



UNIVERSITATEA BABEŞ-BOLYAI  
Facultatea de Matematică și Informatică



# Programare orientată obiect

---

Curs 01

Laura Dioşan

# Conținut

---

- Introducere
  - Programare structurată la nivel inferior
  - Programare structurată la nivel superior
  
- Elemente de bază ale limbajului C++
  - Generalități
  - Procesul de compilare
  - Elemente de limbaj

# Introducere

---

- Programare structurată
  - Una dintre marile paradigme de programare
  - Programare structurată la nivel inferior
  - Programare structurată la nivel superior

# Programare structurată la nivel inferior

---

- ❑ Fără instrucțiuni GOTO și BREAK
- ❑ Utilizarea corectă a instrucțiunii FOR
- ❑ Nume sugestive pentru variabile
- ❑ Cod indentat
- ❑ Utilizarea comentariilor
- ❑ Utilizarea unor programe simple, ierarhice, bazate pe structuri:
  - secvențiale
  - selective (*if...then...else...endif*, *switch* or *case*)
  - repetitive (*while*, *repeat*, *for* or *do...until*)

# Programare structurată la nivel superior

---

- Programare procedurală
  - subalgoritmi: funcții și proceduri
  - evitarea utilizării variabilelor globale
  - utilizarea corectă a parametrilor
  
- Programare top-down
  - descompunerea problemei în subprobleme → arbore programului
  
- Programare modulară (TAD-uri)
  - modul = unitate de cod care poate fi compilat
  - module independente
  - biblioteci
  - avantaje:
    - Lucrul în echipă
    - Testare și întreținere ușoară
    - Complexitate ascunsă
  
- Programare orientată obiect

# Exemplu: TAD

---

## □ Domeniu

- Mulțimea tuturor valorilor posibile ale TAD-ului

## □ Operații

- Inițializare (creare și alocare de memorie)
- Conversie (din/în alte tipuri de date)
- Selecție (set & get)
- Verificare de proprietăți
- Operații specifice TAD-ului

# Programare orientată obiect (POO)

---

## □ De ce POO?

- POO permite programatorilor să se concentreze
  - pe tipul de date, în primul rând
  - și apoi pe metode

## □ Caracteristici

- Abstractizarea datelor
  - încapsularea datelor și operațiilor
  - ascunderea datelor (reprezentării lor)
- Moștenirea
  - reutilizarea codului
- Polimorfismul
  - un obiect își poate schimba comportamentul în funcție de starea sa

# POO – elemente de bază

---

- Clasa
  - un tip de date
  - e.g. Implementarea unui TAD
- Obiect
  - instanța unei clase
  - O variabilă de tip clasă
- Metodă
  - Un mod de comunicare între obiecte

*Def: POO este o metodă de proiectare și dezvoltare a programelor (aplicațiilor) cu următoarele caracteristici:*

- *obiectele reprezintă elementele de bază cu care se lucrează*
- *fiecare obiect corespunde unui tip de date*
- *clasele interacționează prin relații de moștenire și compoziție*



# POO – avantaje și limbaje

---

## □ Avantaje

- complexitate redusă
- reutilizarea codului
- întreținere ușoară
- modificări ușor de realizat
- cod lizibil

## □ Limbaje

- SIMULA67 – primul limbaj OO; noțiunea de *clasă*
- SMALLTALK – limbaj pur OO
- JAVA – foarte apropiat de un limbaj OO pur
- C++ – limbaj OO hibrid; Bjarne Stroustrup (1983-1985)

# Elemente de bază ale limbajului C++

---

- Generalități
- Procesul de compilare
- Elemente de limbaj

# Generalități

## □ Medii

- Borland C
- Microsoft Visual C++
- C++ Builder
- Etc.

## □ Structura unui program

- Directive de procesare
- Declarații de date/variabile globale
- Declarații/definiții de funcții
- Funcția principală (main)

## ■ Unul sau mai multe fișiere cu extensiile: .cpp, .h

- fișiere header
  - declarații, interfețe
  - o legătură între biblioteci și utilizator
- fișiere sursă
  - definiții, implementări

## ■ Directive de pre-procesare

- **#ifndef** – pentru evitarea unor declarații multiple
- în fișierele header: eg. *MyHeader.h*
- **#pragma once** → fișierul sursă curent este inclus o singură dată într-o singură compilare

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

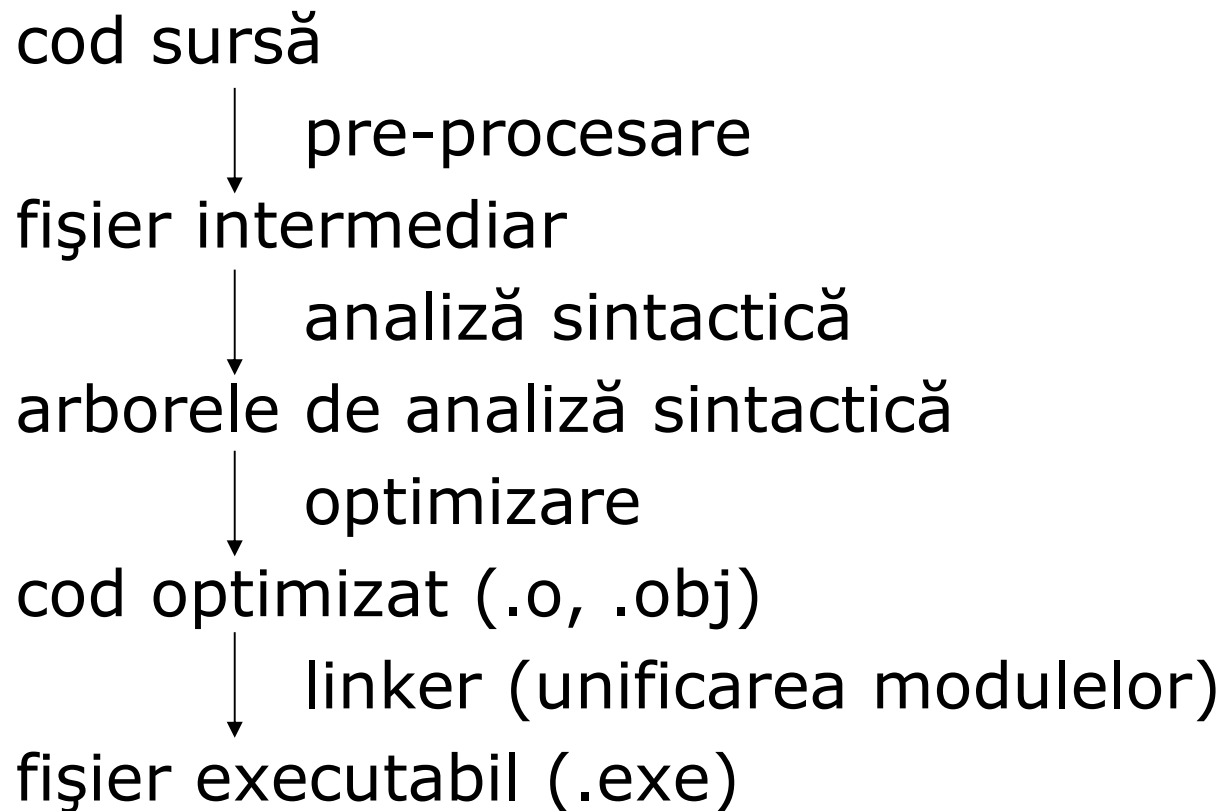
int globalVar;

int main(){
    int a;
    cin >> a;
    cout << a;
    return 0;
}
```

```
#ifndef MYHEADER_H
#define MYHEADER_H
    //data and method declarations
#endif
```

# Procesul de compilare

---



# Elemente de limbaj

---

- *Case sensitive* –  $a \neq A$
- Identificatori
  - șiruri de litere, cifre, “\_” care încep cu o literă
- Comentarii
  - `//` comentariu pe o singură linie
  - `/*` comentariu pe mai multe linii `*/`
- Declarații
  - introduc compilatorului un nume definit de către utilizator
  - nu se alocă memorie
  - declararea variabilelor:
    - `<tip de date> listă_identificatori`
    - tipuri de date:
      - pre-definite: `char`, `int`, `float`, `double`
      - derivate: referințe (&) și pointeri (\*)
      - definite de utilizator
    - specificatori
      - `short/long` – schimbă domeniul unui tip
      - `signed/unsigned` – precizează modul în care compilatorul folosește bitul de semn

```
int i;  
short int si;  
long int li;  
unsigned int ui;  
double d;  
long double ld;
```

# Elemente de limbaj

## □ Funcții

- declarare:

```
tip_fc nume_fc(listă_param);
```

- definire:

```
tip_fc nume_fc(listă_param){  
    //corpul funcției  
}
```

```
int maxim(int a, int b){  
    if (a > b)  
        return a;  
    else  
        return b;  
}
```

- funcția **main**

```
std {  
    int main()  
    int main(int argc, char** argv)  
    void main()  
    void main(int argc, char** argv)
```

→ pentru procesarea argumentelor  
din linia de comandă  
} fără returnarea unei valori  
(~procedură în Pascal)

# Elemente de limbaj

## □ Operatori

- aritmetici: +, -, \*, /, % (mod)
- pe biți
  - logical: &, | (or), ^(xor), ~(compl.)
  - shift: <<, >>
- logici: && (and), || (or), ! (not)
- atribuire: =
- compuși: +=, -=, \*=, /=, &=
- relaționali: ==, !=, <, >, <=, >=
- incrementare/decrementare: ++, --
  - formă pre-fixată: ++a; --a
  - formă post-fixată: a++; a--
- conversie explicită: (*tip*) operand
- dimensiune: *sizeof*
- condițional (ternar): ?
- rezoluție: ::

```
int a = 30; //global variab

int main(){
    int a = 5;
    int b = 3;
    double da = a;           //da = 5.0
    double x = 4.567;
    int y = (int) x;         // y = 4;

    int c = a / b;           //c = 1;
    double d = a / b;       //d = 1;
    double e = ((double) a) / b; //e = 1.6667

    b += c; //b = b + c

    int f = 5;
    int g = 5;
    cout << f++; //5 and f = 6
    cout << f; //6
    cout << ++g; //6 and g = 6
    cout << g; //6

    cout << "sizeof(int) = " << sizeof(a) << endl; //4
    cout << "sizeof(double) = " << sizeof(e) << endl; //8

    int max = a > b ? a : b;

    cout << "local variab a = " << a << endl;
    cout << "global variab a = " << ::a << endl;

    return 0;
}
```

# Elemente de limbaj

---

## □ Instrucțiuni

- instrucțiunea vidă

```
;
```

- expresii

```
a = b + c;  
m = maxim(a,b);  
d++;
```

- instrucțiunea compusă

```
{  
    int x;  
    int y = 5;  
    x = 2;  
    int m = maxim(x, y);  
}
```



# Elemente de limbaj

## □ Instrucțiuni

### ■ *if*

```
if (cond)
    statement1;
[else
    statement2;]
```

```
if (a > b)
    maxim = a;
else
    maxim = b;
```

### ■ *switch*

```
switch (expression){
    case c1:
        st1;
        [break;]
    case c2:
        st2;
        [break;]
    ...
    [default:
        st;]
}
```

```
char s = '+'; //-,*
switch (s){
    case '+':
        res = a + b;
        break;
    case '-':
        res = a - b;
        break;
    case '*':
        res = a * b;
        break;
    default:
        res = 0;
}
```

# Elemente de limbaj

## □ Instrucțiuni

### ■ *while*

```
while (cond){  
    statements;  
}
```

```
int n = 1234;  
while (n > 0){  
    int lastDigit = n % 10;  
    n = n / 10;  
    cout << lastDigit;  
}
```

### ■ *do-while*

```
do{  
    statements;  
}while (cond);
```

```
int n = 1234;  
do{  
    int lastDigit = n % 10;  
    n = n / 10;  
    cout << lastDigit;  
}while (n > 0);
```

### ■ *for*

```
for(expr_init; expr_cont;expr_step){  
    statements;  
}
```

```
int n = 1234;  
int lastDigit;  
for(lastDigit = n % 10; n > 0; n = n / 10){  
    cout << lastDigit;  
    lastDigit = n % 10;  
}  
cout << lastDigit;
```

# Elemente de limbaj

---

## □ Instrucțiuni

- *break*

```
int n = 12034;
while (n > 0){
    int lastDigit = n % 10;
    if (lastDigit == 0)
        break;
    n = n / 10;
    cout << lastDigit;
}
```

- *continue*

```
int n = 12034;
int lastDigit;
for(lastDigit = n % 10; n > 0; n = n / 10){
    cout << lastDigit;
    if (lastDigit == 0){
        cout << "a zero";
        continue;
    }
    lastDigit = n % 10;
}
cout << lastDigit;
```

# Variabile

---

## □ Declaraire

```
tip nume;
```

## □ Definiere (inițializare)

```
nume = valoare;
```

## □ E.g.

- declaraire
- inițializare  
sau
- declaraire și inițializare

```
void main(){  
  
    int i;  
    i = 10;  
  
    double d = 0.7;  
  
}
```

# Vizibilitatea variabilelor

- Scop
  - locul unde variabila este:
    - validă
    - creată
    - distrusă
  
- Variabile gloabale
  - definite în afara corpului funcției
  - disponibile pentru toate părțile programului
  - durata de viață – până la sf. programului
  - **extern**
    - pentru utilizarea în alt(e) fișier(e)
    - variabila există, chiar dacă compilatorul încă nu a "văzut-o" în fișierul curent pe care-l compilează

## file1.cpp

```
int globalVar;

void modify();

void myFunc(){
    int x = globalVar;
    cout << "x = " << x;
    globalVar++;
}

int main(){
    globalVar = 5;
    cout << "gv = " << globalVar; //5
    myFunc(); //5;
    cout << "gv = " << globalVar; //6
    modify();
    cout << "gv = " << globalVar; //7
    return 0;
}
```

## file2.cpp

```
extern int globalVar;

void modify(){
    globalVar++;
}
```

# Vizibilitatea variabilelor

---

## □ Variabile locale

- apar într-un anumit scop → variabile automate
- sunt "locale" unei funcții
- variabile registru
  - pentru creșterea vitezei de access
  - pot fi declarate doar în interiorul unui bloc sau ca și parametri
    - nu pot fi definite variabile statice sau variabile globale de tip registru
  - nu pot primi sau calcula adresa unei variabile de tip registru

# Vizibilitatea variabilelor

## □ Variabile statice

- alocare în memoria programului
  - variabile ne-statice sunt alocate pe stivă
- o singură inițializare automată
- *static* înseamnă:
  - inaccesibile în afara scopului funcției
    - pentru memorarea anumitor informații despre variabilele locale de la un apel la altul al funcției
  - inaccesibile în afara fișierului
    - pentru variabile globale și funcții
  - clase

## file.cpp

```
void fc(int a){
    static int x;
    int y = 0;
    cout << " before: x = " << x << endl;
    cout << " before: y = " << y << endl;
    x += a;
    y += a;
    cout << " after: x = " << x << endl;
    cout << " after: y = " << y << endl;
}

int main(){
    fc(2);
    fc(2);
    fc(2);
    return 0;
}
```

## file1.cpp

```
static int globalVar;

void modify();

void myFunc(){
    int x = globalVar;
    cout << "x = " << x;
    globalVar++;
}

int main(){
    globalVar = 5;
    cout << "gv = " << globalVar; //5
    myFunc(); //5;
    cout << "gv = " << globalVar; //6
    myFunc(); //6;
    cout << "gv = " << globalVar; //7
    //modify(); //error
    cout << "gv = " << globalVar;
    return 0;
}
```

## file2.cpp

```
extern int globalVar;

void modify(){
    globalVar = 10;
}
```

# Constante

---

## □ Tipologie

### ■ numerice

- decimale: 123, 111
- octale: **0**77
- hexadecimale: **0X**AABE
- În virgulă flotantă: 2.3456, 6.023**e**23

### ■ caracter

- imprimabile: `'a'`, `'P'`, `''`
- funcționale: `'\b'=backspace`, `'\r'=return`, `'\n'=newline`,  
`'\ '=apostrophe`, `'\\'=backslash`, `'\v'=verticalTab`,  
`'\f'=newPage`, `'\0'=null`

### ■ șiruri de caractere

- `"mesaj"`



# Constante

---

## □ Declaraire/definire

- cuvânt rezervat *const*
- într-un anumit scop (similar variabilelor obișnuite)
- trebuie să fie inițializate

## □ Exemple:

- Decimale
- Octale
- Hexadecimale
- Caractere

```
const double PI = 3.14;
const int alpha = 5;

const int beta = 06;
//const int theta = 09;

const int hexa1 = 0XA1B;
const int hexa2 = 0xcba;

const char c1 = 'A';
const char c2 = ';';
```

# Funcții

## □ Declaraire

```
<tip_returnat> <nume_fc>([<lista_param_formali>]);
```

■ *e.g.*

```
void func1();  
void func2(void);  
void func3(int a, double b);  
double func4(int a, int b);  
int func5(int a, ...);
```

## □ Definiere

```
<tip_returnat> <nume_fc>([<lista_param_formali>]){  
    instrucțiuni;  
}
```

■ *e.g.*

```
void func3(int a, double b){  
    cout << a << b;  
}  
  
double func4(int a, int b){  
    if (a < b)  
        return ((double) a) / 10;  
    return 0.0;  
}
```

# Funcții

## □ Constrângeri

- lista param. actuali trebuie să respecte lista param. formali
  - tipul parametrilor
  - numărul de parametri
  - ordinea parametrilor
- lista param. formali trebuie să conțină cel puțin tipul parametrilor
- *void* ~ nimic sau orice
- dacă o funcție are tip, ea trebuie să conțină cel puțin o instrucțiune *return*
- instrucțiunea *return* este ultima instr. executată
- o funcție poate conține mai multe instr. *return*

```
double avg(int a, int b){
    return ((double)(a + b)) / 2;
}

int main(){
    int x = 5;
    int y = 6;
    double m = avg(x, y);
    return 0;
}
```

```
double avg(int, int);

int main(){
    int x = 5;
    int y = 6;
    double m = avg(x, y);
    return 0;
}

double avg(int a, int b){
    return ((double)(a + b)) / 2;
}
```

```
int compare(int a, int b){
    if (a < b)
        return -1;
    else
        if (a > b)
            return 1;
    return 0;
    int d = a + b;
    cout << d;
}
```

# Funcții

## □ Tipologie

### ■ inline

- expandate în blocul de apel
- funcții simple
- cantitate redusă de resurse implicată în apelul fc.
- compilatorul este liber să decidă

### ■ cu argumente implicite (param. impliciți)

- param. cu valori predefinite
- argument implicit = o valoare dată la declarare pe care compilatorul o folosește automat dacă nu este furnizată o altă valoare
- dacă la apel se transmite o altă valoare, compilatorul o va utiliza pe aceasta
- argumentele implicite trebuie să fie cele mai din dreapta din lista param.

```
//inline int minim(int a, int b);  
  
inline int minim(int a, int b){  
    return a + b;  
}
```

```
void putInStack(int start, int capacity = 5){  
    for(int i = 0; i < capacity; i++){  
        cout << start + i << endl;  
    }  
}  
  
void main(){  
    putInStack(3); //3 4 5 6 7  
    putInStack(3, 3); //3 4 5  
}
```

# Funcții

## □ Transmiterea parametrilor

- prin valoare
- prin adresă (&)
  - Orice modificare a adresei *în interiorul* funcției va determina modificări ale parametrului *înafara* funcției

## □ Supraîncărcarea funcțiilor

- Aceeași funcție, dar cu parametri diferiți ca
  - tip
  - număr

```
void argByValue(int a){
    a++;
    cout << a;
}
void argByRef(int &a){
    a++;
    cout << a;
}
void main(){
    int x = 5;
    argByValue(x); //6
    cout << x; //5
    x = 3;
    argByRef(x); //4
    cout << x; //4
}
```

```
int sum(int a, int b){
    return a + b;
}
float sum(float a, float b){
    return a + b;
}
float sum(int a, int b, float c){
    return a + b + c;
}
void main(){
    int x = 2;
    int y = 3;
    float z = 4.5;
    float t = 6.7;
    cout << sum(x, y); //5
    cout << sum(z, t); //11.2
    cout << sum(x, y, z); //9.5
}
```

# Tipuri de date structurate

---

- Vectori
- Structuri
- Unions
- Enumerări

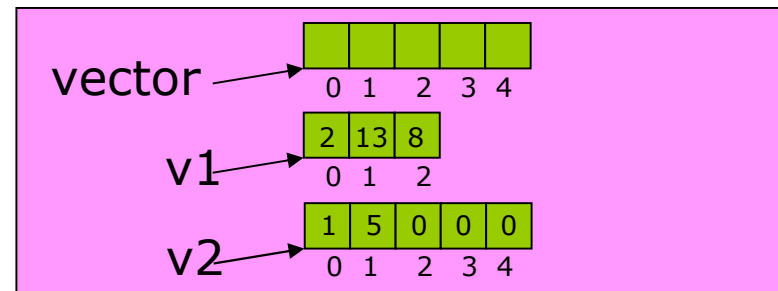
# Vectori

## □ Declaraire

```
<tip_elem> <identif>[dim_1][dim_2]...[dim_n]
```

□ E.g.

```
int vector[5];  
int matrix[3][5];  
int v1[] = {2, 13, 8};  
int v2[5] = {1, 5};
```



## □ Utilizare

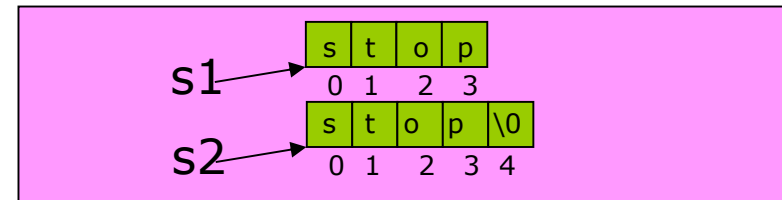
- acces la elemente: cu operatorul []
  - v1[1], v2[3], matrix[i][j]
- primul elemente se află pe poziția 0

```
for (int i = 0; i < 5; i++)  
    cout << v2[i];
```

# Vectori

## □ Vectori de caractere

```
char s1[] = {'s', 't', 'o', 'p'};  
char s2[] = "stop";
```



- header-e: <cstring> sau <string.h>
  - strlen, strcpy, strcat, strstr, strcmp

## □ Vectori ca și parametri în funcții

- apelați prin referință

```
void change(int &len, int v[]){  
    len = 3;  
    for(int i = 0; i < len; i++){  
        v[i] = i;  
    }  
}  
void main(){  
    int n = 2;  
    int vect[10];          //vect = {-89..., -87..., -89..., -87..., ...}  
    change(n, vect);     //vect = {0, 1, 2, -89..., -87..., ...}  
}
```



# Structuri

- Similar *record*-urilor din Pascal
- Declarare

```
struct [<identif>] {  
    tip1 câmp1;  
    tip2 câmp 2;  
    ...  
}[listă_variab];
```

- Utilizare:
  - declararea unei date *struct*:
  - acces la câmpuri:

```
struct Flower{  
    char name[10];  
    int price;  
};  
  
void main(){  
    Flower f;  
    f.price = 10;  
}
```

```
struct Flower{  
    char name[10];  
    int price;  
}f;  
  
void main(){  
    f.price = 10;  
}
```

- Structuri ca și parametri în funcții

```
struct Flower{  
    char name[10];  
    int price;  
}f;  
  
void changePrice(Flower &f1){  
    f1.price *= 2;  
}  
  
void main(){  
    f.price = 10;  
    changePrice(f);  
    cout << f.price;    //20  
}
```

# Enumerări

## □ Declaraire

```
enum <identif> {id0[=expr0], id1[=expr1], ..., idn[=exprn]}[listă_var]
```

## □ Utilizare

```
enum SpringMonth {March, April, May};

void main(){
    SpringMonth m = April;
    int flowerNo = 0;
    switch (m){
        case March:
            flowerNo += 5;
            break;
        case April:
            flowerNo += 10;
            break;
        default:
            flowerNo += 20;
    }
    cout << flowerNo;
}
```

```
enum Flag{red, yellow, green};

void useFlag(Flag f){
    switch (f){
        case red:
            cout << f << " stop" << endl;
            break;
        case yellow:
            cout << f << " wait" << endl;
            break;
        default:
            cout << f << " go" << endl;
    }
}

void main(){
    Flag myFlag = red;
    useFlag(myFlag);
}
```

# Unions

## □ Declarare

```
union <identif>{  
    type1 field1;  
    type2 field2;  
    ...}[var_list];
```

## □ Utilizare

```
union MyUnion(  
    int i;    //4b  
    char c;   //1b  
    float f;  //4b  
);  
  
void main(){  
    MyUnion u;  
    cout << sizeof(u); //4  
    MyStruct s;  
    cout << sizeof(s); //12  
}  
  
struct MyStruct(  
    int i;    //4b  
    char c;   //1b  
    float f;  //4b  
);
```

## □ Avantaje

- economie de memorie
- acumulare de date într-un singur spațiu;
- se ocupă spațiul necesar pentru cel mai larg element al **union**; acesta va da dimensiunea **union**

# Cursul următor

---

- Elemente de bază ale limbajului C++ (cont)
  - Referințe și pointeri
  - Vectori
  
- TAD-uri