

# Programare II

## Limbajul C/C++

CURS 6



# Despre ce vom discuta?

## Programare procedurală

- ❑ Limbajul C
  - ❑ Tipuri de date și instrucțiuni
  - ❑ Funcții și macrodefiniții
  - ❑ Tablouri și pointeri
  - ❑ Structuri și uniuni

## Programare orientată obiect

- ❑ Limbajul C++
  - ❑ Concepte noi în C++
  - ❑ Clase, obiecte, constructori, destructori
  - ❑ Redefinirea operatorilor
  - ❑ Moștenire și polimorfism
  - ❑ Funcții/clase șablon (template)
  - ❑ Standard Template Library
  - ❑ Concepte de OO

# Curs curent

- ❑ Programare orientată obiect. Abstractizarea datelor
- ❑ Diferențe / noutăți C – C++

# Paradigme de programare

- ❑ Programare nestructurată

- ❑ Programe simple / mici ca dimensiune care conțin doar o **singură metodă**

- ❑ Programare procedurală

- ❑ **Procedura** stochează algoritmul pe care dorim să îl (re)folosim

- ❑ Programarea modulară

- ❑ Partiționarea programelor astfel încât datele să fie ascunse în module (**data hiding principiele**)

- ❑ Abstractizarea datelor

- ❑ Realizarea de **tipuri de date definite de utilizator** care se comportă ca și tipurile implicite

- ❑ Programarea orientată obiect

- ❑ **Obiecte** care **interacționează**, fiecare gestionând **starea proprie**

- ❑ Programarea generică

- ❑ Algoritmii **independenți** de **detaliile de reprezentare**

# Programarea orientată pe obiect

❑ Un limbaj sau o tehnică este orientat obiect dacă și numai dacă îndeplinește următoarele condiții [Stroustrup, 1995]

❑ Abstractizare

❑ clase și obiecte

❑ Moștenire

❑ posibilitatea de a construi noi abstractizări peste cele existente

❑ Polimorfism la execuție

❑ mecanism de legare dinamică (runtime binding)

# Ce este C++?

## □ Definiție 1

- C + + este un limbaj de programare cu **scop general**, cu o înclinație spre sisteme de programare care suportă **eficient** calcule de nivel scăzut, **abstractizare a datelor**, **programarea orientată pe obiect** și programarea **generică**. [Stroustrup, 1999]

## □ Definiție 2

- C + + este un limbaj **static** (statically-typed) cu **scop general** bazându-se pe **clase** și **funcții virtuale** pentru a sprijini programarea **orientată pe obiect**, **template-uri** pentru a sprijini programarea **generică**, precum și furnizarea de facilități de nivel scăzut pentru a sprijini sistemele de programare detaliate. [Stroustrup, 1996]

# Idei de proiectare C++

❑ C++ a fost conceput pentru a accepta o gamă largă de stiluri bune și folosite. Indiferent dacă acestea au fost orientate-obiect [Stroustrup, 1995]:

1. **Abstractizarea** – capacitatea de a reprezenta concepte direct într-un program și ascunde detaliile din spate (interfețe bine definite)  
❑ este cheia pentru orice sistem flexibil și ușor de înțeles de dimensiuni considerabile.
2. **Încapsulare** - capacitatea de a oferi garanții că o abstractizare este utilizată numai în conformitate cu specificațiile sale  
❑ este esențial să se apere împotriva coruperi abstractizări.
3. **Polimorfism** - capacitatea de a oferi aceeași interfață pentru obiecte cu implementări diferite  
❑ este utilă pentru a simplifica codul folosind abstractizări.
4. **Moștenirea** - abilitatea de a compune abstractizări noi pornind de la unele deja existente  
❑ este unul dintre cele mai puternice moduri de construcție de abstractizări utile.
5. **Genericele** - capacitatea de parametrizare a tipurilor și funcțiilor, prin tipuri și valori  
❑ este esențială pentru crearea de tipuri sigure și este un instrument puternic pentru a scrie algoritmi generali.
6. **Coexistența cu alte limbaje și sisteme** - utilă pentru funcționarea în medii de execuție reale.
7. **Compactitatea și viteza la execuție** - utilă pentru programare pe sisteme clasice.
8. **Limbaj static** – o familie de limbaje din care face parte și C++ care asigură eficiență de spațiu și la rulare a programelor

# Istoric C++

- ❑ 1979 - C with Classes: Bjarne Stroustrup (AT&T Bell Labs) transpune conceptele (precum clase, moștenire) din Simula67 în C
- ❑ 1982 - From C with Classes to C++: prima variantă de C++ și publicarea cărții care definește limbajul C++ în Octombrie 1985
- ❑ 1985 - Versiunea 2.0: Evoluția primei versiunii (publicarea cărți The C++ Programming Language - Bjarne Stroustrup)
- ❑ 1988 – Standardele actuale: ISO și ANSI
- ❑ 1994 - Standard Template Library
- ❑ 1998 - International C++ standard
- ❑ 2011 - un nou standard pentru C++11
- ❑ 2013 The C++ Programming Language, 4th edition
- ❑ 2014 C++14 mici modificări aduse standardului
- ❑ 2017 C++17 modificări bazate pe standardele de C++11
- ❑ 2020 C++20 modificări bazate pe standardele de C++17



# Cuprins

❑ Programare orientată obiect. Abstractizarea datelor

❑ Diferențe / noutăți C – C++

# Diferențe/noutăți C/C++

❑ Tipul de date boolean

❑ Pointeri `void`

❑ Operații de scriere/citire

❑ Spațiu de nume

❑ Operatorul de rezoluție

❑ Alocarea dinamică a memoriei

❑ Variabile referință

❑ Supradefinirea numelui de funcții

❑ Funcții cu parametri implicați

❑ Tratarea excepțiilor

# Tipul de date boolean

- ❑ C++ introduce un nou tip de date implicit pentru tipul boolean

- ❑ Tip: `bool`

- ❑ Valori: `true`, `false`

- ❑ Exemplu: găsirea unui număr  $x$  într-un tablou  $t[1..n]$

```
bool gasit;  
  
int i=0;  
  
gasit = false;  
  
while(! gasit && i < n){  
    if (t[i] == x) gasit = true;  
  
    i++;  
}
```

# Pointeri void

❑ Pentru tipul (void\*) C++ admite doar conversia implicită în sensul:

❑ Tip pointer -> void\*

```
int main() {  
    void * vp; int *ip;  
    vp = ip; //corect in C si C++  
    ip = vp; //corect in C, incorrect in C++  
    ip = (int*)vp; //corect in C si C++ return 0  
}
```

# Constanta nullptr

❑ C++11 s-a introdus cuvântul cheie `nullptr`, care indica spre un pointer NULL, nu constanta 0.

❑ Exemplu:

❑ `double *d = nullptr;`

❑ Ambele variante de inițializare cu valoare nulă sunt acceptate, dar standardul de C++11 recomandă folosirea noii constante.

# Operații de citire/scriere

## ❑ Operatorii

- ❑ Scriere <<

- ❑ Citire >>

## ❑ Streamuri

- ❑ Citire de la tastatură cin

- ❑ Scriere pe ecran cout

## ❑ Biblioteca

- ❑ iostream

## ❑ Exemplu

```
#include <iostream>
using namespace std;
int main() {
    int a; double d;

    cout << "a=" ; cin >> a;
    cout << "d="; cin >> d;

    cout << "a=" << a << endl;

    char s[10];
    cin >> s; cin.ignore();
}
```

# Spații de nume

❑ Folosit pentru a rezolva conflictele de nume care pot apărea între diferite biblioteci

❑ Același nume de funcție în două biblioteci

```
#include <iostream>
using namespace std;
// primul spatiu de nume
namespace first_space {
    void func() { cout << "Primul spatiu de nume" << endl; }
}
// al doilea spatiu de nume
namespace second_space {
    void func() { cout << "Al doilea spatiu de nume" << endl; }
}
int main () {
    // Apel functie din primul spatiu de nume.
    first_space::func();
    // Apel functie din al doilea spatiu de nume.
    second_space::func();
    return 0;
}
```

Includerea globală a spațiului de nume, funcțiile din spațiu de nume se pot folosi fără a fi prefixate de spațiul de nume.

Definirea unui nou spațiu de nume

Utilizarea funcției definită în spațiul de nume

# Operatorul de rezoluție

## ❑ Operatorul rezoluție ::

- ❑ permite accesul la un identificador cu domeniul fișier, dintr-un bloc în care acesta nu e vizibil din cauza unei operații
- ❑ Permite accesul la informațiile definite într-un spațiu de nume

## ❑ Exemplu

```
char str[20]="Sir global";
void fct() {
    char *str;
    str = "Sir local";
    std::cout << ::str << std::endl;
    std::cout << str << std::endl;
}

void main() {
    fct();
}
```

Accesul la un identificador cu domeniul fișier

Accesul la conținutul spațiului de nume



# Alocarea dinamică a memoriei

## ❑ Limbajul C

- ❑ implementat în biblioteci auxiliare
- ❑ pus la dispoziția programatorilor prin intermediul unor **funcții**

## ❑ Limbajul C++

- ❑ introduce doi noi **operatori** pentru managementul memorie

- ❑ new – alocare

- ❑ Sintaxă:

- ```
tip _ptr =new tip
tip _ptr =new tip(val_initalizare)
tip _ptr =new tip[n]
```

- ❑ delete – dealocare

- ❑ Sintaxă:

- ```
delete _ptr
```

# Alocarea dinamică a memoriei

## ❑ Exemple

### ❑ Alocarea/dealocarea unui pointer întreg

```
int *p = new int;  
*p=3;  
delete p;
```

### ❑ Alocarea/dealocarea și inițializarea unui pointer întreg

```
int *p = new int(3);  
delete p;
```

### ❑ Alocarea/dealocarea unui șir de 10 elemente întregi

```
int *p = new int[10];  
delete [] p;
```

Dacă nu se pun parantezele drepte se șterge doar legătura dintre variabila *p* și zona spre care indică, NU se va dealoca spațiul alocat pentru cei 10 întregi

# Variabile referință

- ❑ C++ permite declararea unor identificatori ca referințe de obiecte de un anumit tip.
- ❑ Referințele pot fi văzute ca **aliasuri** ale **numelui** unei **variabile**
- ❑ Variabile referință se prefixează cu **&**
- ❑ Variabile referință **trebuie inițializate**

## ❑ Exemplu

```
int main() {  
    int n=6;  
    int & ir = n;  
    cout<< "n=" << n << " ir=" <<ir << endl;  
    ir = 100; // echivalent cu n =100;  
    cout<< "n=" << n << " ir=" <<ir << endl;  
}
```

# Variabile referință

## ❑ Parametri ai funcției

- ❑ Nu se realizează o copie a variabilei, echivalente cu folosirea parametrilor de tip pointer

## ❑ Exemplu

```
void swap1(int &a, int&b);  
void main(){  
    int n=4, m=6;  
    cout << "n=" << n << "m=" << m << endl;  
    swap1(n, m);  
    cout << "n=" << n << "m=" << m << endl;  
}  
void swap1(int &a, int&b){  
    int temp = a;  
    a = b;  
    b = temp;  
}
```

# Supraîncărcarea numelui de funcție

- ❑ C nu este permisă existența a două funcții care au același nume
- ❑ C++ acest lucru este posibil cu următoarea precizare
  - ❑ funcțiile **nu pot** să difere doar prin valoarea returnată, este obligatoriu ca lista de parametri să fie diferită (tipurile sau numărul variabilelor din lista de parametri)

## ❑ Exemplu

```
int add(int a, int b){ return a+b; }
char * add(char *s1, char *s2){ ... }

void main(){
    int a = 10, b =10;
    cout << "Adunare de numere " << add(a,b) << endl;
    char *s1="Primul ", *s2="laborator!";
    cout << "Adunare de caractere " << add(s1,s2) << endl;
}
```

# Funcții cu parametri implicați

❑ Permite ca la apelarea funcțiilor să se omită parametrii cu valori implicite, acești parametri se vor completa cu valoare implicită.

## ❑ Restricții

- ❑ Valorile implicite se specifică o singură dată în prototipul sau definiția funcției.
- ❑ Argumentele cu valori implicite trebuie plasate la sfârșitul listei de argumente a funcției.

## ❑ Exemplu

```
//double fct1(char ='A', int, float = 12); //eroare de compilare  
void fct3(int, long=99, float=10.6);
```

```
void main(){  
  
    float f = 1.4;  
    fct3(0); //apel corect  
    fct3(1,f); //apel incorect  
    fct3(2, 2.6); //apel incorect  
}  
void fct3(int i, long l, float f){  
  
    cout<<"i="<<i<<"l="<<l<<"f="<<f<<endl;  
}
```

# Tratarea excepțiilor

## ❑ Definiție

- ❑ O excepție este o **eroare** apare în momentul **execuției** unui program (run-time).

## ❑ Exemple

- ❑ Memorie insuficientă
- ❑ Un fișier nu poate fi deschis
- ❑ Un obiect invalid pentru o operație
- ❑ etc

# Tratarea excepțiilor

❑ Ce facem când apare o excepție?

❑ Terminarea programului → **soluție inacceptabilă**

❑ Returnarea unei valori reprezentative pentru eroare → **Care ar fi un cod de eroare reprezentativ? Trebuie ca funcția apelantă să verifice codul întors de funcția apelată**

❑ Întoarce o valoare legală și lasă programul într-o stare ilegală (errno) → **apelantul trebuie să testeze starea variabilei astfel încât să nu realizeze un apel invalid**

❑ Apelarea unei funcții care va trebui apelată în caz de eroare → **nu există control asupra codului apelantului**

❑ Combinarea codului uzual cu codul de tratare a excepțiilor → programe greu de citit și întreținut



# Tratarea excepțiilor

❑ Ce se întâmplă dacă **nu folosim un mecanism** de tratare a excepțiilor?

❑ Când apare o excepție, **controlul este predat sistemului de operare**, de obicei

❑ Un mesaj de eroare este afișat

❑ Programul se termină

❑ Ce se întâmplă dacă **folosim un mecanism** de tratare a excepțiilor?

❑ Programele pot **înălța excepțiile**

❑ Există posibilitatea **tratării** și să **continuării** execuției programului

# Tratarea excepțiilor

## ❑ Abordare

- ❑ Codul care detectează o eroare

- ❑ O **propagă** mai departe folosind instrucțiunea `throw`

- ❑ O **tratează** folosind blocul `try ... catch`

# Tratarea excepțiilor

## Propagare

```
void operation() {  
    if(daca_un_caz_de_erore_exista) {  
        // aruncă un obiect de  
        // tip excepție  
        throw TipExcepție();  
    }  
    ...  
}
```

## Tratare

```
void f() {  
    try{  
        // ... cod care ar putea  
        // arunca o excepție ...  
    } catch (TipExcepție1 e1) {  
        // tratează o excepție de tip1  
    } catch (TipExcepție2 e2) {  
        ...  
    }  
}
```

# Tratarea excepțiilor

## ❑ Exemplu

### ❑ Operațiile de adăugare scoatere de elemente dintr-o Stivă

```
void adauga(int x){  
    /**functia adauga elementul x in stiva*/  
    if (index ==dim) throw "Stiva Plina!"; //tip implicit  
    tab[index] = x;  
    index++;  
}
```

```
int scoate(){  
    /**functia scoate un element din stiva*/  
    if (index ==0) throw StivaGoala(); // tip definit de utilizator  
    return tab[index--];  
}
```

# Tratarea excepțiilor

## ❑ Exemplu

### ❑ Tratarea excepțiilor

```
try{
    s.scoate();
    s.adauga(4);
    s.adauga(5);
} catch (StivaGoala a) {
    cout << "Stiva este goala!" <<
        "Exceptia prinsa: " << a << endl;
}
```

# Tratarea excepțiilor

## ❑ Observații

- ❑ Un bloc `try` poate avea atașat **unul sau mai multe blocuri** `catch`

- ❑ Un bloc `catch` poate avea următoarele forme

- ❑ `catch(tipExcepție)`

- ❑ Prinde orice excepție de tipul specificat (`tipExcepție`) fără a permite regăsirea datelor stocate în excepție

- ❑ `catch(tipExcepție variabila)`

- ❑ Prinde orice excepție de tipul specificat (`tipExcepție`), permite regăsirea datelor stocate în excepție

- ❑ `catch(...)`

- ❑ Prinde orice excepție fără a permite identificarea tipului excepției sau datelor stocate în ea

- ❑ În interiorul unei funcții pot exista mai multe blocuri `try-catch`, blocurile `try-catch` pot fi imbricate

# Tratarea excepțiilor

## ❑ Observații

```
void fct(int a) {  
    if (a<0) throw "Eroare valoare parametru";  
    a=a+5;  
}
```

- ❑ În acest caz dacă se intră pe ramura de *then* a instrucțiuni *if* execuția instrucțiuni *throw* este **echivalentă** cu apelul instrucțiuni *return* adică întrerupe execuția funcției și codul de **după** instrucțiunea *throw* **nu se mai execută**

# Tratarea excepțiilor

## ❑ Obsevații

```
void fct(int a) {  
    if (a<0) throw "Eroare valoare parametru";  
    a=a+5;  
}  
  
int main() {  
    cout << "Inainte apel\n";  
    fct(-3);  
    cout << "Dupa apel\n" ;  
}
```

❑ În acest caz se va afișa pe ecran doar mesajul "Inainte apel" deoarece apelul funcției *fct()* aruncă o eroare care duce **la întreruperea execuției metodei *main()*** .



# Tratarea excepțiilor

## ❑ Observații

```
void fct(int a){
    if (a<0) throw "Eroare valoare parametru";
    a=a+5;
}
int main(){
    cout << "Inainte apel\n";
    try{
        fct(-3);
        cout << "Dupa apel\n"
    }catch (const char *ex) {
        cout <<"Exeptie: " <<ex << endl;
    }
    cout << "Dupa try\n";
}
```

❑ În acest caz se vor afișa pe ecran doar mesajele: "Inainte apel", "Exeptie : Eroare valoare parametru" si "Dupa try";deoarece apelul funcției *fct()* aruncă o eroare si codul care urmează acestei linii pana la întâlnirea unui bloc try care tratează acesta eroare nu se mai executa

# Tratarea excepțiilor. Specificarea excepțiilor

- ❑ Lista de excepții aruncate de o funcție se poate specifica ca parte a declarării funcțiilor

- ❑ Sintaxă

- ❑ `tipDeReturn nume_functie (lista_argumente) throw (listă_de_excepții)`

- ❑ Exemplu

- ❑ `void adunare(Matrice &m1, Matrice &m2) throw (ExceptieMatematica, AlocareIncorecta)`

- ❑ Dacă lista de excepții lipsește o funcție poate arunca orice excepție

- ❑ `void f() throw();` - nu aruncă nici o excepție

# Tratarea excepțiilor. Specificarea excepțiilor

## ❑ Observații

- ❑ Specificarea excepțiilor trebuie inclusă atât în declarația cât și în definiția funcției
- ❑ O funcție virtuală poate fi supradefinită doar de o funcție care este cel puțin la fel de restrictivă ca funcție inițială
- ❑ Dacă o altă excepție este aruncată în afara celor specificate în clauza `throw` → apel `std::unexped()` , care apelează `std::terminate()` care apelează funcție `abrupt()`
- ❑ Se poate supradefini comportamentelor funcțiilor `std::unexped()` , `std::terminate()` prin suprascrierea handlerelor `set_unexpected` respectiv `set_terminate`

# Tratarea excepțiilor. Excepții neprinse

❑ Dacă o excepție este aruncată dar nu este prinsă, funcția `std::terminate()` va fi apelată

❑ Pentru a prinde toate excepțiile netratate se poate utiliza următorul bloc

```
int main() {  
    try {  
        // cod care poate arunca excepții  
    } catch(...) {  
        // tratează orice excepție neprinsă până acum  
    }  
}
```

❑ Aruncarea unei excepții este mai puțin costisitoare decât apelul unei funcții

# Tratarea excepțiilor. Re-transmiterea excepțiilor

- ❑ După ce s-a realizat o corecție a datelor într-o clauză catch, excepția poate fi aruncată mai departe pentru a fi procesată în altă parte

```
void func() {  
    try{  
        //cod care aruncă o excepție bad_alloc  
    } catch (bad_alloc e) {  
        // cod de abordare a excepției  
        throw;  
    }  
}
```

# Tratarea excepțiilor. Imbricarea blocurilor try-catch

❑ Excepțiile sunt întotdeauna tratate de cel mai apropiat handler

```
try {  
  
    try {  
        throw 5;  
    } catch (int x) {  
        cout << x << endl;  
        // excepția va fi prinsă aici  
    }  
  
} catch (int x) {  
    cout << x-5 << endl;  
}
```



ÎNTREBĂRI