UNIVERSITATEA BABEŞ-BOLYAI
Facultatea de Matematică şi 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++
  - Generalităţi
  - 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 instrucţiuni GOTO şi BREAK
- Utilizarea corectă a instrucţiunii FOR
- Nume sugestive pentru variabile
- Cod identat
- Utilizarea comentarilor
- Utilizarea unor programe simple, ierarhice, bazate pe structuri:
  - secvenţiale
  - selective (*if…then..else…endif*, *switch* or *case*)
  - repetitive (*while*, *repeat*, *for* or *do…until*)

Programare structurată la nivel superior

- Programare procedurală
  - subalgoritmi: funcţii şi proceduri
  - evitarea utilizarii 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 şi întreţinere uşoară
    - Complexitate ascunsă
- Programare orientată obiect

Exemplu: TAD

- Domeniu
  - Mulţimea tuturor valorilor posibile ale TAD-ului
- Operaţii
  - Iniţializare (creare şi alocare de memorie)
  - Conversie (din/în alte tipuri de date)
  - Selecţie (set & get)
  - Verificare de proprietăţi
  - Operaţii specifice TAD-ului

Programare orientată obiect (POO)

- De ce POO?
  - POO permite programatorilor să se concentreze
    - pe tipul de date, în primul rând
    - şi apoi pe metode
- Caracteristici
  - Abstractizarea datelor
    - încapsularea datelor şi operaţiilor
    - ascunderea datelor (reprezentării lor)
  - Moştenirea
    - reutilizarea codului
  - Polimorfismul
    - un obiect îşi poate schimba comportamentul în funcţie de starea sa

POO – elemente de bază

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

Def: POO este o metodă de proiectare şi dezvoltare a programelor (aplicaţiilor) cu următoarele caracteristici:

- *obiectele reprezintă elementele de bază cu care se lucrează*
- *fiecare obiect corespunde unui tip de date*
- *clasele interacţionează prin relaţii de moştenire şi compoziţie*

POO – avantaje şi limbaje

- Avantaje
  - complexitate redusă
  - reutilizarea codului
  - întreţinere uşoară
  - modificări uşor de realizat
  - cod lizibil

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

Elemente de bază ale limbajului C++

- Generalităţi
- Procesul de compilare
- Elemente de limbaj

Generalități

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

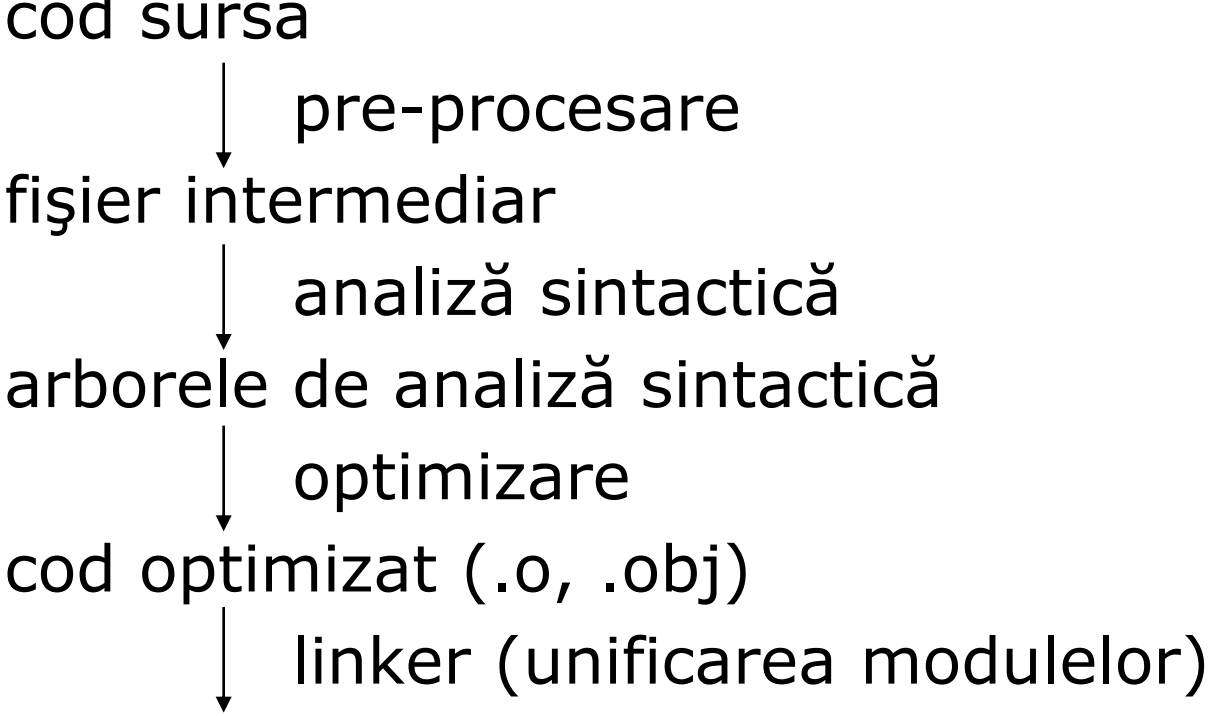
```
#include <iostream>
using namespace std;

int globalVar;

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

- Structura unui program
  - Directive de procesare
  - Declaraţii de date/variabile globale
  - Declaraţii/definiţii de funcţii
  - Funcţia principală (main)

  - Unul sau mai mutle fişiere cu extensiile: .cpp, .h
    - fişiere header
      - declaraţii, interfeţe
      - o legătură între biblioteci şi utilizator
    - fişiere sursă
      - definiţii, implementări
  - Directive de pre-procesare
    - ***#ifndef*** – pentru evitarea unor declaraţii multiple
    - în fişierele header: eg. *MyHeader.h*
    - ***#pragma once*** → fişierul sursă curent este inclus o singură dată într-o singură compilare

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

Procesul de compilare



Elemente de limbaj

- *Case sensitive* – a ≠ A
- Identificatori
  - şiruri de litere, cifre, "_" care încep cu o literă
- Comentarii
  - // comentariu pe o singură linie
  - /* comentariu pe
    mai multe linii */
- Declaraţii
  - 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: referinţe (&) şi pointeri (*)
      - definite de utilizator
    - specificatori
      - *short/long* – schimbă domeniul unui tip
      - *signed/unsigned* – precizează modul în care compilatorul foloseşte bitul de semn

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

Elemente de limbaj

- Funcţii
  - declarare:
    tip_fc nume_fc(listă_param);
  - definire:
    tip_fc nume_fc(listă_param){
    //corpul funcţiei
    }
  - funcţia ***main***

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

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 biţi
    - logical: &, | (or), ^(xor), ~(compl.)
    - shift: <<, >>
  - logici: && (and), || (or), ! (not)
  - atribuire: =
  - compuşi: +=, -=, *=, /=, &=
  - relaţionali: ==, !=, <, >, <=, >=
  - incrementare/decrementare: ++, --
    - formă pre-fixată: ++a; --a
    - formă post-fixată: a++; a—
  - conversie explicită: *(tip) operand*
  - dimensiune: *sizeof*
  - condiţional (ternar): *?*
  - rezoluţie: *::*

```
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

- Instrucţiuni
  - instrucţiunea vidă
  - expresii
  - instrucţiunea compusă

```
;
```

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

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

Elemente de limbaj

- Instrucţiuni
  - *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

- Instrucţiuni
  - *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

- Instrucţiuni
  - *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

- Declarare

```
tip nume;
```

- Definire (iniţializare)

```
nume = valoare;
```

- E.g.
  - declarare
  - iniţializare

    sau
  - declarare şi iniţializare

```
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 funcţiei
  - disponibile pentru toate părţile programului
  - durata de viaţă – până la sf. programului
  - ***extern***
    - pentru utilizarea în alt(e) fişier(e)
    - variabila există, chiar dacă compilatorul încă nu a "vazut-o" în fişierul 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 automatice
  - sunt "locale" unei funcţii
  - variabile registru
    - pentru creşterea vitezei de access
    - pot fi declarate doar în interiorul unui bloc sau ca şi 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 iniţializare automată
  - *static* înseamnă:
    - inaccesibile în afara scopului funcţiei
      - pentru memorarea anumitor informaţii despre variabilele locale de la un apel la altul al funcţiei
    - inaccesibile în afara fişierului
      - pentru variabile globale şi funcţii
    - 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’, ‘”’
    - funcţionale: '\b'=*backspace*, '\r'=*return*, '\n'=*newline*, '\''=*apostrophe*, '\\'=*backslash*, '\v'=*verticalTab*, '\f'=*newPage*, '\0'=*null*
  - şiruri de caractere
    - “mesaj”

Constante

- Declarare/definire
  - cuvânt rezervat *const*
  - într-un anumit scop (similar variabilelor obişnuite)
  - trebuie să fie iniţializate

- 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

- Declarare

```
<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,...);
```

- Definire

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

  - *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 conţină cel puţin tipul parametrilor
  - *void* ~ nimic sau orice
  - dacă o funcţie are tip, ea trebuie să conţină cel puţin o instrucţiune *return*
  - instrucţiunea *return* este ultima instr. executată
  - o funcţie poate conţine 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
    - funcţii simple
    - cantitate redusă de resurse implicată în apelul fc.
    - compilatorul este liber să decidă
  - cu argumente implicite (param. impliciţi)
    - param. cu valori predefinite
    - argument implicit = o valore dată la declarare pe care compilatorul o foloseşte 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* funcţiei va determina modificări ale parametrului *înafara* funcţiei

```
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
}
```

- Supraîncărcarea funcţiilor
  - Aceeaşi funcţie, dar cu parametri diferiţi ca
    - tip
    - număr

```
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

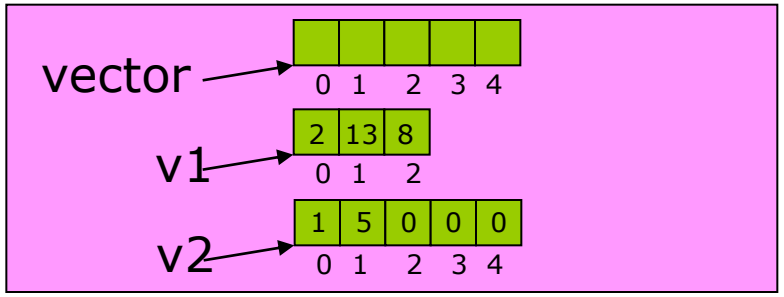
Vectori

- Declarare

```
<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};
```



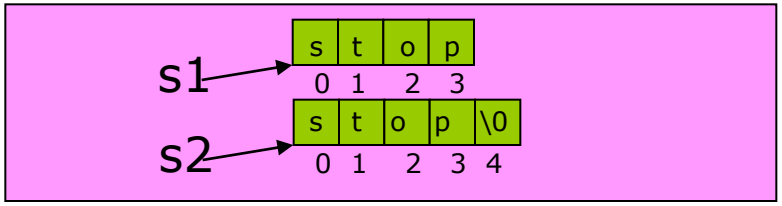
- Utilizare
  - acces la elemente: cu operatorul []
    - v1[1], v2[3], matrix[i][j]
  - primul elemente se află pe poziţia 0

```
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 şi parametri în funcţii
  - apelaţi prin referinţă

```
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 şi parametri în funcţii

```
struct Flower{
    char name[10];
    int price;
}f;

void changePrice(Flower &fl){
    fl.price *= 2;
}

void main(){
    f.price = 10;
    changePrice(f);
    cout << f.price;     //20
}
```

Enumerări

- Declarare

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
};

struct MyStruct{
    int i;      //4b
    char c;     //1b
    float f;    //4b
};

void main(){
    MyUnion u;
    cout << sizeof(u);  //4
    MyStruct s;
    cout << sizeof(s);  //12
}
```

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

Cursul următor

- Elemente de bază ale limbajului C++ (cont)
  - Referinţe şi pointeri
  - Vectori
- TAD-uri