2("Melville", "Moby Dick", CHECKED_OUT);
    b1.show();
    b2.show();
    return 0;
}
```



constructori cu un paramentru

- se creeaza o conversie implicita de date

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

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;
}
```



Tablouri de obiecte

- Dacă o clasă are constructori parametrizați, putem initializa diferit fiecare obiect din vector.

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

class X {
    int a,b,c;
public:
    X(int i) { a = i; b = 0; c = 0; }
    X(int i, int j) { a = i; b = j; c = 0; }
    X(int i, int j, int k) { a = i; b = j; c = k; }
};

int main()
{
    X v[3] = {X(10,20) , X (1,2,3), X(0) };
    return 0;
}
```



Tablouri de obiecte

- Caz particular

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

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

int main()
{
    X v[3] = {10,20,30};
    return 0;
}
```