

Programare Orientată Obiect

ZURINI MĂDĂLINA

OBJECTIVE

Obiectivul principal:

- Prezentarea noțiunilor aferente paradigmei Orientate Obiect în limbajul C++

Obiective secundare:

- Modelarea obiectată obiect
- Editarea și compilarea codului sursă C++
- Depanarea și rezolvarea erorilor de compilare și de execuție

EVALUARE

60% EXAMEN FINAL

- examen oral – subiect extras și rezolvat la calculator

40% ACTIVITATE SEMINAR

- 20% test la seminar
- 10% participare activă
- 10% test grilă (S14)

Condiții intrare în examen în sesiunea din iarnă: Obținerea a minim **1.5 pct** din cele 4 aferente activității de seminar

Condiții promovare disciplină: Minim 3 pct din 6 la examenul final și minim 4.5 pct din 10 per total la nivel de disciplină

BIBLIOGRAFIE

- Ion Smeureanu – “Programarea în limbajul C/C++”, Editura CISON, 2001
- Ion Smeureanu, Marian Dardala – “Programarea orientată obiect în limbajul C++”, Editura CISON, 2002
- acs.ase.ro/cpp

STRUCTURĂ CURS

- Recapitulare (Structuri, pointeri, masive, transmiterea parametrilor, tipuri de erori)
- Clase (Definire, attribute, metode, constructori, operatori)
- Fișiere (Lucrul cu stream-uri, fișiere standard, text și binare)
- Derivare (Ierahii de clase, polimorfism, funcții virtuale)
- STL (Clase template, Standard Template Library)

DE CE ORIENTAT OBIECT?

I. ABSTRACTIZARE

- Modelarea mai ușoară a elementelor din lumea reală

II. INCAPSULARE

- Ascunderea complexității/informațiilor/comportamentelor de alte entități din exterior

III. DERIVARE

- Crearea de entități noi pe baza unora deja existente

IV. POLIMORFISM

- Comportamente multiple în funcție de context

DE CE C++?

- Foarte rapid
- Top 5 limbaje la nivel mondial de peste 30 de ani (<https://www.tiobe.com/tiobe-index/>)
- Încorporează toate conceptele POO într-un mod pur
- Are sintaxa limbajului C

MEDIU DE DEZVOLTARE

- Microsoft Visual Studio 2022 Community la curs și laborator
- **Atenție!** Chiar dacă paradigma este aceeași, diferite compilatoare de C++ pot returna rezultate diferite pentru același cod sursă





RECAPITULARE C01+C02

TIPURI DE ERORI

ERORI DE COMPILARE

Error List

Entire Solution ✖ 2 Errors ⚠ 1 Warning i 0 of 1 Message 7 Build + IntelliSense

	Code	Description	Project	File	Line	Details
	E0020	identifier "a" is undefined	Project2	FileName2.cpp	8	
	C6001	Using uninitialized memory 'x'.	Project2	FileName2.cpp	6	
	C2065	'a': undeclared identifier	Project2	FileName2.cpp	8	

ERORI DE EXECUȚIE

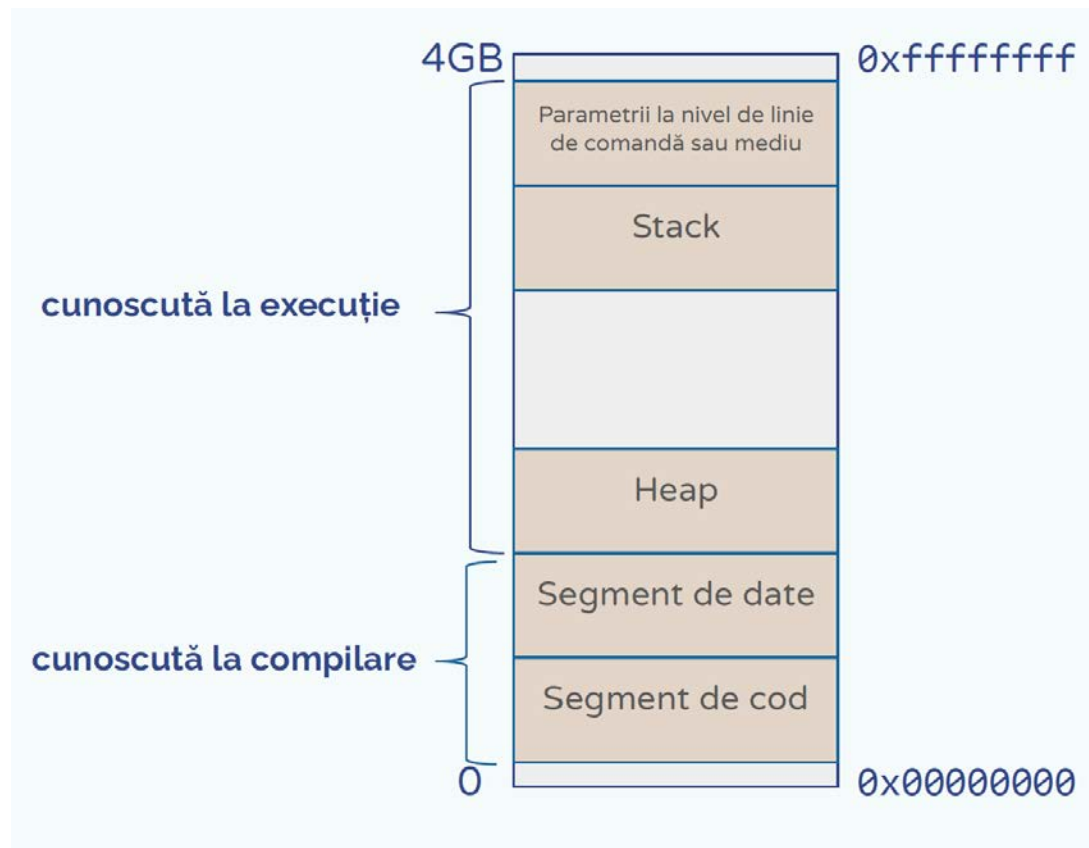
```
D:\Proiecte Visual\Project2\Debug\Project2.exe (process 25624) exited with code -1073741676.  
Press any key to close this window . . .|
```

NEXT TIME

Continuare recapitulare

- a. Pointeri
- b. Alocare statică și dinamică
- c. Tipuri de memorie
- d. Variabile constante
- e. Masive uni și bidimensionale
- f. Șiruri de caractere
- g. Transmiterea parametrilor
- h. Directive de preprocesare

MEMORIA



POINTERI

- Variabile responsabile cu salvarea unor adrese de memorie
- sunt salvați în stack și pointează către zone din heap (acesta este cel mai întâlnit scenario)
- Ocupă 4 bytes indiferent de tipul de pointer (ce se regăsește la adresa de
- memorie către care pointează) pentru un procesor/compiler pe 32 de biți

DEFINIRE

- la definire de obicei valoarea este una arbitrara
- nu este recomandat ca un pointer sa ramana neinitializat

```
tip_data* nume_pointer;  
tip_data* nume_pointer = nullptr;
```

INITIALIZARE

- pointerul poate salva doar adrese unde se gaseste tipul de data specificat
- operatorul & extrage adresa de memorie a unei variabile

tip_data nume_variabila = valoare;

tip_data* nume_pointer = &nume_variabila;

UTILIZARE

//va afisa o adresa de memorie

```
cout << nume_pointer;
```

//va afisa continutul de la acea adresa de memorie

```
cout << *nume_pointer;
```


INCREMENTARE/DECREMENTARE

- Incrementarea unui pointer de tip T^* va duce la mărirea adresei cu `sizeof(T)` sau, cu alte cuvinte, deplasarea în memorie înainte către adresa următoarei variabile de tip T
- Decrementarea unui pointer de tip T^* va duce la scăderea adresei cu `sizeof(T)` sau, cu alte cuvinte, deplasarea în memorie înapoi către adresa precedentei variabile de tip T

IDENTIFICATORUL CONST LA POINTERI

1. pointeri constanți: odată salvată o adresă, nu mai poate fi modificată
2. pointeri la o zonă de memorie constantă: valoarea de la adresa către care pointează nu poate fi modificată prin intermediul lor
3. pointeri constanți la o zonă de memorie constantă: combinație între cele două situații de mai sus

IDENTIFICATORUL CONST LA POINTERI

```
int a;
```

```
const int* px1;
```

```
int* const px2 = &a;
```

```
const int* const px3 = &a;
```

```
int const* px4;
```

MASIVE UNIDIMENSIONALE

- Tipuri de date ce folosesc o zonă de memorie contiguă pentru a salva mai multe valori de același tip
- Pot fi alocați static sau dinamic
- Cei alocați static sunt salvați în stack și trebuie să aibă un număr de elemente cunoscut în momentul compilării
- Cei alocați dinamic se alocă și se dezalocă în heap și pot avea un număr de elemente cunoscut la momentul execuției

MASIVE BIDIMENSIONALE

- Masive bidimensionale ce permit accesul elementelor pe linii și coloane
- Memoria fiind liniară (unidimensională) pentru a putea fi salvate sunt liniarizate
- Cele alocate static sunt salvate în memorie ca vectori consecutivi ce conțin elementele de pe fiecare linie
- Cele alocate dinamic sunt salvate ca vectori de vectori (un vector ce conține adresele de memorie ale fiecărui vector corespunzător fiecărei linii)

OPERATORUL [] SI POINTERI

- $\text{vector}[i] \Leftrightarrow *(\text{vector} + i)$
- $\text{matrice}[i][j] \Leftrightarrow *(*(\text{matrice} + i) + j)$

VECTORI DE CARACTERE

- În limbajul C nu există un tip de dată special definit pentru șiruri de caractere
- Pentru a salva totuși astfel de șiruri se folosește o convenție: se utilizează un vector de caractere (vector de char) ce are drept ultim element `'\0'` (cod ASCII 0) pentru a ști când vectorul de caractere se sfârșește
- Vectorii de caractere pot fi prelucrați ca orice alt vector sau, profitând de faptul că putem ști când ajungem la ultimul element, prin utilizarea de funcții specifice

SIRURI DE CARACTERE - STRING

- În limbajul C++ avem tip de dată special definit pentru șiruri de caractere - string
- string este o clasă în C++ așadar prelucrările se fac în mod direct prin operatori sau metode

CLASE C03

ENUMERARI

- Folosite atunci cand o data ia valori doar intr-un interval finit dat
- La compilare sunt transformate in constante
- Ajuta la o intelegere mai usoara a codului

```
enum formaStudiu {  
    ZI, ID, IDD };  
formaStudiu f = formaStudiu::ZI;
```

UNIUNI

- Date ce poate salva o singura valoare la un moment dat dintr-un set de valori disponibile
- Ocupa zona de memorie egal cu maximul dintre valorile posibile

```
union id {  
    int cui;  
    long cnp;  
};
```

```
id idClient;  
idClient.cnp = 1674546;  
cout << idClient.cnp << " " << idClient.cui;  
idClient.cui = 34567;  
cout << endl << idClient.cnp << " " << idClient.cui;
```

STRUCTURI

- Permite crearea unui tip de data complex ce grupeaza mai multe caracteristici
- Inceputul conceptului de incapsulare
- In C, permite doar existenta atributelor

```
struct Student {  
    string nume;  
    int grupa;  
    char serie;  
};
```

```
Student s;  
s.nume = "Gigel";  
s.grupa = 1056;  
s.serie = 's';
```

CLASE

- Asemănătoare structurilor, încapsulează caracteristici și comportamente
- Baza POO

```
class Masina {  
    //atribute si metode  
};
```

ATTRIBUTE

- modeleaza caracteristicile, starea unui obiect

```
class Masina {  
    int anProductie;  
    string marca;  
    string culoare;  
    //....  
};
```

METODE

- modeleaza comportamentul unui obiect

```
class Masina {  
    int anProductie;  
    string marca;  
    string culoare;  
    int kmParcursi;  
    //....  
  
    void adaugaKm(int nrKm) {  
        kmParcursi += nrKm;  
    }  
};
```

OBIECTE

- instante ale claselor (variabile de tipul clasei)

```
Masina m;  
m.culoare = "Galbena";  
m.kmParcursi = 1500;  
m.adaugaKm(100);  
cout << endl << m.kmParcursi;
```


DOMENII DE VIZIBILITATE

- PRIVATE (tot ce este definit in aceasta zona poate fi accesat doar din interiorul clasei, deci nu poate fi accesat din exteriorul ei)
- PROTECTED (poate fi accesat din interiorul clasei precum si din clasele derivate din ea)
- PUBLIC (poate fi accesat de oriunde)

UTILIZARE DOMENII DE VIZIBILITATE

```
class Masina {  
private:  
    int anProductie;  
    string marca;  
  
public:  
    string culoare;  
  
    void adaugaKm(int nrKm) {  
        kmParcursi += nrKm;  
    }  
  
protected:  
    int kmParcursi;  
};
```

CLASE VS STRUCTURI

CLASA	STRUCTURA
Membrii clasei sunt implicit privati	Membrii structurii sunt implicit publici
Se declara folosind cuvantul cheie class	Se declara folosind cuvantul cheie struct
Se foloseste de regula pentru abstractizare si mostenire	Se foloseste pentru a grupa diferite tipuri de date