Facultatea de Matematică şi Informatică
Universitatea din Bucureşti

Programare orientată pe obiecte

- suport de curs -

Andrei Păun
Anca Dobrovăț

An universitar 2024 – 2025
Semestrul II
Seriile 13, 14, 15

Curs 2

Agenda cursului

- Recapitularea discuţiilor din cursul anterior (Generalităţi despre curs, Reguli de comportament)
- Generalităţi despre OOP (Principiile programării orientate pe obiecte)
  - Clase, obiecte, modificatori de acces, functii si clase prieten, constructori / destructor

Generalităţi despre curs

1. Curs – seria 15: luni (10 -13), seria 14: marti (19 – 12), seria 13: vineri (9 - 12)

2. Laborator – pe semigrupe, in fiecare saptamana

3. Seminar - o data la 2 saptamani

COLOCVIU: Data va fi anuntata (in principiu, ultima saptamana de curs) – se sustine fizic in facultate;

EXAMEN SCRIS: 12 iunie 2025, ora 9.00 – se sustine fizic in facultate

Daca cineva are o problema cu aceste date il/o rog sa ne anunte

In 2 saptamani datele acestea sunt fixate/finalizate

1. Scurta recapitulare C++

Completări aduse de limbajul C++ faţă de limbajul C

- C++98: a definit standardul inițial, toate chestiunile de limbaj, STL
- C++03: bugfix o unică chestie nouă: value initialization
- C++11: initializer lists, rvalue references, moving constructors, lambda functions, final, constant null pointer, etc.
- C++14: generic lambdas, binary literals, auto, variable template, etc.
- C++17: schimbări la STL pentru paralelizare, nested namespaces, inline variables, eliminat trigraphs, functii deprecated
- C++20: concepts, modules, three-way comparison, coroutines, constinit, volatile deprecated
- C++23: <expected>, <stacktrace>
- Următoarea plănuită în 2026 (C++2c)

1. Scurta recapitulare C++

Completări aduse de limbajul C++ faţă de limbajul C

- **Intrări și ieșiri**
  - Obiectele **cin** și **cout**, în plus față de funcțiile scanf și printf din limbajul C.
  - Nu necesită specificarea formatelor.
  - Pentru I/O type-safe cu format: <format> (C++20)
- **Comentarii pe o singură linie**
- **Citirea string-urilor până la primul caracter alb**

1. Scurta recapitulare C++

Supraîncărcarea funcţiilor (un caz de *Polimorfism la compilare*)

Utilizarea mai multor funcţii care au acelaşi nume

Identificarea se face prin numărul de parametri şi tipul lor. **Tipul de întoarcere nu e suficient pentru a face diferența**

Simplicitate/corectitudine de cod

1. Scurta recapitulare C++

Supraîncărcarea funcţiilor

Ambiguitati pentru polimorfism de functii

```
int myfunc(int i); // Error: differing return types are
float myfunc(int i); // insufficient when overloading.
```

```
void f(int *p);
void f(int p[]); // error, *p is same as p[]

void f(int x);
void f(int& x);
```

```
int myfunc(int i) {    return i; }
int myfunc(int i, int j = 0) {   return i*j; }

Apel: cout << myfunc(10);
```

1. Scurta recapitulare C++

Supraîncărcarea funcţiilor

Ambiguitati pentru polimorfism de functii

Obs.

```
int myfunc(double d);  // ...
cout << myfunc('c'); // not an error, conversion applied
```

Dar…

```
float myfunc(float i){   return i;}
double myfunc(double i){   return -i;}

int main(){
cout << myfunc(10.1) << " "; // unambiguous, calls myfunc(double)
cout << myfunc(10); // ambiguous
 return 0; }
```

- problema nu e de definire a functiilor myfunc,
- problema apare la apelul functiilor

1. Scurta recapitulare C++

Funcţii cu valori implicite

Într-o funcţie se pot declara valori implicite pentru unul sau mai mulţi parametri.

La apel se poate omite specificarea valorii pentru acei parametri formali care au declarate valori implicite.

Argumentele cu valori implicite trebuie să fie amplasate la sfârşitul listei.

Valorile implicite se specifică o singură dată în definiţie (de obicei în prototip).

1. Scurta recapitulare C++

Funcţii cu valori implicite

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

void f (int a, int b = 12); // prototip cu mentionarea valorii implicite pentru b

int main()
{
  f(1);
  f(1,20);
return 0;
}

void f (int a, int b) {  cout<<a<<" - "<<b<<endl; }
```

1. Scurta recapitulare C++

Alocare dinamica

- new, delete (operatori nu funcții)
- se pot folosi încă malloc() şi free() dar vor fi deprecated în viitor

- new: alocă memorie şi întoarce un pointer la începutul zonei respective
- delete: sterge zona respectivă de memorie
- la eroare se “aruncă” excepţia bad_alloc din <new>

1. Scurta recapitulare C++

Alocare dinamica

```
int *pi;

pi=new int;
delete pi; // elibereaza  zona adresata de pi -o considera neocupata

pi=new int(2);// aloca zona si initializeaza zona cu valoarea 2

pi=new int[2]; // aloca un vector de 2 elemente  de tip intreg
delete [ ] pi; //elibereaza intreg vectorul

    //-pentru new se foloseste delete
    //- pentru new [ ] se foloseste delete [ ]
```

1. Scurta recapitulare C++

Pointerii in C/C++

- O variabilă care ţine o adresă din memorie
- Are un tip, compilatorul știe tipul de date către care se pointează
- Operațiile aritmetice țin cont de tipul de date din memorie
- Pointer ++ == pointer+sizeof(tip)
- Definiție: tip *nume_pointer;
  - Merge şi tip* nume_pointer;

1. Scurta recapitulare C++

Pointerii in C/C++

```c
#include <stdio.h>
int main(void)
{
    double x = 100.1, y;
    int *p;
    /* The next statement causes p (which
    is an    integer pointer) to point to a
    double. */
    p = (int *)&x;
    /* The next statement does not operate
    as    expected. */ y = *p;
    printf("%f", y); /* won't output 100.1 */
    return 0;
}
```

- Schimbarea de tip nu e controlată de compilator
- în C++ conversiile trebuiesc făcute cu schimbarea de tip

1. Scurta recapitulare C++

Pointerii in C/C++

pointeri şi array-uri

- numele array-ului este pointer
- lista[5]==*(lista+5)

- array de pointeri, numele listei este un pointer către pointeri (dublă indirectare)
- int **p; (dublă indirectare)

1. Scurta recapitulare C++

Const in C++

- idee: să se elimine comenzile de preprocesor #define
- #define făceau substituție de valoare
- se poate aplica la pointeri, argumente de funcții, param de întoarcere din funcții, obiecte, funcții membru
- fiecare dintre aceste elemente are o aplicare diferită pentru const, dar sunt în aceeași idee/filosofie

1. Scurta recapitulare C++

Const in C/C++

```
// Using const for safety
#include <iostream>
using namespace std;

const int i = 100;  // Typical constant
const int j = i + 10; // Value from const expr
long address = (long)&j; // Forces storage
char buf[j + 10]; // Still a const expression

int main() {
  cout << "type a character & CR:";
  const char c = cin.get(); // Can't change //valoarea
nu e cunoscuta la compile time si necesita storage
  const char c2 = c + 'a';
  cout << c2;
  // ...
}
```

- dacă știm că variabila nu se schimbă să o declaram cu const
- dacă încercam să o schimbăm primim eroare de compilare

1. Scurta recapitulare C++

Pointeri si Const in C/C++

- const poate fi aplicat valorii pointerului sau elementului pointat
- const se alatură elementului cel mai apropiat

const int* u;

- u este pointer către un int care este const

int const* v; la fel ca mai sus

1. Scurta recapitulare C++

Pointeri si Const in C/C++

- pentru pointeri care nu își schimbă adresa din memorie

```
int d = 1;
int* const w = &d;
```

- w e un pointer constant care arată către întregi+inițializare

1. Scurta recapitulare C++

Pointeri si Const in C/C++

```
//: C08:ConstPointers.cpp
const int* u;
int const* v;

int d = 1;

int* const w = &d;

const int* const x = &d; // (1)
int const* const x2 = &d; // (2)

int main() {} ///:~
```

1. Scurta recapitulare C++

Pointeri si Const in C/C++

- se poate face atribuire de adresă pentru obiect non-const către un pointer const

- nu se poate face atribuire pe adresă de obiect const către pointer non-const

1. Scurta recapitulare C++

Pointeri si Const in C/C++

```
int d = 1;

const int e = 2;

int* u = &d; // OK -- d not const

//! int* v = &e; // Illegal -- e const

int* w = (int*)&e; // Legal but bad practice

int main() {} ///:~
```

1. Scurta recapitulare C++

Const si argumente de funcții, param de întoarcere

- apel prin valoare cu const: param formal nu se schimbă în functie
- const la întoarcere: valoarea returnată nu se poate schimba
- dacă se transmite o adresă: promisiune că nu se schimbă valoarea la adresa respectivă

1. Scurta recapitulare C++

Const si argumente de funcții, param de întoarcere

```
void f1(const int i) {
i++; // Illegal -- compile-time error
}
```

cod mai clar echivalent mai jos:

```
void f2(int ic) {
const int& i = ic;
i++; // Illegal -- compile-time error
}
```

1. Scurta recapitulare C++

Const si argumente de funcții, param de întoarcere

```
// Returning consts by value
// has no meaning for built-in types

int f3() { return 1; }
const int f4() { return 1; }

int main() {
  const int j = f3(); // Works fine
  int k = f4(); // But this works fine too!
}
```

1. Scurta recapitulare C++

Tipul referinta

O referinţă este, in esenta, un pointer implicit, care actioneaza ca un alt nume al unui obiect (variabila).

```
int i;
int *pi,j;
```

int & ri=i; //ri este alt nume pentru variabila I

```
pi=&i;  // pi este adresa variabilei I
*pi=3;   //in zona adresata de pi se afla valoarea 3
```

Pentru a putea fi folosită, o referinţă trebuie iniţializată in momentul declararii, devenind un alias (un alt nume) al obiectului cu care a fost iniţializată.

1. Scurta recapitulare C++

Tipul referinta

```
int a = 20;
int& ref = a;
cout<<a<<" "<<ref<<endl; // 20 20
```

ref++;

```
cout<<a<<" "<<ref<<endl; // 21 21
```

Obs: spre deosebire de pointeri care la incrementare trec la un alt obiect de acelasi tip, incrementarea referintei implica, de fapt, incrementarea valorii referite.

1. Scurta recapitulare C++

Tipul referinta

```
int a = 20;
int& ref = a;
cout<<a<<" "<<ref<<endl; // 20 20

int b = 50;
ref = b;

ref--;
cout<<a<<" "<<ref<<endl; // 49 49
```

Obs: in afara initializarii, nu puteti modifica obiectul spre care indica referinta.

1. Scurta recapitulare C++

Tipul referinta

- o variabilă de tip referință trebuie să fie inițializată când este definită
- membrii unei clase de tip referință trebuie inițializați în constructori
- un parametru de funcție de tip referință se inițializează implicit la apel
- referințele nule sunt interzise intr-un program C++ valid.
  - putem întoarce referințe dangling (warning la compilare); // dangling – un pointer/referinta care “arata” catre o zona de memorie deja dezalocata
- nu se poate obține adresa unei referințe (&ref ne dă adresa variabilei)
  - intern sunt tot pointeri, dar așa e mai greu să îi "stricăm"
- se pot crea tablouri de referințe cu std::reference_wrapper

întoarcere de referințe

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

char &replace(int i); // return a reference

char s[80] = "Hello There";

int main()
{
    replace(5) = 'X'; // assign X to space
after Hello
    cout << s;
    return 0;
}

char &replace(int i)
{
    return s[i];
}
```

- putem face atribuiri către apel de funcție
- replace(5) este un element din s care se schimbă
- e nevoie de atenție ca obiectul referit să nu iasă din scopul de vizibilitate

1. Scurta recapitulare C++

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++

```
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;
}
```

1. Scurta recapitulare C++

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 ⇒ modificarile parametrului nu au efect asupra argumentului;
- apel prin referinta - in parametru este copiata adresa unui argument ⇒ modificarile parametrului au efect asupra argumentului.
- functiile, cu exceptia celor de tip void, pot fi folosite ca operand in orice expresie valida.

1. Scurta recapitulare C++

Functii in structuri

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

struct test {
  int x;
  void (*afis)(struct test *this);
};

void afis_implicit(struct test *this) {
  printf("x= %d",this->x);
}

int main() {
  struct test A = {3, afis_implicit};
  A.afis(&A);
  return 0;
}
```

Q: Exista un mecanism prin care putem avea totusi functii in structuri in C?
A: Da, utilizand pointerii la functii

Q: Codul alaturat este valid si in C++?
A: Nu, pentru ca am folosit “this” ca identificator (mai tarziu despre “this”).

Q: Daca putem folosi, totusi, functii in structuri in C, de ce folosim clase?
A: Pentru ca e dificil de emulat ascunderea informatiei, principiu de baza in POO.

1. Scurta recapitulare C++

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; }
```

2. Principiile programarii orientate pe obiecte

Clasa

Obiect

Incapsularea

Modularitatea

Ierarhizarea.

Polimorfism la compilare si polimorfism la executie, etc.

2. Principiile programarii orientate pe obiecte

Obiecte

- au stare şi acțiuni (metode/funcţii)
- au interfață (acțiuni) şi o parte ascunsă (starea)
- Sunt grupate în clase, obiecte cu aceleași proprietăți
- Un **program orientat obiect** este o colecție de obiecte care interactionează unul cu celălalt prin mesaje (aplicand o metodă).

2. Principiile programarii orientate pe obiecte

Clase

O **clasa** defineste atribute si metode.

```
class X{
        //date membre
        //metode (functii membre – functii cu argument implicit obiectul curent)
        };
```

- menționează proprietățile generale ale obiectelor din clasa respectivă
- folositoare la incapsulare (ascunderea informației)
- reutilizare de cod: moștenire

2. Principiile programarii orientate pe obiecte

Clase

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

2. Principiile programarii orientate pe obiecte

Clase

```
class nume_clasă {
        private variabile şi funcţii membru
specificator_de_acces:
        variabile şi funcţii membru
specificator_de_acces:
        variabile şi funcţii membru
// ...
specificator_de_acces:
        variabile şi funcţii membru
} listă_obiecte;
```

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

2. Principiile programarii orientate pe obiecte

Clase

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

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

2. Principiile programarii orientate pe obiecte

Clase

- 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

2. Principiile programarii orientate pe obiecte

Clase

- variabilele de instanta (instance variabiles)

- membri de tip date ai clasei
  - in general private
  - pentru viteza se pot folosi “public” dar **NU LA ACEST CURS**

2. Principiile programarii orientate pe obiecte

Clase

```
#define SIZE 100
// This creates the class stack.
class stack {
        int stck[SIZE];
        int tos;
public:
        void init();
        void push(int i);
        int pop();
};
```

- init(), push(), pop() sunt funcții membru
- stck, tos: variabile membru

2. Principiile programarii orientate pe obiecte

Clase

- se creează un tip nou de date
- un obiect instanţiază clasa
- funcțiile membru sunt date prin semnătură
- pentru definirea fiecărei funcții se folosește ::

stack mystack;

```
void stack::push(int i)  {
        if(tos==SIZE) {
                cout << "Stack is full.\n";
        return; }
        stck[tos] = i;
        tos++;
}
```

2. Principiile programarii orientate pe obiecte

Clase

- :: scope resolution operator
- şi alte clase pot folosi numele push() şi pop()
- după instantiere, pentru apelul push()

- mystack.push(5);

stack mystack;

- programul complet în continuare

2. Principiile programarii orientate pe obiecte

Clase

```
#include <iostream>
using namespace std;
#define SIZE 100
// This creates the class stack.
class stack {
        int stck[SIZE];
        int tos;
public:
        void init();
        void push(int i);
        int pop();
};
void stack::init()
{
        tos = 0;
}
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 stack1, stack2; // create two stack objects
        stack1.init();
        stack2.init();
        stack1.push(1);
        stack2.push(2);
        stack1.push(3);
        stack2.push(4);
        cout << stack1.pop() << " ";
        cout << stack1.pop() << " ";
        cout << stack2.pop() << " ";
        cout << stack2.pop() << "\n";
        return 0;
}
```

2. Principiile programarii orientate pe obiecte

Incapsularea

- ascunderea de informaţii (data-hiding);
- separarea aspectelor externe ale unui obiect care sunt accesibile altor obiecte de aspectele interne ale obiectului care sunt ascunse celorlalter obiecte;
- defineşte apartenenţa unor proprietăţi şi metode faţă de un obiect;
- doar metodele proprii ale obiectului pot accesa starea acestuia.

2. Principiile programarii orientate pe obiecte

Incapsularea (ascunderea informatiei)

foarte importanta
public, protected, private

Avem acces?	**public**	**protected**	**private**
Aceeasi clasa	da	da	da
Clase derivate	da	da	nu
Alte clase	da	nu	nu

2. Principiile programarii orientate pe obiecte

Incapsularea (ascunderea informatiei)

```
class Test  {
  private:
int x;
  public:
  void set_x(int a) {x = a;}
};

int main() {
  Test t;
  t.x = 34; // inaccesibil
  t.set_x(34); // ok
  return 0;
}
```

2. Principiile programarii orientate pe obiecte

- **Agregarea** (ierarhia de obiecte) compunerea unui obiect din mai multe obiecte mai simple. (relaţie de tip “has a”)

- **Moştenirea** (ierarhia de clase) relaţie între clase în care o clasă mosteneste structura şi comportarea definită în una sau mai multe clase (relaţie de tip “is a” sau “is like a”);

2. Principiile programarii orientate pe obiecte

Agregarea

```
class Profesor
{
string nume;
int vechime;
};

class Curs
{
  string denumire;
  Profesor p;
};
```

2. Principiile programarii orientate pe obiecte

Moștenire

- multe obiecte au proprietăți similare
- reutilizare de cod

2. Principiile programarii orientate pe obiecte

Moștenire

- terminologie
  - clasă de bază, clasă derivată
  - superclasă subclasă
  - părinte, fiu

- mai târziu: funcții virtuale, identificare de tipuri in timpul rulării (RTTI)

2. Principiile programarii orientate pe obiecte

Moștenire

- încorporarea componentelor unei clase în alta
- refolosire de cod
- detalii mai subtile pentru tipuri şi subtipuri
- clasă de bază, clasă derivată
- clasa derivată conține toate elementele clasei de bază, mai adăugă noi elemente

2. Principiile programarii orientate pe obiecte

Moștenire

```
class building {
        int rooms;
        int floors;
        int area;
public:
        void set_rooms(int num);
        int get_rooms(); // …
};

// house e derivată din building
class house : public building {
        int bedrooms;
        int baths;
public:
        void set_bedrooms(int num);
        int get_bedrooms();};
```

tip acces: public, private, protected

mai multe mai târziu

public: membrii publici ai building devin publici pentru house

2. Principiile programarii orientate pe obiecte

Moștenire

- house NU are acces la membrii privați ai lui building
- așa se realizează encapsularea

- clasa derivată are acces la membrii publici ai clasei de baza şi la toți membrii săi (publici şi privați)

2. Principiile programarii orientate pe obiecte

Moștenire

```
class building {
        int rooms;
        int floors;
        int area;
public:
        void set_rooms(int num);
        int get_rooms(); //…};
```

```
class house : public building {
        int bedrooms;
        int baths;
public:
        void set_bedrooms(int num);
        int get_bedrooms();
        void set_baths(int num);
        int get_baths();
};
```

```
// school este de asemenea derivată din building
class school : public building {
        int classrooms;
        int offices;
ublic:
        void set_classrooms(int num);
        int get_classrooms();
        void set_offices(int num);
        int get_offices();
        };
```

Moștenire

```
void building::set_rooms(int num)
{ rooms = num; }
void building::set_floors(int num)
{ floors = num; }
void building::set_area(int num)
{ area = num; }
int building::get_rooms()
{ return rooms; }
int building::get_floors()
{ return floors; }
int building::get_area()
{ return area; }
void house::set_bedrooms(int num)
{ bedrooms = num; }
void house::set_baths(int num)
{baths = num; }
int house::get_bedrooms()
{ return bedrooms; }
int house::get_baths()
{ return baths; }
void school::set_classrooms(int num)
{ classrooms = num; }
void school::set_offices(int num)
{ offices = num; }
int school::get_classrooms()
{ return classrooms; }
int school::get_offices()
{ return offices; }
```

```
int main()
{
        house h;
        school s;
        h.set_rooms(12);
        h.set_floors(3);
        h.set_area(4500);
        h.set_bedrooms(5);
        h.set_baths(3);
        cout << "house has " << h.get_bedrooms();
        cout << " bedrooms\n"; s.set_rooms(200);
        s.set_classrooms(180);
        s.set_offices(5);
        s.set_area(25000);
        cout << "school has " << s.get_classrooms();
        cout << " classrooms\n";
        cout << "Its area is " << s.get_area();
        return 0;
}
```

house has 5 bedrooms
school has 180 classrooms
Its area is 25000

2. Principiile programarii orientate pe obiecte

Polimorfism

- tot pentru claritate/ cod mai sigur
- Polimorfism la compilare: ex. max(int), max(float)
- Polimorfism la execuție: RTTI

2. Principiile programarii orientate pe obiecte

Șabloane

- din nou cod mai sigur/reutilizare de cod
- putem implementa listă înlănțuită de
  - întregi
  - caractere
  - float
  - obiecte

Perspective

Curs 3

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