



Programare orientată pe obiecte

- suport de curs -

**Anca Dobrovăț
Andrei Păun**

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

Curs 1

Generalități despre curs

1. Curs – luni: 10 – 13 (seria 15), marti: 9 – 12 (seria 14) si vineri: orele 9 – 12 (seria 13)
2. Laborator - in fiecare saptamana; pe semigrupe
3. Seminar - o data la 2 saptamani (in conexiune cu laboratorul / vom reveni cu detalii)
4. Prezenta la curs nu e obligatorie!

Laborator – OBLIGATORIU

Să ne cunoaștem

Cine predă?

Curs: Anca Dobrovăț (seriile 13 si 15), Andrei Paun (seria 14)

- anca.dobrovat@fmi.unibuc.ro; apaun@fmi.unibuc.ro

Laboratoare: (se vor completa informatiie pentru 143 si 144)

- Marius Micluța – Câmpeanu (131 + seminar)
- Obreja Camelia (132, 133)
- Bahrim Dragos (134 + seminar)
- Vrinceanu Radu (141 + seminar)
- Campian Ioan Razvan (142 + seminar)
- Magureanu Livia (151 + seminar)
- Maxim Tiberiu (152 + seminar)
- TBA (143, 144)

Agenda cursului

1. Regulamente UB si FMI
2. Utilitatea cursului de Programare Orientata pe Obiecte
3. Prezentarea disciplinei
4. Primul curs

Agenda cursului

1. Regulamente UB si FMI

2. Utilitatea cursului de Programare Orientata pe Obiecte

3. Prezentarea disciplinei

4. Primul curs

1. Regulamente UB si FMI

Lucruri bine de stiut de studenti:

- <https://fmi.unibuc.ro/regulamente/>
- regulament privind activitatea studenților la UB: <https://unibuc.ro/wp-content/uploads/2024/04/Regulamentul-privind-activitatea-profesionala-a-studentilor-2024.pdf>
- http://fmi.unibuc.ro/ro/pdf/2015/consiliu/Regulament_etica_FMI.pdf

Se consideră **incident minor** cazul în care un student/ o studentă:
a. preia codul sursă/ rezolvarea unei teme de la un coleg/ o colegă și pretinde că este rezultatul efortului propriu;

Se consideră **incident major** cazul în care un student/ o studentă:
a. copiază la examene de orice tip;

3 incidente minore = un incident major = exmatriculare

Agenda cursului

1. Regulamente UB si FMI

2. Utilitatea cursului de Programare Orientata pe Obiecte

3. Prezentarea disciplinei

4. Primul curs

2. Utilitatea cursului de Programare Orientata pe Obiecte

PYPL PopularitY of Programming Language

Captura din: <http://pypl.github.io/PYPL>

Worldwide, Oct 2023 :				Worldwide, Feb 2024 :				Worldwide, Feb 2025 :				
Rank	Change	Language	Share	Rank	Change	Language	Share	Rank	Change	Language	Share	1-year trend
1		Python	28.05 %	1		Python	28.11 %	1		Python	29.85 %	+1.6 %
2		Java	15.88 %	2		Java	15.52 %	2		Java	15.15 %	-0.7 %
3		JavaScript	9.27 %	3		JavaScript	8.57 %	3		JavaScript	7.92 %	-0.8 %
4		C#	6.79 %	4	↑	C/C++	6.92 %	4		C/C++	7.19 %	+0.5 %
5		C/C++	6.59 %	5	↓	C#	6.73 %	5		C#	6.13 %	-0.5 %
6		PHP	4.86 %	6	↑	R	4.75 %	6		R	4.55 %	-0.1 %
7		R	4.45 %	7	↓	PHP	4.57 %	7		PHP	3.72 %	-0.8 %
8		TypeScript	2.93 %	8		TypeScript	2.78 %	8	↑↑	Rust	3.07 %	+0.6 %
9	↑↑	Swift	2.69 %	9		Swift	2.75 %	9	↑↑	Objective-C	2.86 %	+0.5 %
10	↓	Objective-C	2.29 %	10		Objective-C	2.37 %	10	↓↓	TypeScript	2.74 %	-0.1 %
11	↑↑	Rust	2.05 %	11		Rust	2.23 %	11	↓↓	Swift	2.46 %	-0.3 %
12	↓↓	Go	1.95 %	12		Go	2.04 %	12		Go	2.07 %	-0.1 %

Majoritatea pot fi considerate limbaje OO.

Limbaje destul de cunoscute care nu sunt OO sunt Go, Julia și Rust

2. Utilitatea cursului de Programare Orientata pe Obiecte

Paradigme de programare → Stil fundamental de a programa

Dictează:

- **Cum se reprezintă datele problemei** (variabile, funcții, obiecte, fapte, constrângeri etc.)
- **Cum se prelucrează reprezentarea** (atribuiri, evaluări, fire de execuție, continuări, fluxuri etc.)
- **Favorizează un set de concepte si tehnici de programare**
- **Influențează felul în care sunt gândiți algoritmi de rezolvare a problemelor**
- **Limbaje – în general multiparadigmă (ex: Python – imperativ, funcțional, orientat pe obiecte)**

Agenda cursului

1. Regulamente UB si FMI
2. Utilitatea cursului de Programare Orientata pe Obiecte
3. **Prezentarea disciplinei**
 - 3.1 Obiectivele disciplinei
 - 3.2 Programa cursului
 - 3.3 Bibliografie
 - 3.4 Regulament de notare si evaluare
4. Primul curs

3. Prezentarea disciplinei

3.1 Obiectivele disciplinei

- Curs de programare OO
- Oferă o **baza** de pornire pentru alte cursuri

- **Obiectivul general al disciplinei:**

Formarea unei imagini generale, preliminară, despre programarea orientată pe obiecte (POO).

- **Obiective specifice:**

- 1. Înțelegerea fundamentelor paradigmei programării orientate pe obiecte;
- 2. Înțelegerea conceptelor de clasă, interfață, moștenire, polimorfism;
- 3. Familiarizarea cu șabloanele de proiectare;
- 4. Dezvoltarea de aplicații de complexitate medie respectând principiile de dezvoltare ale POO;
- 5. Deprinderea cu noile facilități oferite de limbajul C++.

3. Prezentarea disciplinei

3.2 Programa cursului

1. Prezentarea disciplinei.

1.1 Principiile programării orientate pe obiecte.

1.2. Caracteristici.

1.3. Programa cursului, obiective, desfășurare, examinare, bibliografie.

2. Recapitulare limbaj C (procedural) și introducerea în programarea orientată pe obiecte.

2.1 Funcții, transferul parametrilor, pointeri.

2.2 Deosebiri între C și C++.

2.3 Supradefinirea funcțiilor, Operații de intrare/ieșire, Tipul referință, Funcții în structuri.

3. Prezentarea disciplinei

3.2 Programa cursului

3. Proiectarea ascendentă a claselor. Incapsularea datelor în C++.

3.1 Conceptele de clasă și obiect. Structura unei clase.

3.2 Constructorii și destructorul unei clase.

3.3 Metode de acces la membrii unei clase, pointerul this. Modificatori de acces în C++.

3.4 Declararea și implementarea metodelor în clasă și în afara clasei.

4. Supraîncărcarea funcțiilor și operatorilor în C++.

4.1 Clase și funcții friend.

4.2 Supraîncărcarea funcțiilor.

4.3 Supraîncărcarea operatorilor cu funcții friend.

4.4 Supraîncărcarea operatorilor cu funcții membru.

4.5 Observații.

3. Prezentarea disciplinei

3.2 Programa cursului

5. Conversia datelor în C++.

5.1 Conversii între diferite tipuri de obiecte (operatorul cast, operatorul= și constructor de copiere).

5.2 Membrii constanți și statici ai unei clase în C++.

5.3 Modificatorul const, obiecte constante, pointeri constanți la obiecte și pointeri la obiecte constante.

6. Tratarea excepțiilor în C++.

7. Proiectarea descendenței a claselor. Mostenirea în C++.

7.1 Controlul accesului la clasa de bază.

7.2 Constructori, destructori și moștenire.

7.3 Redefinirea membrilor unei clase de bază într-o clasă derivată.

7.4. Declarații de acces.

3. Prezentarea disciplinei

3.2 Programa cursului

8. Funcții virtuale în C++.

8.1 Parametrizarea metodelor (polimorfism la executie).

8.2 Funcții virtuale în C++. Clase abstracte.

8.3 Destructori virtuali.

9. Mostenirea multiplă și virtuală în C++

9.1 Moștenirea din clase de bază multiple.

9.2 Exemple, observații.

10. Controlul tipului în timpul rulării programului în C++.

10.1 Mecanisme de tip RTTI (Run Time Type Identification).

10.2 Moștenire multiplă și identificatori de tip (dynamic_cast, typeid).

3. Prezentarea disciplinei

3.2 Programa cursului

11. Parametrizarea datelor. Șabloane în C++. Clase generice

11.1 Funcții și clase Template: Definiții, Exemple, Implementare.

11.2 Clase Template derivate.

11.3 Specializare.

12. Biblioteca Standard Template Library - STL

12.1 Containere, iteratori și algoritmi.

12.2 Clasele string, set, map / multimap, list, vector, etc.

3. Prezentarea disciplinei

3.2 Programa cursului

13. Șabloane de proiectare (Design Pattern)

13.1 Definiție și clasificare.

13.2 Exemple de șabloane de proiectare (Singleton, Abstract Object Factory).

14. Recapitulare, concluzii, tratarea subiectelor de examen.

3. Prezentarea disciplinei

3.3 Bibliografie

1. Bruce Eckel. Thinking in C++ (2nd edition). Volume 1: Introduction to Standard C++. Prentice Hall, 2000.
2. Bruce Eckel, Chuck Allison. Thinking in C++ (2nd edition). Volume 2: Practical Programming. Prentice Hall, 2003.
3. Bjarne Stroustrup: The C++ Programming Language, Addison-Wesley, 3rd edition, 1997.
4. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

3. Prezentarea disciplinei

3.4 Regulament de notare și evaluare

Curs: 3 ore pe săptămână

Laborator: 2 ore pe săptămână.

Seminar: 1 ora pe săptămână => 2 ore, la fiecare 2 săptămâni.

Disciplina: semestrul II, durata de desfășurare de 14 săptămâni.

Materia este de nivel elementar - mediu și se bazează pe cunoștințele de C++ anterior dobândite.

Limbajul de programare folosit la curs și la laborator este **C++**.

3. Prezentarea disciplinei

3.4 Regulament de notare si evaluare

Programa disciplinei este împărțită în 14 cursuri.

Evaluarea studenților se face cumulativ prin:

- 3 lucrări practice (proiecte) – nr de proiecte depinde de tutorii de laborator dar se trece prin mai multe concepte
- Test practic (colocviu)
- Test scris

Toate cele 3 probe de evaluare sunt obligatorii.

Condiții de promovare - minim **nota 5 la fiecare** parte de evaluare enunțată - mai sus se păstrează la oricare din eventualele examene restante ulterioare aferente acestui curs.

3. Prezentarea disciplinei

3.4 Regulament de notare si evaluare

Nota laborator = medie aritmetica a celor 3 note obtinute pe proiecte (sau nota pe proiectul “mare”).

Pentru evidentierea unor lucrari practice, tutorele de laborator poate acorda un bonus de **pana la 2 puncte** la nota pe proiecte astfel calculata.

Studentii care **nu obtin cel putin nota 5 pentru activitatea pe proiecte nu pot intra in examen** si vor trebui sa refaca aceasta activitate, inainte de prezentarea la restanta.

3. Prezentarea disciplinei

3.4 Regulament de notare si evaluare

Testul practic (Colocviu) - in saptamana 14

- Consta dintr-un program care trebuie realizat individual intr-un timp limitat (90 de minute – si va avea un nivel mediu.
- Notare: de la 1 la 10, nu neaparat intregi (pot exista pana la 3 puncte bonus).

Testul practic este obligatoriu.

Studentii care **nu obtin cel putin nota 5 la testul practic de laborator nu pot intra in examen** si vor trebui sa il dea din nou, inainte de prezentarea la restanta.

3. Prezentarea disciplinei

3.4 Regulament de notare si evaluare

Testul scris:

Propunere: **12 iunie 2025** la ora 9:00

Studentii nu pot lua examenul decat daca obtin cel putin nota 5 la testul scris.

3. Prezentarea disciplinei

3.4 Regulament de notare si evaluare

Testul scris:

Consta dintr-un set de 18 intrebari

- 6 intrebari de teorie
- 12 intrebari practice.

Notarea testului scris se va face cu o nota de la 1 la 10 (1 punct din oficiu si cate 0,5 puncte pentru fiecare raspuns corect la cele 18 intrebari).

Studentii nu pot lua examenul decat daca obtin cel putin nota 5 la testul scris.

3. Prezentarea disciplinei

3.4 Regulament de notare si evaluare

Examenul se considera luat daca studentul respectiv a obtinut **cel putin nota 5 la fiecare** dintre cele 3 evaluari (activitatea practica din timpul semestrului, testul practic de laborator si testul scris).

In aceasta situatie, nota finala a fiecarui student se calculeaza ca medie ponderata intre notele obtinute la cele 3 evaluari, ponderile cu care cele 3 note intra in medie fiind:

- 25% - nota pe lucrarile practice (proiecte)
- 25% - nota la testul practic
- 50% - nota la testul scris

3. Prezentarea disciplinei

Modificari

- Laborator: notare mai “clara”
- Seminar: 0.5 bonus la nota de la examenul scris pentru max. 25% din studenti
- Prezenta la curs: 0.5 bonus la nota de la examenul scris pentru primii 25% dintre studenti KAHOOT
- - vom reveni cu detalii saptamana viitoare!!
- bonus dupa ce se promoveaza examenul scris

3. Prezentarea disciplinei

Kahoot

- Se va defini un nume unic de forma 131popescu (unde popescu este numele de familie si 131 este grupa)
- Daca sunt mai multi studenti cu acelasi nume in grupa respectiva (adaugati si initiala / initialele prenumelui)
- 131popescup si 131popescupr

Agenda cursului

1. Regulamente UB si FMI
2. Utilitatea cursului de Programare Orientata pe Obiecte
3. Prezentarea disciplinei
4. Primul curs
 - 4.1 Completări aduse de limbajul C++ față de limbajul C
 - 4.2 Principiile programarii orientate pe obiecte

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

- Bjarne Stroustrup în 1979 la Bell Laboratories in Murray Hill, New Jersey
- 7 revizii: 1998 ANSI+ISO, 2003 (corrigendum), 2011 (C++11/0x), 2014, 2017 (C++ 17/1z), 2020, 2023
- Următoarea plănuită în 2026 (C++2c)
- Versiunea 1998: Standard C++, C++98
- C++ modern: \geq C++11

4. Curs 1

4.1 Completări aduse de limbajul C++ față 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, nullptr literal, sincronizare cu C99, etc.
- C++14: generic lambdas, binary literals, auto, variable template, etc.

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

C++17:

- If constexpr()
- Inline variables
- Nested namespace definitions
- Class template argument deduction
- Hexadecimal literals
- etc

typename is permitted for template template parameter declarations (e.g.,
template<**template**<**typename**> **typename** X> **struct** ...)

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

- C++20:
 - Concepte: sintaxă mai "simplă" și erori mai "clare" la templates
 - Module
 - Spaceship operator `<=>` și operator `==()` = default
- C++23: `<stacktrace>`, `<expected>`

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

- `<iostream>` (fără `.h`)
- `using namespace std;`
- `cout, cin` (fără `&`)
- `//` comentarii pe o linie
- declarare variabile
- Tipul de date `bool`
- se definesc `true` și `false` (1 și 0);
- C99 nu îl definește ca `bool` ci ca `_Bool` (fără `true/false`)
- `<stdbool.h>` pentru compatibilitate

4. Curs 1

4.1 Completări aduse de limbajul C++ față 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.

// operator este o functie care are ca nume un simbol (sau mai multe simboluri)

```
int x,y,z;
```

```
cin >>x; // operator>>(cin,x) : intoarce fluxul (prin referinta ) cin din care s-a extras data x
```

```
cin>>y>>z; // operator>>(operator >>(cin,y), z)
```

```
cout<<x; // operator<<(cout,x) : intoarce fluxul cout (prin referinta) in care s-a inserat x
```

```
cout<<y<<z; // operator<<(operator<<(cout,y),z)
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    cout << "This is output.\n"; // this is a single line comment
    /* you can still use C style comments */
    // input a number using >>
    cout << "Enter a number: ";
    cin >> i;
    // now, output a number using <<
    cout << i << " squared is " << i*i << "\n";
    return 0;
}
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

```
#include <iostream>
using namespace std;
int main( )
{
    float f;
    char str[80];
    double d;
    cout << "Enter two floating point numbers: ";
    cin >> f >> d;
    cout << "Enter a string: ";
    cin >> str;
    cout << f << " " << d << " " << str;
    return 0;
}
```

citirea string-urilor se face până la primul caracter alb
se poate face afișare folosind toate caracterele speciale \n, \t, etc.

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Variabilele locale

```
/* Incorrect in C89. OK in C++. */  
int f()  
{  
    int i;  
    i = 10;  
    int j; /* aici problema de compilare in C */  
    j = i*2;  
    return j;  
}
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

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

Utilizarea mai multor funcții care au același nume

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

Simplicitate/corectitudine de cod

La compilare: transformat în nume unic prin name mangling

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Compiler de C

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void f(int x){printf("1");}
5  void f(int x, int y){printf("2");}
6
7  int main()
8  {
9      f(1);
10     f(2,3);
11     return 0;
12 }
```

Logs & others		
Code::Blocks Search results Cccc Build log Build messages		
File	L...	Message
=== Build: Debug in testC (compiler: GNU GCC Compiler) ===		
C:\Proie...	5	error: conflicting types for 'f'
C:\Proie...	4	note: previous definition of 'f' was here

Compiler de C++

```
main.cpp x
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void f(int x){printf("1");}
5  void f(int x, int y){printf("2");}
6
7  int main()
8  {
9      f(1);
10     f(2,3);
11     return 0;
12 }
```

```
12
Process returned 0 (0x0)
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

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

// abs is overloaded three ways
int abs(int i);
double abs(double d);
long abs(long l);

int main()
{
    cout << abs(-10) << "\n";
    cout << abs(-11.0) << "\n";
    cout << abs(-9L) << "\n";
    return 0;
}

int abs(int i)
{
    cout << "Using integer abs()\n";
    return i<0 ? -i : i;
}
```

```
double abs(double d)
{
    cout << "Using double abs()\n";
    return d<0.0 ? -d : d;
}

long abs(long l)
{
    cout << "Using long abs()\n";
    return l<0 ? -l : l;
}
```

```
Using integer abs()
10
Using double abs()
11
Using long abs()
9
```


4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Supraîncărcarea funcțiilor

tipuri diferite pentru parametrul i

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

```
int myfunc(int i) { return i; }
```

```
int main() {  
    cout << myfunc(10);  
    cout << myfunc(5.4);  
    return 0;  
}
```

numar diferit de parametri

```
int myfunc(int i) {return i;}
```

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

```
int main()  
{  
    cout << myfunc(10) ;  
    cout << myfunc(4, 5);  
    return 0;  
}
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Supraîncărcarea funcțiilor

Ambiguitati pentru polimorfism de functii

- erori la compilare – daca diferenta este doar in tipul de date intors sau sunt tipuri care par sa fie diferite
- majoritatea datorita conversiilor implicite

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);

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Supraîncărcarea funcțiilor

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

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Supraîncărcarea funcțiilor

Ambiguitati pentru polimorfism de functii

ambiguitate intre char si unsigned char

```
char myfunc(unsigned char ch){ return ch-1;}  
char myfunc(char ch){return ch+1;}
```

```
int main(){  
    cout << myfunc('c'); // this calls myfunc(char)  
    cout << myfunc(88) << " "; // ambiguous  
    return 0;  
}
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Funcții cu valori implicite

Într-o funcție se pot declara valori implicite pentru unul sau mai mulți 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 sfârșitul listei.

```
void f (int a, int b = 12){ cout<<a<<" - "<<b<<endl;}
```

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

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Funcții cu valori implicite

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

```
#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; }
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Funcții cu valori implicite

Atentie la ambiguitate!

```
int myfunc(int i) { return i; }
```

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

```
int main()
{
    cout << myfunc(4, 5) << " "; // unambiguous
    cout << myfunc(10); // ambiguous
    return 0;
}
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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;
}
```


4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Pointerii in C/C++

- O variabilă care ține 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 și `tip* nume_pointer;`

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Pointerii in C/C++

Operatori pe pointeri

- *, &, schimbare de tip
- *== “la adresa”
- &==“adresa lui”

```
int i=7, *j;
```

```
j=&i;
```

```
*j=9;
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Pointerii in C/C++

Compiler de C

```
main.c x
#include <stdio.h>
#include <stdlib.h>

int main()
{
    float x = 12.34, y;
    int *p;
    p = &x;
    y = *p;
    printf("%d\n", y);
    return 0;
}
```

"C:\Proiecte Codeblocks\testC" x

536870912

Process returned 0 (0x0)

Compiler de C++

```
main.cpp x
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      float x = 12.34, y;
8      int *p;
9      p = &x;
10     y = *p;
11     cout<<y;
12     return 0;
13 }
```

blocks x Search results x Cccc x Build log x Build message

L... Message

In function 'int main()':

9 error: cannot convert 'float*' to 'int*' in assignment

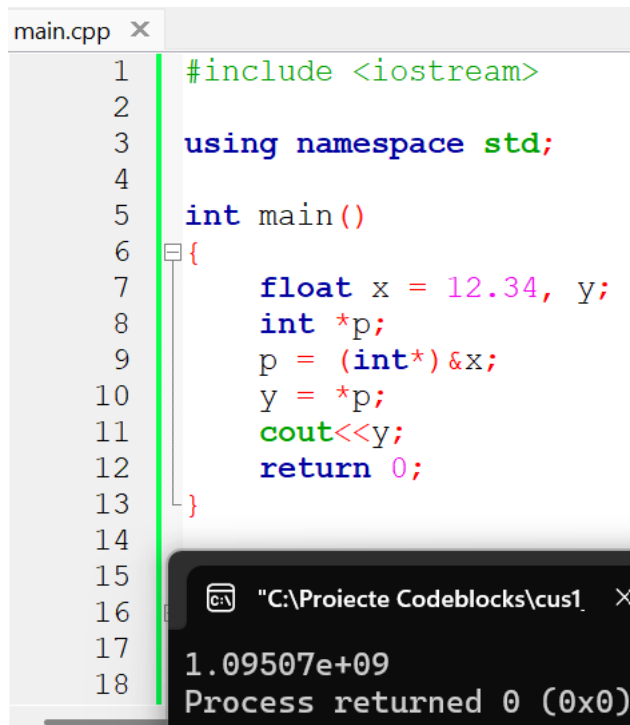
- atentie la corectitudine conversiilor!

în C++ conversiile trebuie făcute cu schimbarea de tip

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Pointerii in C/C++



```
main.cpp x
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      float x = 12.34, y;
8      int *p;
9      p = (int*) &x;
10     y = *p;
11     cout<<y;
12     return 0;
13 }
14
15
16
17
18
```

```
"C:\Proiecte Codeblocks\cus1" x
1.09507e+09
Process returned 0 (0x0)
```

- Schimbarea de tip nu e controlată de compilator
- În C++ conversiile trebuie făcute cu schimbarea de tip
- Atentie la corectitudinea conversiilor

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Pointerii in C/C++

Aritmetica pe pointeri

- `pointer++`; `pointer--`;
- `pointer+7`;
- `pointer-4`;
- `pointer1-pointer2`; întoarce un întreg
- comparații: `<`, `>`, `==`, etc.

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Pointerii in C/C++

pointeri și 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)

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Pointerii in C/C++

- în C++ tipurile pointerilor trebuie să fie la fel

```
int *p;
```

```
float *q;
```

```
p=q; //eroare
```

se poate face cu schimbarea de tip (type casting) dar ieșim din verificările automate făcute de C++

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Atentie cand lucram cu pointeri!

Ce problema apare in codul de mai jos?

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      for(int i = 0; i<5; i++)
8      {
9          int* p=new int();
10         (*p)++;
11         cout<<p<<" "<<*p<<endl;
12     }
13     return 0;
14 }
```

```
0xdb1780 1
0xdb17a0 1
0xdb1900 1
0xdb1920 1
0xdb1940 1
```

“Memory leak” – pointerul nu se dezaloca

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

O rezolvare in C++ prin utilizare de smart pointers (detalii mai tarziu)

```
1  #include <iostream>
2  #include <memory>
3
4  using namespace std;
5
6  int main()
7  {
8      for(int i = 0; i<5; i++)
9      {
10         shared_ptr<int> p(new int());
11         (*p)++;
12         cout<<p<<" "<<*p<<endl;
13     }
14     return 0;
15 }
16
```

```
0xec1780 1
0xec1780 1
0xec1780 1
0xec1780 1
0xec1780 1
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Alocare dinamica in C

Funcția malloc

Prototipul funcției:

void * malloc(int dimensiune);

unde:

- ***dimensiune*** = numărul de octeți ceruți a se alocă

1. Dacă există suficient spațiu liber în HEAP atunci un bloc de memorie continuu de dimensiunea specificată va fi marcat ca ocupat, iar funcția **malloc va returna un pointer ce conține adresa de început a aceluși bloc**. Dacă nu există suficient spațiu liber funcția **malloc** întoarce NULL.

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Alocare dinamica în C Funcția calloc

Prototipul funcției:

void * calloc(int numar, int dimensiune);

unde:

- *numar* reprezintă numărul de blocuri/elemente a se alocă
 - *dimensiune* = numărul de octeți ceruți pentru fiecare bloc
1. Dacă există suficient spațiu liber în HEAP atunci un bloc de memorie continuu de dimensiunea specificată va fi marcat ca ocupat, iar funcția **calloc va returna un pointer ce conține adresa de început a acelui bloc**. Dacă nu există suficient spațiu liber funcția *calloc* întoarce NULL.
 2. Diferența față de *malloc*: funcția *calloc* inițializează toate blocurile cu 0 (vector de frecvențe).

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Alocare dinamica in C

Funcția realloc

Prototipul funcției:

void * realloc(void *p, int dimensiune);

unde:

- ***p*** reprezinta un pointer (începutul unui bloc de memorie pe care vreau să îl redimensionez (de obicei avem nevoie de mai multă memorie))
 - ***dimensiune*** = numărul de octeți ceruți pentru alocare
1. Dacă există suficient spațiu liber în HEAP atunci un bloc de memorie continuu de dimensiunea specificată va fi marcat ca ocupat, iar funcția **realloc va returna un pointer ce conține adresa de început a aceluși bloc. Tot conținutul blocului de memorie inițial se copiază.** Dacă nu există suficient spațiu liber ***realloc*** întoarce NULL.

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Alocare dinamica in C

Funcția free

Prototipul funcției:

void free(void *p);

unde:

- *p* reprezinta un pointer (începutul unui bloc de memorie pe care vrem să-l eliberăm)
1. Funcția **free** eliberează zona de memoria alocată dinamic a cărei adresă de început este dată de *p*. Zona de memorie dezalocată este marcată ca fiind disponibilă pentru o nouă alocare.
 2. Un bloc de memorie nu trebuie eliberat de mai multe ori.

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Alocare dinamica in C++ (se poate utiliza si alocarea din C)

```
int *pi;  
pi=new int;
```

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

pi=**new** int(2); // alocă zona și initializează zona cu valoarea 2

pi=**new** int[2]; // alocă un vector de 2 elemente de tip întreg

delete [] pi; //elibereaza întreg vectorul

//-pentru new se foloseste delete

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

la eroare se “aruncă” excepția bad_alloc din <new>

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

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

int main()
{
    int *p;
    try {
        p = new int;
// allocate space for an int

    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }
    *p = 100;
    cout << "At " << p << " ";
    cout << "is the value " << *p << "\n";
    delete p;
    return 0;
}
```

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

int main()
{
    int *p;
    try {
        p = new int(100);
// initialize with 100

    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }

    cout << "At " << p << " ";
    cout << "is the value " << *p << "\n";
    delete p;
    return 0;
}
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Ne vom reaminti de alocarea dinamica, atunci cand vom discuta (mai tarziu) despre:

- vector in STL (Standard Template Library)
- Allocator
- Comanda Push_back

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C Const în 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

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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 încercăm să o schimbăm primim eroare de compilare

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Pointeri si Const in C/C++

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

`const int* u;`

- u este pointer către un int care este const
- `int const* v;` la fel ca mai sus

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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() {} ///:~
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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() {} ///:~
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Const si argumente de funcții, param de întoarcere

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

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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  
}
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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!
```

```
}
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Tipul referinta

O referință 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 referință trebuie inițializată in momentul declararii, devenind un alias (un alt nume) al obiectului cu care a fost inițializată.

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Tipul referinta

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

int main()
{
    int a = 20;
    int& ref = a;
    cout<<a<<" "<<ref<<endl; // 20 20
    ref++;
    cout<<a<<" "<<ref<<endl; // 21 21
    return 0;
}
```

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

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Tipul referinta

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

int main()
{
    int a = 20;
    int& ref = a;
    cout<<a<<" "<<ref<<endl; // 20 20
    int b = 50;
    ref = b;
    ref--;
    cout<<a<<" "<<ref<<endl; // 49 49
    return 0;
}
```

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

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

Tipul referinta

- o referinta trebuie să fie initializata când este definita, dacă nu este membra a unei clase, un parametru de functie sau o valoare returnata;
- referintele nule sunt interzise intr-un program C++ valid.
- nu se poate obtine adresa unei referinte.
- nu se pot crea tablouri de referinte.
- nu se poate face referinta catre un camp de biti.

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

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

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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++

```
#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;
}
```


4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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 \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.

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul C

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;
}
```

4. Curs 1

4.1 Completări aduse de limbajul C++ față de limbajul 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.

4. Curs 1

4.2 Principiile programarii orientate pe obiecte

- ce este programarea
- definirea programatorului:
 - rezolva problema
- definirea informaticianului:
 - rezolva problema **bine**

4. Curs 1

4.2 Principiile programarii orientate pe obiecte

Rezolvarea “mai bine” a unei probleme

- “bine” depinde de caz
 - drivere: cat mai repede (asamblare)
 - jocuri de celulare: memorie mica
 - rachete, medicale: erori duc la pierderi de vieti
- programarea OO: cod mai corect
 - Microsoft: nu conteaza erorile minore, conteaza data lansarii
 - Utilizare principii S.O.L.I.D. (design patterns) – detalii mai tarziu!

4. Curs 1

4.2 Principiile programarii orientate pe obiecte

Principalele concepte utilizate in POO sunt:

Obiecte

Clase

Mostenire

Ascunderea informatiei

Polimorfism

***Sabloane* – nu sunt utilizate strict POO (mai general, se refera la Programarea generica)**

4. Curs 1

4.2 Principiile programarii orientate pe obiecte

O **clasa** definește atribute și metode.

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

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

4. Curs 1

4.2 Principiile programarii orientate pe obiecte

Un **obiect** este o instanta a unei clase care are o anumita stare (reprezentata prin valoare) si are un comportament (reprezentat prin functii) la un anumit moment de timp.

- au stare si actiuni (metode/functii)
- au interfata (actiuni) si o parte ascunsa (starea)
- Sunt grupate in clase, obiecte cu aceleasi proprietati

Un **program orientat obiect** este o colectie de obiecte care interactioneaza unul cu celalalt prin mesaje (aplicand o metoda).

4. Curs 1

4.2 Principiile programarii orientate pe obiecte

Principalele concepte (caracteristici) ale POO sunt:

Incapsularea – contopirea datelor cu codul (metode de prelucrare si acces la date) în clase, ducând la o localizare mai bună a erorilor și la modularizarea problemei de rezolvat;

Moștenirea - posibilitatea de a extinde o clasa prin adaugarea de noi functionalitati;

- multe obiecte au proprietati similare
- reutilizare de cod

4. Curs 1

4.2 Principiile programarii orientate pe obiecte

Principalele concepte (caracteristici) ale POO sunt:

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

4. Curs 1

4.2 Principiile programarii orientate pe obiecte

Principalele concepte (caracteristici) ale POO sunt:

Polimorfismul (la executie – *discutii mai ample mai tarziu*) – într-o ierarhie de clase obtinuta prin mostenire, o metodă poate avea implementari diferite la nivele diferite in acea ierarhie;

4. Curs 1

4.2 Principiile programarii orientate pe obiecte

Alt concept important in POO:

Sabloane

- cod mai sigur/reutilizare de cod
- putem implementa lista inlantuita de
 - intregi
 - caractere
 - float
 - obiecte

Perspective

1. Se vor discuta directiile principale ale cursului, feedback-ul studentilor fiind hotarator in acest aspect

- intelegerea notiunilor
- intrebari si sugestii

2. Cursul 2:

- Introducere in OOP. Clase. Obiecte