



# Programare orientată pe obiecte

- suport de curs -

**Andrei Păun  
Anca Dobrovăț**

**An universitar 2021 – 2022  
Semestrul II  
Seriile 13, 14 si 15**

**Curs 1**



## Generalități despre curs

1. Curs – seria 13: luni (10 -12), seria 14: marti (8 – 10), seria 15: vineri (12 - 14)
  2. Laborator – pe semigrupe, în fiecare săptămână
  3. Seminar - o dată la 2 săptămâni
  4. Prezenta la curs/seminar: nu e obligatorie!
- Laborator – OBLIGATORIU**



# Să ne cunoaștem

Cine predă?

Curs: Anca Dobrovăț (13, 15) și Andrei Păun (14)

- [anca.dobrovat@fmi.unibuc.ro](mailto:anca.dobrovat@fmi.unibuc.ro)
- [apaun@fmi.unibuc.ro](mailto:apaun@fmi.unibuc.ro)

Seminar: Daniela Cheptea (13), Florin Bilbie (14), TBA (15)

Laboratoare:

- Marius Micluta – Campeanu
- Eduard Stefan Deaconu
- Eduard Szmeteanca
- Octavian Comanescu
- Alexandra Murgoci



## **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:

- regulament privind activitatea studenților la UB: <https://www.unibuc.ro/wp-content/uploads/sites/7/2018/07/Regulament-privind-activitatea-profesionala-a-studentilor-2018.pdf>
- regulament de etică și profesionalism la FMI:

[http://fmi.unibuc.ro/ro/pdf/2015/consiliu/Regulament\\_etica\\_FMI.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, Feb 2022 compared to a year ago:				
Rank	Change	Language	Share	Trend
1		Python	28.52 %	-1.7 %
2		Java	18.12 %	+1.2 %
3		JavaScript	8.9 %	+0.4 %
4	↑	C/C++	7.62 %	+1.1 %
5	↓	C#	7.39 %	+0.6 %
6		PHP	5.81 %	-0.3 %
7		R	4.04 %	+0.2 %
8		Objective-C	2.46 %	-1.1 %
9		Swift	2.03 %	+0.0 %

Majoritatea pot fi considerate limbaje OO.

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



## 2. Utilitatea cursului de Programare Orientată 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 și tehnici de programare**
- **Influențează felul în care sunt gândiți algoritmii 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, preliminare, despre programarea orientată pe obiecte (POO).

- **Obiective specifice:**

- 1. Înțelegerea fundamentelor paradigmii programării orientate pe obiecte;
- 2. Înțelegerea conceptelor de clasă, interfață, moștenire, polimorfism;
- 3. Familiarizarea cu şabloanele 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/iesire, 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 clasa 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 in C++.

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

##### 6. Tratarea exceptiilor in C++.

##### 7. Proiectarea descendenta a claselor. Mostenirea in 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 clasa 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. Moștenirea 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. Şabloane 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 și laborator: fiecare cu 2 ore pe săptămână.

Seminar: 1 ora pe săptămână.

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)
- Test practic
- 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

**Regulamentul de laborator este orientativ. Fiecare tutore de laborator are dreptul sa-l adapteze cerințelor grupelor sale!**

Cele 3 lucrări practice se realizează și se notează în cadrul laboratorului, după următorul program:

**Săptămâna 1:** Test de evaluare a nivelului de intrare.

**Săptămâna 2:** Atribuirea temelor pentru LP1.

**Săptămâna 3:** Consultații pentru LP1.

**Săptămâna 4:** Predare LP1. **Termen predare LP1: TBA.**

**Săptămâna 5:** Evaluarea LP1.

**Săptămâna 6:** Atribuirea temelor pentru LP2.

**Săptămâna 7:** Consultații pentru LP2.

**Săptămâna 8:** Predarea LP2.

**Termen predare LP2: TBA.**

**Săptămâna 9:** Evaluarea LP2.

**Săptămâna 10:** Atribuirea temelor pentru LP3.

**Săptămâna 11:** Consultații pentru LP3.

**Săptămâna 12:** Predarea LP3. **Termen predare LP3: TBA.**

**Săptămâna 13:** Evaluarea LP3.

**Săptămâna 13/14:** Test practic de laborator.

**Prezenta la laborator în săptămânile 1, 2, 5, 6, 9, 10, 13, 14 pentru atribuirea și evaluarea lucrărilor practice și pentru susținerea testului practic este obligatorie.**



### 3. Prezentarea disciplinei

#### 3.4 Regulament de notare si evaluare

**Regulamentul de laborator este orientativ. Fiecare tutore de laborator are dreptul sa-l adapteze cerintelor grupelor sale!**

Consultațiile de laborator se desfășoară pe baza întrebărilor studentilor.

Prezenta la laborator în săptămânile 3, 4, 7, 8, 11, 12 pentru consultații este recomandată, dar facultativă.

Lucrările practice se realizează individual.

Notarea fiecărei lucrări practice se va face cu note de la 1 la 10.

Atribuirea temelor pentru lucrările practice se face prin prezentarea la laborator în săptămâna precizată mai sus sau în oricare din următoarele 2 săptămâni. **Indiferent de data la care un student se prezintă pentru a primi tema pentru una dintre lucrările practice, termenul de predare a acesteia rămâne cel precizat în regulament.** În consecință, tema pentru o lucrare practică nu mai poate fi preluată după expirarea termenului ei de predare.



### 3. Prezentarea disciplinei

#### 3.4 Regulament de notare si evaluare

**Regulamentul de laborator este orientativ. Fiecare tuteur de laborator are dreptul sa-l adapteze cerintelor grupelor sale!**

Predarea lucrarilor practice se face la adresa indicata de tutorele de laborator, inainte de termenele limita de predare, indicate mai sus pentru fiecare LP.

**Dupa expirarea termenelor respective, lucrarea practica se mai poate trimite prin email pentru o perioada de gratie de 2 zile (48 de ore).**

**Pentru fiecare zi partiala de intarziere se vor scadea 2 puncte din nota atribuita pe lucrare.**

**Dupa expirarea termenului de gratie, lucrarea nu va mai fi acceptata si va fi notata cu 1.**



### 3. Prezentarea disciplinei

#### 3.4 Regulament de notare si evaluare

**Nota laborator = medie aritmetica a celor 3 note obtinute pe proiecte.**

Pentru evidențierea unor lucrări practice, tutorele de laborator poate acorda un bonus de **pana la 2 puncte** la nota pe proiecte astfel calculată.

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 – in varianta fata in fata si 2h in varianta online) si va avea un nivel mediu.
- Notare: de la 1 la 10 (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:

Constă dintr-un set de 18 intrebări

- 6 intrebări de teorie
- 12 intrebări practice.

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

**Studentii nu pot lua examenul decat dacă obțin cel puțin 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 lucrările practice (proiecte)
- 25% - nota la testul practic
- 50% - nota la testul scris

**Seminar** - maxim 0.5p care se adauga la nota de la testul scris, **daca si numai daca**, nota de la testul scris  $\geq 5$ .



### 3. Prezentarea disciplinei **Modificari**

- Laborator: notare mai “clara”
- Seminar: 0.5 bonus la nota de la examenul scris pentru max. 20% din studenti
- Prezenta la curs: 0.5 bonus la nota de la examenul scris pentru primii 20% dintre studenti KAHOOT
- **bonusuri dupa ce se promoveaza examenul scris**



### 3. Prezentarea disciplinei

Kahoot

- Se va defini un nume unic de forma popescu131 (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)
- 131: **popescup131** si **popescupr131**



## **Agenda cursului**

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



## 4. Curs 1

### 4.1 Scurta recapitulare C++

- Bjarne Stroustrup în 1979 la Bell Laboratories in Murray Hill, New Jersey
- 5 revizii: 1998 ANSI+ISO, 2003 (corrigendum), 2011 (**C++11/0x**), 2014, 2017 (**C++17/1z**)
- Următoarea planuită în 2020 (C++2a)
- Versiunea 1998: Standard C++, C++98



## 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, constant null pointer, etc.
- C++14: generic lambdas, binary literals, auto, variable template, etc.



## 4. Curs 1

### 4.1 Scurta recapitulare 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 Scurta recapitulare 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 Scurta recapitulare C++

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

- este folosirea aceluiasi nume pentru functii diferite
- functii diferite, dar cu inteleles apropiat
- compilatorul foloseste numarul si tipul parametrilor pentru a diferenția apelurile



## 4.1 Scurta recapitulare C++

```
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;
}
```

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

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

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



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

int myfunc(int i); // these differ in types of parameters
double myfunc(double i);

int main() {
    cout << myfunc(10) << " "; // calls myfunc(int i)
    cout << myfunc(5.4); // calls myfunc(double i)
    return 0;
}

double myfunc(double i) { return i; }

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

tipuri diferite pentru parametrul i



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

```
int myfunc(int i); // these differ in number of parameters
int myfunc(int i, int j);
```

```
int main()
{
    cout << myfunc(10) << " "; // calls myfunc(int i)
    cout << myfunc(4, 5); // calls myfunc(int i, int j)
    return 0;
}
```

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

**numar diferit de parametri**



- daca diferența este doar în tipul de date întors:  
eroare la compilare

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

- sau tipuri care par să fie diferite

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

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



# Ambiguitati pentru polimorfism de functii

- erori la compilare
- majoritatea datorita conversiilor implicite

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



```
#include <iostream>
using namespace std;
float myfunc(float i);
double myfunc(double i);

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

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

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



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

```
#include <iostream>
using namespace std;
char myfunc(unsigned char ch);
char myfunc(char ch);

int main()
{
    cout << myfunc('c'); // this calls myfunc(char)
    cout << myfunc(88) << " "; // ambiguous
    return 0;
}
char myfunc(unsigned char ch)
{
    return ch-1;
}
char myfunc(char ch){return ch+1;}
```

```
int myfunc(int i);
int myfunc(int i, int j=1);
```

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

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

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

- ambiguitate intre char si unsigned char
- ambiguitate pentru functii cu param. impliciti



```
// This program contains an error. #include
<iostream>
using namespace std;
void f(int x);
void f(int &x); // error

int main() {
    int a=10;
    f(a); // error, which f()?
    return 0;
}

void f(int x) { cout << "In f(int)\n"; }
void f(int &x) { cout << "In f(int &)\n"; }
```

- doua tipuri de apel:  
prin valoare si prin  
referinta, ambiguitate!
- mereu eroare de  
ambiguitate



## 4. Curs 1

### 4.1 Scurta recapitulare 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.**

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



## 4. Curs 1

### 4.1 Scurta recapitulare C++

#### Funcții cu valori implicite

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

void f (int a, int b = 12); // prototip cu mentionarea valorii implicate 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 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 [ ]



## 4. Curs 1

### 4.1 Scurta recapitulare C++

#### Tipul referinta

O referință este, în esență, un pointer implicit, care acționează 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 în momentul declarării, devenind un alias (un alt nume) al obiectului cu care a fost inițializată.**



## 4. Curs 1

### 4.1 Scurta recapitulare 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 același tip, incrementarea referinței implica, de fapt, incrementarea valorii referite.**



## 4. Curs 1

### 4.1 Scurta recapitulare C++

#### Funcții 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; }
```



## 4. Curs 1

### 4.1 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 [ ]



## 4. Curs 1

### 4.1 Scurta recapitulare C++

#### Tipul referinta

O referință este, în esență, un pointer implicit, care acționează 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 în momentul declarării, devenind un alias (un alt nume) al obiectului cu care a fost inițializată.**



## 4. Curs 1

### 4.1 Scurta recapitulare 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 același tip, incrementarea referinței implica, de fapt, incrementarea valorii referite.**



## 4. Curs 1

### 4.1 Scurta recapitulare 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 Scurta recapitulare C++

#### Tipul referinta

- o referinta trebuie să fie initializata când este definita, dacă nu este membră a unei clase, un parametru de funcție sau o valoare returnată;
- referintele nule sunt interzise într-un program C++ valid.
- nu se poate obține adresa unei referințe.
- nu se poate face referință către un camp de biti.



## 4. Curs 1

### 4.1 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++

```
#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 Scurta recapitulare C++

#### Transmiterea parametrilor

Observatii generale

- parametrii formalii - sunt creati la intrarea intr-o functie si distrusi la return;
- 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 Scurta recapitulare C++

Când tipul returnat de o funcție nu este declarat explicit, î se atribuie automat int.

Tipul trebuie cunoscut înainte de apel.

```
f (double x)
{
return x;
}
```

**Prototipul unei functii: permite declararea în afara și a numărului de parametri / tipul lor:**

```
void f(int); // antet / prototip
int main() { cout<< f(50); }
void f( int x)
{
    // corp functie;
}
```



## 4.1 Scurta recapitulare 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;
}
```



## 4.1 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.



## 4. Curs 1

### 4.2 Principiile programării orientate pe obiecte

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



## 4. Curs 1

### 4.2 Principiile programării 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



## 4. Curs 1

### 4.2 Principiile programării orientate pe obiecte

**Principalele concepte utilizate în POO sunt:**

**Obiecte**

**Clase**

**Mostenire**

**Ascunderea informației**

**Polimorfism**

**Sabioane – nu sunt utilizate strict POO (mai general, se referă la Programarea generică)**



## 4. Curs 1

### 4.2 Principiile programării orientate pe obiecte

O **clasa** definește atribute și metode.

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

- menționează proprietatile generale ale obiectelor din clasa respectiva
- clasele nu se pot “rula”
- folositoare la encapsulare (ascunderea informației)
- reutilizare de cod: mostenire



## 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 programării orientate pe obiecte

**Principalele concepte (caracteristici) ale POO sunt:**

**Incapsularea** – contopirea datelor cu codul (metode de prelucrare și 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 funcționalități;

- multe obiecte au proprietăți similare
- reutilizare de cod



## 4. Curs 1

### 4.2 Principiile programării orientate pe obiecte

**Principalele concepte (caracteristici) ale POO sunt:**

#### **Ascunderea informatiei**

foarte importantă

**public, protected, private**

<b>Avem acces?</b>	<b>public</b>	<b>protected</b>	<b>private</b>
Aceeasi clasa	da	da	da
Clase derivate	da	da	nu
Alte clase	da	nu	nu



## 4. Curs 1

### 4.2 Principiile programării orientate pe obiecte

**Principalele concepte (caracteristici) ale POO sunt:**

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



## 4. Curs 1

### 4.2 Principiile programării orientate pe obiecte

Alt concept important în POO:

#### Sabioane

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



## Perspective

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

- intelegerarea notiunilor
- intrebari si sugestii

**2. Cursul 2:**

- Introducere in OOP. Clase. Obiecte



# Programare orientată pe obiecte

**- suport de curs -**

**Andrei Păun  
Anca Dobrovăț,**

**An universitar 2021 – 2022  
Semestrul II  
Seriile 13, 14, 15**

**Curs 2**



## Agenda cursului

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



## Generalități despre curs

1. Curs – seria 13: luni (10 -12), seria 14: marti (8 – 10), seria 15: vineri (12 - 14)
2. Laborator – pe semigrupe, în fiecare săptămână
3. Seminar - o dată la 2 săptămâni

**COLOCVIU: 23 mai 2022 (ora 8) – se sustine fizic în facultate**

**EXAMEN SCRIS: 14 iunie 2022 (ora 9) – se sustine fizic în facultate**

Dacă cineva are o problemă cu aceste date îl/o rog să ne anunțe

În 2 săptămâni datele acestea sunt fixate/finalizate



## 1. Scurta recapitulare C++

- Bjarne Stroustrup în 1979 la Bell Laboratories in Murray Hill, New Jersey
- 5 revizii: 1998 ANSI+ISO, 2003 (corrigendum), 2011 (**C++11/0x**), 2014, 2017 (**C++ 17/1z**)
- Următoarea planuită în 2020 (C++2a)
- Versiunea 1998: Standard C++, C++98



## 1. Scurta recapitulare 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.



## 1. Scurta recapitulare 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 ...**)



## 1. Scurta recapitulare 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.



## 1. Scurta recapitulare C++

**Suprîncărcarea 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 diferență**

Simplicitate/corectitudine de cod



## 1. Scurta recapitulare 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.**

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



## 1. Scurta recapitulare C++

### Funcții cu valori implicate

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

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

```
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++

### Tipul referinta

O referință este, în esență, un pointer implicit, care acționează 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 în momentul declarării, devenind un alias (un alt nume) al obiectului cu care a fost inițializată.**



## 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 referinta trebuie să fie initializata când este definita, dacă nu este membră a unei clase, un parametru de funcție sau o valoare returnată;
- referintele nule sunt interzise într-un program C++ valid.
- nu se poate obține adresa unei referințe.
- nu se pot crea tablouri de referințe.
- nu se poate face referință către un camp de biti.



## 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 return;
- 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.



## 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. Prințipiile programării orientate pe obiecte

**Clasa**

**Obiect**

**Incapsularea**

**Modularitatea**

**Ierarhizarea.**

**Polimorfism la compilare și polimorfism la execuție, etc.**



## 2. Principiile programării orientate pe obiecte

### Obiecte

- au stare și acțiuni (metode/funcții)
- au interfață (acțiuni) și 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 programării orientate pe obiecte

### Clase

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



## 2. Prințipiile programării orientate pe obiecte

### Clase

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



## 2. Prințipiile programării orientate pe obiecte

### Clase

**class** nume\_clasă {

**private** variabile și funcții membru

**specificator\_de\_acces:**

        variabile și funcții membru

**specificator\_de\_acces:**

        variabile și funcții membru

// ...

**specificator\_de\_acces:**

        variabile și funcții membru

} listă\_objekte;

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



## 2. Principiile programării 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 programării orientate pe obiecte

### Clase

- se folosește mai mult a doua variantă
- un membru (ne-static) al clasei nu poate avea initializare
- nu putem avea ca membri obiecte de tipul clasei (putem avea pointeri la tipul clasei)
- nu auto, extern, register



## 2. Principiile programării orientate pe obiecte

### Clase

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



## 2. Principiile programării 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 programării orientate pe obiecte

### Clase

- se creează un tip nou de date **stack mystack;**
- un obiect instanțiază clasa
- funcțiile membru sunt date prin semnătură
- pentru definirea fiecărei funcții se folosește ::

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



## 2. Principiile programării orientate pe obiecte

### Clase

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

**stack mystack;**

- mystack.push(5);
- programul complet în continuare



```
#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 programării orientate pe obiecte

### Incapsularea

- ascunderea de informații (data-hiding);
- separarea aspectelor externe ale unui obiect care sunt accesibile altor obiecte de aspectele interne ale obiectului care sunt ascunse celorlalte obiecte;
- definește apartenența unor proprietăți și metode față de un obiect;
- doar metodele proprii ale obiectului pot accesa starea acestuia.



## 2. Principiile programarii orientate pe obiecte

### Incapsularea (ascunderea informației)

foarte importantă  
**public, protected, private**

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



## 2. Principiile programării orientate pe obiecte

### Incapsularea (ascunderea informației)

```
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 programării orientate pe obiecte

- **Agregarea** (ierarhia de obiecte) compunerea unui obiect din mai multe obiecte mai simple. (relație de tip “has a”)
- **Moștenirea** (ierarhia de clase) relație între clase în care o clasă moșteneste structura și comportarea definită în una sau mai multe clase (relație de tip “is a” sau “is like a”);



## 2. Principiile programării orientate pe obiecte

### Agregarea

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

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



## 2. Principiile programării orientate pe obiecte

### Moștenire

- multe obiecte au proprietăți similare
- reutilizare de cod



## 2. Principiile programării orientate pe obiecte

### Moștenire

- terminologie
  - clasă de bază, clasă derivată
  - superclasă subclasă
  - părinte, fiu
- mai târziu: funcții virtuale, identificare de tipuri în timpul rulării (RTTI)



## 2. Principiile programării orientate pe obiecte

### Moștenire

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



## 2. Principiile programării orientate pe obiecte

```
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();};
```

**Moștenire**

**tip acces: public, private, protected**

mai multe mai târziu

public: membrii publici ai building devin publici pentru house



## 2. Principiile programării orientate pe obiecte

### Moștenire

- house NU are acces la membrii privați ai lui building
  - aşa se realizează encapsularea
- 
- clasa derivată are acces la membrii publici ai clasei de baza și la toți membrii săi (publici și privați)



## 2. Principiile programării orientate pe obiecte

```
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();  
};
```

### Moștenire

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



```

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;
}

```

## Moștenire

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



## 2. Principiile programării orientate pe obiecte

### Polimorfism

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



## 2. Principiile programării orientate pe obiecte

### Şabloane

- 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



# Programare orientată pe obiecte

- suport de curs -

**Andrei Păun  
Anca Dobrovăț,**

**An universitar 2021 – 2022  
Semestrul II  
Seriile 13, 14, 15**

**Curs 3**



# Cuprinsul cursului

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



# Struct si class

- singura diferență: struct are default membri ca public iar class ca private
- struct definește o clasa (tip de date)
- putem avea în struct și funcții
- pentru compatibilitate cu cod vechi
- extensibilitate
- **a nu se folosi struct pentru clase**



// Utilizarea unei structuri pentru a defini o clasa.

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

struct mystr {
    void buildstr(char *s)
    { if(!*s) *str = '\0';
      else strcat(str, s);}
    void showstr() {cout << str << "\n"; }

private:
    char str[255];
};
```

```
class mystr {
    char str[255];
public:
    void buildstr(char *s); // public
    void showstr();
};
```



# Union și class

- la fel ca struct
- toate elementele de tip data folosesc aceeași locație de memorie
- membrii sunt publici (by default)



# Union și class

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

union swap_byte {
    void swap() { unsigned char t = c[0]; c[0] = c[1]; c[1] = t; }
    void set_byte(unsigned short i) { u = i; }
    void show_word() { cout << u; }

    unsigned short u;
    unsigned char c[2];
};

int main() {
    swap_byte b;
    b.set_byte(49034);
    b.swap();
    b.show_word();
    return 0;
}
```



# Union ca o clasa

- union nu poate mosteni
- nu se poate mosteni din union
- nu poate avea functii virtuale (nu avem mostenire)
- nu avem variabile de instanta statice
- nu avem referinte in union
- nu avem obiecte care fac overload pe =
- obiecte cu (con/de)structor definiti nu pot fi membri in union



# Union anonte

- nu au nume pentru tip
- nu se pot declara obiecte de tipul respectiv
- folosite pentru a spune compilatorului cum se aloc/procesez variabilele respective in memorie
  - folosesc aceeasi locatie de memorie
- variabilele din union sunt accesibile ca si cum ar fi declarate in blocul respectiv



# Union anonte

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

int main()
{
    union {
        long l;
        double d;
        char s[4];
    } ;
    l = 100000;
    cout << l << " ";
    d = 123.2342;
    cout << d << " ";
    strcpy(s, "hi");
    cout << s;
    return 0;
}
```



# Union anonte

- nu poate avea functii
- nu poate avea private sau protected (fara functii nu avem acces la altceva)
- union-uri anonte globale trebuie precizate ca statice



# Functii prieten

- Cuvantul cheie: **friend**
- pentru accesarea campurilor protected, private din alta clasa
- folositoare la overload-area operatorilor, pentru unele functii de I/O, si portiuni interconectate (exemplu urmeaza)
- in rest nu se prea folosesc



# Functii prieten pentru o clasa

```
#include <iostream>
using namespace std;
class myclass {
    int a, b;
public:
    friend int sum(myclass x); // poate accesa direct a si b private
    void set_ab(int i, int j) { a = i; b = j; }
};

int sum(myclass x) {      return x.a + x.b; }

int main() {
    myclass n;
    n.set_ab(3, 4);
    cout << sum(n);
    return 0;
}
```



# Functii prieten pentru mai multe clase

```
#include <iostream>
using namespace std;
class C2;
class C1 { int x;
public:
    void set_x(int a){x = a;}
    friend void f(C1, C2);
};

class C2 { int y;
public:
    void set_y(int b){y = b;}
    friend void f(C1, C2);
};
```

```
void f(C1 ob1, C2 ob2)
{ cout<<ob1.x + ob2.y<<"\n"; }
```

```
int main()
{
    C1 A;
    C2 B;
    A.set_x(10);
    B.set_y(20);
    f(A,B);
}
```



# Functii prieten din alte obiecte

```
#include <iostream>
using namespace std;
class C2;
class C1 { int x;
public:
    void set_x(int a){x = a;}
    void f(C2);
};

class C2 { int y;
public:
    void set_y(int b){y = b;}
    friend void C1::f(C2);
};
```

```
void C1::f(C2 ob2)
{ cout<<this->x + ob2.y<<"\n";}
```

```
int main()
{
    C1 A;
    C2 B;
    A.set_x(10);
    B.set_y(20);
    A.f(B);
}
```



# Clase prieten

- Declararea unei clase Y ca prieten al unei clase X, are ca efect ca toate functiile membre ale clasei Y au acces la membrii privati ai clasei X.

```
#include <iostream>
class C1 { int x;
public:
    friend class C2;
};

class C2 {
public:
    void set_x(int a, C1& ob){ ob.x = a;}
    int get_x (C1 ob) {return ob.x;}
};

int main()
{
    C1 A;
    C2 B;
    B.set_x(10,A);
    std::cout<<B.get_x(A);
}
```



# Functii inline

- executie rapida
- este o sugestie/cerere pentru compilator
- pentru functii foarte mici
- pot fi si membri ai unei clase
- foarte comune in clase
- doua tipuri: explicit (**inline**) si implicit



# Explicit inline

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

inline int max(int a, int b)
{
    return a>b ? a : b;
}

int main()
{
    cout << max(10, 20);
    cout << " " << max(99, 88);
    return 0;
}
```

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

int main()
{
    cout << (10>20 ? 10 : 20);
    cout << " " << (99>88 ? 99 : 88);
    return 0;
}
```



# Explicit inline în clase

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

class myclass {
    int a, b;
public:
    void init(int i, int j);
    void show();
};

inline void myclass::init(int i, int j)
{ a = i; b = j; }

inline void myclass::show()
{ cout << a << " " << b << "\n"; }
```

```
int main() {
    myclass x;
    x.init(10, 20);
    x.show();
    return 0;
}
```



# Definirea functiilor inline implicit (in clase)

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

class myclass {
    int a, b;
public:
// automatic inline
    void init(int i, int j)
    {
        a = i;
        b = j;
    }

    void show() { cout << a << " " << b << "\n"; }
};
```

```
int main() {
    myclass x;
    x.init(10, 20);
    x.show();
    return 0;
}
```



# Constructori/Destructori

- inițializare automată
- efectueaza operații prealabile utilizării obiectelor create
- obiectele nu sunt statice
- constructor: funcție specială, numele clasei
- constructorii nu pot întoarce valori (nu au tip de întoarcere)



# Constructori/Destructori

Caracteristici speciale:

- numele = numele clasei (~ numele clasei pentru destructori);
- la declarare nu se specifica tipul returnat;
- nu pot fi mosteniti, dar pot fi apelati de clasele derivate;
- nu se pot utiliza pointeri către functiile constructor / destructor;
- constructorii pot avea parametri (inclusiv impliciti) și se pot supradefini. Destructorul este unic și fără parametri.



# Constructori/Destructori

Constructorii de copiere (discuții mai ample mai tarziu)

- creaza un obiect preluand valorile corespunzatoare altui obiect;
- exista implicit (copiaza bit-cu-bit, deci trebuie redefinit la date alocate dinamic).



# Constructori/Destructori

*Orice clasa, are by default:*

- un constructor de initializare
- un constructor de copiere
- un destructor
- un operator de atribuire.

**Constructorii parametrizati:**

- argumente la constructori
  - putem defini mai multe variante cu mai multe numere si tipuri de parametri
- overload de constructori (mai multe variante, cu numar mai mare și tipuri de parametri).



# Constructori/Destructori

*Orice clasa, are by default:*

```
class A
{
    int x;
    float y;
    string z;
};

int main()
{
    A a;      // apel constructor de initializare - fara parametri
    A b = a; // apel constructor de copiere
    A e(a); // apel constructor de copiere
    A c;      // apel constructor de initializare
    c = a;    //operatorul de atribuire (=)
}
```



# Constructori/Destructori

## Exemplu – necesitate rescriere constructori

```
class A
{
    int *v;
public:
    A() {v = new int[10]; cout<<"C";}
    ~A() {delete[] v;   cout<<"D";}
    void afis() {cout<<v[3];}
};

void functie (A ob) { ob.afis(); }

int main()
{
    A o1;
    afisare(o1);
    o1.afis();
}
```



# Constructori/Destructori

## Exemplu - Constructori parametrizati

```
class A {  
    int x;  
    float y;  
    string z;  
public:  
    A() {x = -45; }  
    A(int x) {this->x = x; this->y = 5.67; z = "Seria 14"; }  
    // this -> camp ⇔ camp simplu: this ->z echivalent cu z  
    A(int x, float y) {this->x = x; this->y = y; z = "Seria 13 25"; }  
    A(int x, float y, string z) {this->x = x; this->y = y; this->z= z; }  
  
int main() {  
    A a;  
    A b(34);  
    return 0;  
}
```



# Constructori/Destructori

## Exemplu - Constructori parametrizati

```
class A
{
    int x;
    float y;
    string z;
public:
    A(int x = -45, float y = 5.67, string z = "Seria 13") // valori implicite
        {this->x = x; this->y = y; z = z;}
};

int main()
{
A a;
A b(34);
return 0;
}
```



# Constructori/Destructori

**Functiile constructor cu un parametru – caz special (sursa H. Schildt)**

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

class X {
    int a;
public:
    X(int j) { a = j; }
    int geta() { return a; }
};

int main()
{
    X ob = 99; // passes 99 to j
    cout << ob.geta(); // outputs 99
    return 0;
}
```

Se creeaza o conversie  
explicita de date!



# Constructori/Destructori

## Tablouri de obiecte

Daca o clasa are constructori parametrizati, putem initializa diferit fiecare obiect din vector.

```
class X{int a,b,c; ... };
X v[3] = {X(10,20) , X (1,2,3), X(0) };
```

Daca constr are un singur parametru, atunci se poate specifica direct valoarea.

```
class X {
    int i;
public:
    X(int j) { i = j;};
};
```

**X v[3] = {10,15,20 };**



# Constructori/Destructori

## Exemplu – Ordine apel

```

class A
{
    int x;
public:
    A(int x = 0)    {
        x = x;
        cout << "Constructor" << x << endl;  }
    ~A()    {
        cout << "Destructor" << endl;  }
    A(const A&o)    {
        x = o.x;
        cout << "Constructor de copiere" << endl;  }
    void f_cu_referinta(A& ob3)    {
        A ob4(456);  }
    void f_fara_referinta(A ob6)    {
        A ob7(123);  }
} ob;

```

```

int main()
{
    A ob1(20), ob2(55);
    ob2.f_cu_referinta(ob1);
    ob1.f_fara_referinta(ob2);
    A ob5;
    return 0;
}

```



# Constructori/Destructori

## Exemplu – Ordine apel

- 1) In acelasi domeniu de vizibilitate, constructorii se apeleaza in ordinea declararii obiectelor, iar destructorii in sens invers.
- 2) Variabilele globale se declara inaintea celor locale, deci constructorii lor se declara primii.
- 3) Daca o functie are ca parametru un obiect, care nu e transmis cu referinta atunci, se activeaza constructorul de copiere si, implicit, la iesirea din functie, obiectul copie se distrugе, deci se apeleaza destructor

```
/// Ordine:  
/// 1. Constructor ob;  
/// 2. Constructor ob1;  
/// 3. Constructor ob2;  
/// 4. Constructor ob4; - ob3 e referinta/alias-ul ob1, nu se creeaza obiect nou  
/// 5. Destructor ob4;  
/// 6. Constructor copiere ob6  
/// 7. Constructor ob7  
/// 8. Destructor ob7  
/// 9. Destructor ob6  
/// 10. Constructor ob5  
/// 11. Destructor ob5  
/// 12. Destructor ob2  
/// 13. Destructor ob1  
/// 14. Destructor ob
```



# Constructori/Destructori

## Exemplu – Ordine apel

```
class A {  
public:  
    A() {cout<<"Constr A"<<endl;}  
};  
  
class B {  
public:  
    B() {cout<<"Constr B"<<endl;}  
private:  
    A ob;  
};  
  
int main()  
{  
    B ob2;  
    // Apel constructor obiect A si apoi constructorul propriu  
}
```



# Constructori/Destructori

```
class A {  
    int x;  
public:  
    A(int x = 7){this->x = x; cout<<"Const "<<x<<endl;}  
    void set_x(int x){this->x = x;}  
    int get_x(){ return x;}  
    ~A(){cout<<"Dest "<<x<<endl;}  
};  
  
void afisare(A ob) {  
    ob.set_x(10);  
    cout<<ob.get_x()<<endl; }  
  
int main () {  
    A o1;  
    cout<<o1.get_x()<<endl;  
    afisare(o1);  
    return 0;  
}
```

Exemplu – Ce se afiseaza?

Const 7 // obiect o1  
7 // o1.get\_x()  
10 // in functie ob.get\_x()  
Dest 10 // ob  
Dest 7 // o1



# Constructori/Destructori

```
class cls { public:  
    cls() { cout << "Inside constructor 1" << endl; }  
    ~cls() { cout << "Inside destructor 1" << endl; } };
```

```
class clss {  
    cls xx;  
public:  
    clss() { cout << "Inside constructor 2" << endl; }  
    ~clss() { cout << "Inside destructor 2" << endl; } };
```

```
class clss2 {  
    clss xx;  
    cls xxx;  
public:  
    clss2() { cout << "Inside constructor 3" << endl; }  
    ~clss2() { cout << "Inside destructor 3" << endl; } };
```

```
int main() {  
    clss2 s;  
}
```

**Exemplu – Ce se afiseaza?**



# Polimorfism pe constructori

- foarte comun să fie supraincarcați
- de ce?
  - flexibilitate
  - pentru a putea defini obiecte initializate și neinitializate
  - constructori de copiere: copy constructors



# overload pe constructori: flexibilitate

- putem avea mai multe posibilitati pentru initializarea/construirea unui obiect
- definim constructori pentru toate modurile de initializare
- daca seincearca initializarea intr-un alt fel (decat cele definite): eroare la compilare



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

class date { int day, month, year;
public:
    date(char *d);
    date(int m, int d, int y);
    void show_date();
};

// Initialize using string.
date::date(char *d)
{ sscanf(d, "%d%*c%d%*c%d", &month, &day, &year);}

// Initialize using integers.
date::date(int m, int d, int y)
{ day = d; month = m; year = y; }
```

```
void date::show_date()
{
    cout << month << "/" << day;
    cout << "/" << year << "\n";
}

int main()
{
    date ob1(char[4], 2003), ob2("10/22/2003");
    ob1.show_date(); Enter new date: ;
    ob2.show_date();
    return 0; date d(s);
}
    d.show_date();
return 0;
```

"%d%\*c%d%\*c%d

citim din sir

\*: ignoram ce citim

c: un singur caracter

citim 3 intregi sau  
luna/zi/an



# polimorfism de constructori: obiecte initializate și ne-initializate

- important pentru array-uri dinamice de obiecte
- nu se pot initializa obiectele dintr-o lista alocată dinamic
- asadar avem nevoie de posibilitatea de a crea obiecte neinitializate (din lista dinamica) și obiecte initializate (definite normal)



```
#include <iostream>
#include <new>
using namespace std;
class powers
{ int x;
public:
    // overload constructor two ways
```

- ofThree și lista p au nevoie de constructorul fara parametri

```
};

int main()
{
    powers ofTwo[] = {1, 2, 4, 8, 16}; // initialized
    powers ofThree[5]; // uninitialized
    powers *p;
    int i; // show powers of two
    cout << "Powers of two: ";
    for(i=0; i<5; i++) {
        cout << ofTwo[i].getx() << " ";
    }
    cout << "\n\n";
```

```
// set powers of three
ofThree[0].setx(1); ofThree[1].setx(3);
ofThree[2].setx(9); ofThree[3].setx(27);
ofThree[4].setx(81);

// show powers of three
cout << "Powers of three: ";
for(i=0; i<5; i++) { cout << ofThree[i].getx() << " ";
}
cout << "\n\n";
```

```
return 1;}
```

```
// initialize dynamic array with powers of two
for(i=0; i<5; i++) { p[i].setx(ofTwo[i].getx());}
```

```
// show powers of two
cout << "Powers of two: ";
for(i=0; i<5; i++) { cout << p[i].getx() << " ";
}
cout << "\n\n";
delete [] p;
return 0;
}
```



# polimorfism de constructori: constructorul de copiere

- pot aparea probleme cand un obiect initializeaza un alt obiect
  - MyClass B = A;
- aici se copiaza toate campurile (starea) obiectului A in obiectul B
- problema apare la alocare dinamica de memorie:  
A si B folosesc aceeasi zona de memorie pentru ca pointerii arata in acelasi loc
- destructorul lui MyClass elibereaza aceeasi zona de memorie de doua ori (distrugе A si B)



# constructorul de copiere

- aceeasi problema
  - apel de functie cu obiect ca parametru
  - apel de functie cu obiect ca variabila de intoarcere
    - in aceste cazuri un obiect temporar este creat, se copiaza prin constructorul de copiere in obiectul temporar, si apoi se continua
    - deci vor fi din nou doua distrugeri de obiecte din clasa respectiva (una pentru parametru, una pentru obiectul temporar)



# constructorul de copiere

Cazuri de utilizare:

**Initializare explicita:**

```
MyClass B = A;  
MyClass B (A);
```

**Apel de functie cu obiect ca parametru:**

```
void f(MyClass X) {...}
```

**Apel de functie cu obiect ca variabila de intoarcere:**

```
MyClass f() {MyClass obiect; ... return obiect;}.  
MyClass x = f();
```

Copierea se poate face și prin operatorul `=` (*detalii mai târziu*).



# putem redefini constructorul de copiere

```
classname (const classname &o) {  
    // body of constructor  
}
```

- *o* este obiectul din dreapta
- putem avea mai multi parametri (dar trebuie sa definim valori implicite pentru ei)
- & este apel prin referinta
- putem avea si atribuire (*o1=o2;*)
  - redefinim operatorii mai tarziu, putem redefini =
  - = diferit de initializare

# Facultatea de Matematică și Informatică Universitatea din București



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

class array {
    int *p;
    int size;
public:
    array(int sz) {
        try {
            p = new int[sz];
        } catch (bad_alloc xa) {
            cout << "Allocation Failure\n";
            exit(EXIT_FAILURE);
        }
        size = sz;
    }
    ~array() { delete [] p; }

    // copy constructor
    array(const array &a);
    void put(int i, int j) {if(i>=0 && i<size) p[i] = j;}
    int get(int i) { return p[i];}
};
```

```
// Copy Constructor
array::array(const array &a) {
    int i;
    try {
        p = new int[a.size];
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        exit(EXIT_FAILURE);
    }
    for(i=0; i<a.size; i++) p[i] = a.p[i];
}
```

```
int main()
{
    array num(10);
    int i;
    for(i=0; i<10; i++) num.put(i, i);
    for(i=9; i>=0; i--) cout << num.get(i);
    cout << "\n";

    // create another array and initialize with num
    array x(num); // invokes copy constructor
    for(i=0; i<10; i++) cout << x.get(i);
    return 0;
}
```



- Observatie: constructorul de copiere este folosit doar la initializari
- daca avem
  - array a(10);
  - array b(10);
  - b=a;
  - nu este initializare, este copiere de stare
  - este posibil sa trebuiasca redefinit si operatorul = (mai tarziu)



# Perspective

## Curs 4

- Static, clase locale
- Operatorul ::
- supraincarcarea functiilor in C++
- supraincarcarea operatorilor in C++



# **Programare orientată pe obiecte**

**- suport de curs -**

**Andrei Păun  
Anca Dobrovăț,**

**An universitar 2021 – 2022  
Semestrul II  
Seriile 13, 14, 15**

**Curs 4**



# Cuprins

- Recapitulare curs 3 (+ move constructor, move assignment)
- Static, clase locale
- Operatorul ::
- supraincarcarea operatorilor in C++



## Membrii statici ai unei clase

- date membre:
  - nestatice (distincte pentru fiecare obiect);
  - **static** (unice pentru toate obiectele clasei, există o singura copie pentru toate obiectele).
- cuvant cheie “**static**”
- create, initialize și accesate – independent de obiectele clasei.
- alocarea și initializarea – în afara clasei.



## Membrii statici ai unei clase

- functiile statice:
  - efectueaza operatii asupra intregii clase;
  - nu au cuvantul cheie “this”;
  - se pot referi doar la membrii statici.
  
- referirea membrilor statici:
  - clasa :: membru;
  - obiect.membru (identic cu nestatic).



# Folosirea uzuala a functiilor statice

```
#include <iostream>
using namespace std;
class static_type {
    static int i;
public:
    static void init(int x) { i = x; }
    void show() {cout << i;}
};
int static_type::i; // define i
int main()
{
    // init static data before object creation
    static_type::init(100);
    static_type x;
    x.show(); // displays 100
    return 0;
}
```



# Operatorul de rezolutie de scop ::

```
int i; // global i
```

```
void f()
```

```
{
```

```
    int i; // local i
```

```
    i = 10; // uses local i.
```

```
}
```

```
int i; // global i
```

```
void f()
```

```
{
```

```
    int I = 7; // local i
```

```
    ::i = 10; // now refers to global i
```

```
Cout<<::I;
```

```
}
```



# Clase locale

- putem defini clase în clase sau funcții
- **class** este o declaratie, deci definește un scop
- operatorul de rezolutie de scop ajuta în aceste cazuri
- rar utilizate clase în clase



```
#include <iostream>
using namespace std;
void f();
int main() {
    f(); // myclass not known here
    return 0; }

void f() {
    class myclass
    {
        int i;
    public:
        void put_i(int n) { i=n; }
        int get_i() { return i; }
    } ob;
    ob.put_i(10);
    cout << ob.get_i();
}
```

- exemplu de clasa în funcția f()
- restrictii: functii definite în clasa
- nu accesează variabilele locale ale funcției
- accesează variabilele definite static
- fără variabile static definite în clasa



# Functii care intorc obiecte

- o functie poate intoarce obiecte
- un obiect temporar este creat automat pentru a tine informatiile din obiectul de intors
- acesta este obiectul care este intors
- dupa ce valoarea a fost intoarsa, acest obiect este distrus
- probleme cu memoria dinamica: solutie **polimorfism pe = si pe constructorul de copiere**



```
// Returning objects from a function.  
#include <iostream>  
using namespace std;  
  
class myclass  
{  
    int i;  
  
public:  
    Myclass(){  
        void set_i(int n) { i=n; }  
        int get_i() { return i; }  
    };  
  
    myclass f(); // return object of type myclass
```

```
int main()  
{  
    myclass o;  
    o = f();  
    cout << o.get_i() << "\n";  
    return 0;  
}  
  
myclass f()  
{  
    myclass x;  
    x.set_i(1);  
    return x;  
}
```



# copierea prin operatorul =

- este posibil să dam valoarea unui obiect altui obiect
- trebuie să fie de același tip (aceeași clasa)



# Supraincarcarea operatorilor în C++

- majoritatea operatorilor pot fi supraincarcati
- similar ca la functii
- una din proprietatile C++ care ii confera putere
- s-a facut supraincarcarea operatorilor si pentru operatii de I/O (<<, >>)
- supraincarcarea se face definind o functie operator: membru al clasei sau nu



# functii operator membri ai clasei

```
ret-type class-name::operator#(arg-list)
{
// operations
}
```

- # este operatorul supraincarcat (+ - \* / ++ -- = , etc.)
- deobicei ret-type este tipul clasei, dar avem flexibilitate
- pentru operatori unari arg-list este vida
- pentru operatori binari: arg-list contine un element



```
class loc {
    int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {
        longitude = lg;
        latitude = lt; }
    void show() {
        cout << longitude << " ";
        cout << latitude << "\n";
    }
    loc operator+(loc op2);
};
```

// Overload + for loc.

```
loc loc::operator+(loc op2)
{
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;
}
```

```
int main()
{
    loc ob1(10, 20), ob2( 5, 30);
    ob1.show(); // displays 10 20
    ob2.show(); // displays 5 30
    ob1 = ob1 + ob2;
    ob1.show(); // displays 15 50
    return 0;
}
```

- un singur argument pentru ca avem **this**
- longitude==this->longitude
- obiectul din stanga face apelul la functia operator
  - ob1 a chemat operatorul + redefinit in clasa lui ob1



- daca intoarcem acelasi tip de date in operator putem avea expresii
- daca intorceam alt tip nu puteam face
$$ob1 = ob1 + ob2;$$
- putem avea si
$$(ob1+ob2).show(); // displays outcome of ob1+ob2$$
- pentru ca functia show() este definita in clasa lui ob1
- se genereaza un obiect temporar
  - (constructor de copiere)



```
#include <iostream>
using namespace std;
class loc { int longitude, latitude;
public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) { longitude = lg; latitude = lt; }
    void show() { cout<<longitude<<" "<<latitude<<"\n"; }
    loc operator+(loc op2);
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};

// Overload + for loc.
loc loc::operator+(loc op2){ loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp; }

loc loc::operator-(loc op2){ loc temp;
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp; }

// Overload asignment for loc.
loc loc::operator=(loc op2){
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; } // object that generated call

// Overload prefix ++ for loc.
loc loc::operator++(){
    longitude++;
    latitude++;
    return *this; }

int main(){
    loc ob1(10, 20), ob2( 5, 30), ob3(90, 90);
    ob1.show(); ob2.show();
    ++ob1; ob1.show(); // displays 11 21
    ob2 = ++ob1; ob1.show(); // displays 12 22
    ob2.show(); // displays 12 22
    ob1 = ob2 = ob3; // multiple assignment
    ob1.show(); // displays 90 90
    ob2.show(); // displays 90 90
    return 0; }
```



- apelul la functia operator se face din obiectul din stanga (pentru operatori binari)
  - din aceasta cauza pentru – avem functia definita asa
- operatorul = face copiere pe variabilele de instanta, intoarce \*this
- se pot face atribuirii multiple (dreapta spre stanga)



# Formele prefix și postfix

- am vazut prefix, pentru postfix: definim un parametru int “dummy”

```
// Prefix increment
type operator++() {
    // body of prefix operator
}
```

```
// Postfix increment
type operator++( int x) {
    // body of postfix operator
}
```



# supraincarcarea +=, \*=, etc.

```
loc loc::operator+=(loc op2)
{
    longitude = op2.longitude + longitude;
    latitude = op2.latitude + latitude;
    return *this;
}
```



# Restrictii

- nu se poate redefini și precedenta operatorilor
- nu se poate redefini numarul de operanzi
  - rezonabil pentru ca redefinim pentru lizibilitate
  - putem ignora un operand dacă vrem
- nu putem avea valori implicate; excepție pentru ( )
- **nu putem face overload pe . (acces de membru)**  
**:: (rezoluție de scop)**  
**.\*(acces membru prin pointer)**  
**? (ternar)**
- e bine să facem operațiuni apropriate de înțelesul operatorilor respectivi



- Este posibil sa facem o decuplare completa intre intelesul initial al operatorului
  - exemplu: << >>
- mostenire: operatorii (mai putin =) sunt mosteniti de clasa derivata
- clasa derivata poate sa isi redefineasca operatorii



# Supraincarcarea operatorilor ca functii prieten

- operatorii pot fi definiti si ca functie nememra a clasei
- o facem functie prietena pentru a putea accesa rapid campurile protejate
- nu avem pointerul “this”
- deci vom avea nevoie de toti operanzii ca parametri pentru functia operator
- primul parametru este operandul din stanga, al doilea parametru este operandul din dreapta



# Facultatea de Matematică și Informatică

## Universitatea din București

```
#include <iostream>
using namespace std;
class loc { int longitude, latitude;
public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) { longitude = lg; latitude = lt; }
    void show() { cout<<longitude<<" "<<latitude<<"\n"; }
    friend loc operator+(loc op1, loc op2); // friend
    loc operator-(loc op2);
    loc operator=(loc op2);
    loc operator++();
};

// Now, + is overloaded using friend function.
loc operator+(loc op1, loc op2){
    loc temp;
    temp.longitude = op1.longitude + op2.longitude;
    temp.latitude = op1.latitude + op2.latitude;
    return temp;
}
```

```
loc loc::operator-(loc op2){ loc temp;
    // notice order of operands
    temp.longitude = longitude - op2.longitude;
    temp.latitude = latitude - op2.latitude;
    return temp; }

// Overload assignment for loc.
loc loc::operator=(loc op2){
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; } // object that generated call

loc loc::operator++(){
    longitude++;
    latitude++;
    return *this; }

int main(){
    loc ob1(10, 20), ob2( 5, 30);
    ob1 = ob1 + ob2;
    ob1.show();
    return 0; }
```



# Restrictii pentru operatorii definiti ca prieten

- nu se pot supraincarca = () [] sau  $\rightarrow$  cu functii prieten
- pentru ++ sau -- trebuie sa folosim referinte



# functii prieten pentru operatori unari

- pentru `++`, `--` folosim referinta pentru a transmite operandul
  - pentru ca trebuie sa se modifice si nu avem pointerul `this`
  - apel prin valoare: primim o copie a obiectului si nu putem modifica operandul (ci doar copia)



```
#include <iostream>
using namespace std;
class loc { int longitude, latitude;
public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) { longitude = lg; latitude = lt; }
    void show() { cout<<longitude<<" "<<latitude<<"\n"; }
    loc operator=(loc op2);
    friend loc operator++(loc& op);
    friend loc operator--(loc& op);
};

// Overload assignment for loc.
loc loc::operator=(loc op2){
    longitude = op2.longitude;
    latitude = op2.latitude;
    return *this; } // object that generated call

// Now a friend, use a reference parameter.
loc operator++(loc& op) {
    op.longitude++;
    op.latitude++;
    return op;
}

// Make – a friend. Use reference
loc operator--(loc& op) {
    op.longitude--;
    op.latitude--;
    return op;
}

int main(){
    loc ob1(10, 20), ob2;
    ob1.show();
    ++ob1;
    ob1.show(); // displays 11 21
    ob2 = ++ob1;
    ob2.show(); // displays 12 22
    --ob2;
    ob2.show(); // displays 11 21
    return 0;}
```



# pentru varianta postfix ++ --

- la fel ca la supraincărcarea operatorilor prin functii membru ale clasei: parametru int

```
// friend, postfix version of ++
friend loc operator++(loc &op, int x);
```



# Diferente supraincarcarea prin membri sau prieteni

- de multe ori nu avem diferente,
  - atunci e indicat sa folosim functii membru
- uneori avem insa diferente: pozitia operanzilor
  - pentru functii membru operandul din stanga apeleaza functia operator supraincarcata
  - daca vrem sa scriem expresie: 100+ob; probleme la compilare=> functii prieten



- în aceste cazuri trebuie să definim două funcții de supraincarcare:
  - int + tipClasa
  - tipClasa + int



```
#include <iostream>
using namespace std;
class loc { int longitude, latitude;
public:
    loc() {} // needed to construct temporaries
    loc(int lg, int lt) { longitude = lg; latitude = lt; }
    void show() { cout<<longitude<<" "<<latitude<<"\n";}
    loc operator=(loc op2);
    friend loc operator+(loc op1, int op2);
    friend loc operator+(int op1, loc op2);
};

// + is overloaded for loc + int.
loc operator+(loc op1, int op2){
    loc temp;
    temp.longitude = op1.longitude + op2;
    temp.latitude = op1.latitude + op2;
    return temp;
}

// + is overloaded for int + loc.
loc operator+(int op1, loc op2){
    loc temp;
    temp.longitude = op1 + op2.longitude;
    temp.latitude = op1 + op2.latitude;
    return temp;
}
```

```
int main(){
    loc ob1(10, 20), ob2(5, 30), ob3(7, 14);
    ob1.show();
    ob2.show();
    ob3.show();
    ob1 = ob2 + 10; // both of these
    ob3 = 10 + ob2; // are valid
    ob1.show();
    ob3.show();

    return 0;
}
```



# supraincarcarea new și delete

- supraincarcare op. de folosire memorie in mod dinamic pentru cazuri speciale

```
// Allocate an object.  
void *operator new(size_t size){  
    /* Perform allocation. Throw bad_alloc on  
failure. Constructor called automatically. */  
    return pointer_to_memory;  
}  
  
// Delete an object.  
void operator delete(void *p){  
    /* Free memory pointed to by p. Destructor called  
automatically. */  
}
```

- **size\_t**: predefinit
- pentru new: constructorul este chemat automat
- pentru delete: destructorul este chemat automat
- supraincarcare la nivel de clasa sau globala



```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc { int longitude, latitude;
public:          In overloaded new.
    loc() {}        In overloaded new.
    loc(int lg, int lt) {longitude = lg; 10 20
    void show() { cout -10 -20
cout << latitude << "\r"           In overloaded delete.
    void *operator new             In overloaded delete.
    void operator delete(void *p) { // delete overloaded relative to loc.
        cout << "In overloaded delete.\n";
        cout << p1 << "\r"; return 1;
    }
    cout << p2 << "\r"; return 1;
}
// new overloaded relat
void *loc::operator new(void *p;
cout << "In overloaded new." ,      return 0; }
p = malloc(size);
if(!p) { bad_alloc ba; throw ba; }
return p;}
```



- daca new sau delete sunt folositi pentru alt tip de date in program, versiunile originale sunt folosite
- se poate face overload pe new si delete la nivel global
  - se declara in afara oricarei clase
  - pentru new/delete definiti si global si in clasa, cel din clasa e folosit pentru elemente de tipul clasei, si in rest e folosit cel redefinit global



```
#include <iostream>
#include <cstdlib>
#include <new>
using namespace std;
class loc { int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt)
        {longitude = lg; latitude = lt;}
    void show() {cout << longitude << " ";
        cout << latitude << "\n";}
};
// Global new
void *operator new(size_t size) {
    void *p;
    p = malloc(size);
    if(!p) { bad_alloc ba; throw ba; }
return p;
}
```

```
// Global delete
void operator delete(void *p) { free(p); }
int main()
{
    loc *p1, *p2;
    float *f;
    try {p1 = new loc (10, 20); }
    catch (bad_alloc xa) {
        cout << "Allocation error for p1.\n";
        return 1; }
    try {p2 = new loc (-10, -20); }
    catch (bad_alloc xa) {
        cout << "Allocation error for p2.\n";
        return 1; }
    try {
        f = new float; // uses overloaded new, too }
    catch (bad_alloc xa) {
        cout << "Allocation error for f.\n";
        return 1; }
    *f = 10.10F;
    cout << *f << "\n";
    p1->show();
    p2->show();
    delete p1; delete p2; delete f;
    return 0; }
```



# new și delete pentru array-uri

- facem overload de două ori

```
// Allocate an array of objects.  
void *operator new[](size_t size) {  
    /* Perform allocation. Throw bad_alloc on failure.  
    Constructor for each element called automatically. */  
    return pointer_to_memory;  
}  
  
// Delete an array of objects.  
void operator delete[](void *p) {  
    /* Free memory pointed to by p. Destructor for each  
    element called automatically. */  
}
```



# supraincarcarea []

- trebuie sa fie functii membru, (nestatice)
- nu pot fi functii prieten
- este considerat operator binar
- o[3] se transforma in
  - o.operator[](3)

```
type class-name::operator[](int i)
```

```
{  
// ...  
}
```



```
#include <iostream>
using namespace std;
class atype { int a[3];
public:
    atype(int i, int j, int k) { a[0] = i; a[1] = j; a[2] = k; }
    int operator[](int i) { return a[i]; }
};
int main() {
    atype ob(1, 2, 3);
    cout << ob[1]; // displays 2
    return 0;
}
```



- operatorul [] poate fi folosit și la stanga unei atribuiriri (obiectul întors este atunci referinta)



```
#include <iostream>
using namespace std;
class atype { int a[3];
public:
    atype(int i, int j, int k) { a[0] = i; a[1] = j; a[2] = k; }
    int &operator[](int i) { return a[i]; }
};
int main() {
    atype ob(1, 2, 3);
    cout << ob[1]; // displays 2
    cout << " ";
    ob[1] = 25; // [] on left of =
    cout << ob[1]; // now displays 25
    return 0;
}
```

- putem în acest fel verifica array-urile
- exemplul urmator



```
// A safe array example.  
#include <iostream>  
#include <cstdlib>  
using namespace std;  
class atype { int a[3];  
public:  
    atype(int i, int j, int k) {a[0] = i; a[1] = j; a[2] = k;}  
    int &operator[](int i);  
};
```

```
// Provide range checking for atype.  
int &atype::operator[](int i)  
{  
    if(i<0 || i> 2) { cout << "Boundary Error\n"; exit(1); }  
    return a[i];  
}
```

```
int main() {  
    atype ob(1, 2, 3);  
    cout << ob[1]; // displays 2  
    cout << " ";  
    ob[1] = 25; // [] appears on left  
    cout << ob[1]; // displays 25  
    ob[3] = 44;  
        // generates runtime error, 3 out-of-range  
    return 0; }
```



# supraincarcarea ()

- nu creem un nou fel de a chema functii
- definim un mod de a chema functii cu numar arbitrar de parametrii

```
double operator()(int a, float f, char *s);
O(10, 23.34, "hi");
      echivalent cu O.operator()(10, 23.34, "hi");
```



```
#include <iostream>
using namespace std;
class loc { int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {longitude = lg;
    latitude = lt;}
    void show() {cout << longitude << " ";
    cout << latitude << "\n";}
    loc operator+(loc op2);
    loc operator()(int i, int j);
};

// Overload () for loc.
loc loc::operator()(int i, int j) {
    longitude = i; latitude = j;
    return *this;
}
```

Overload + for loc.

```
loc loc::operator+(loc op2) {
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude; return
    temp;
```

void main() { loc ob1(10, 20), ob2(1, 1);
 b1.show();
 b1(7, 8); // can be executed by itself ob1.show();
 b1 = ob2 + ob1(10, 10); // can be used in
 expressions
 b1.show(); }

10 20  
7 8  
11 11



# overload pe ->

- operator unar
- obiect->element
  - obiect genereaza apelul
  - element trebuie sa fie accesibil
  - intoarce un pointer catre un obiect din clasa



```
#include <iostream>
using namespace std;
class myclass {
    public:
    int i;
    myclass *operator->() {return this;}
};

int main() {
    myclass ob; ob->i = 10; // same as ob.i
    cout << ob.i << " " << ob->i;
return 0;
}
```



# supraincarcarea operatorului ,

- operator binar
- ar trebui ignorate toate valorile mai putin a celui mai din dreapta operand



```
#include <iostream>
using namespace std;
class loc { int longitude, latitude;
public:
    loc() {}
    loc(int lg, int lt) {longitude = lg; latitude = lt;}
    void show() {cout << longitude << " ";
    cout << latitude << "\n";}
    loc operator+(loc op2);
    loc operator,(loc op2);
};

// overload comma for loc
loc loc::operator,(loc op2){
    loc temp;
    temp.longitude = op2.longitude;
    temp.latitude = op2.latitude;
    cout << op2.longitude << " ";
    cout << op2.latitude << "\n";
    return temp;
}

// Overload + for loc
loc loc::operator+(loc op2) {
    loc temp;
    temp.longitude = op2.longitude + longitude;
    temp.latitude = op2.latitude + latitude;
    return temp;
}

int main() {
    loc ob1(10, 20), ob2( 5, 30), ob3(1, 1); ob1.show();
    ob3.show();
    '\n';
    ob1, ob2+ob2, ob3);
    () // displays 1 1, the value of ob3
    ;
}
```

<b>10 20</b> <b>5 30</b> <b>1 1</b> <b>10 60</b> <b>1 1</b> <b>1 1</b>	<b>10 20</b> <b>5 30</b> <b>1 1</b> <b>10 60</b> <b>1 1</b> <b>1 1</b>
---	---



# 4. Static, supraîncărcarea funcțiilor, pointeri către funcții Perspective

Curs 5

Recapitulare (static, parametrii default funcții) și

- supraîncărcarea funcțiilor in C++
- supraîncărcarea operatorilor in C++



# **Programare orientată pe obiecte**

## **- suport de curs -**

**Andrei Păun**  
**Anca Dobrovăț**

**An universitar 2021 – 2022**  
**Semestrul II**  
**Seriile 13, 14 și 15**

**Curs 6**



# Cuprins

Proiectarea descendenta a claselor. Moștenirea în C++.

- Controlul accesului la clasa de bază.
- Constructori, destrutori și moștenire.
- Redefinirea membrilor unei clase de bază într-o clasă derivată.
- Declarații de acces.

*Obs: în acest curs, exemplele vor fi luate, în principal, din cartea lui B. Eckel - Thinking in C++.*



## **Moștenirea în C++**

- important în C++ - reutilizare de cod;
- reutilizare de cod prin creare de noi clase (nu se dorește crearea de clase de la zero);
- 2 modalități (compunere și moștenire);
- “compunere” - noua clasă “este compusă” din obiecte reprezentând instanțe ale claselor deja create;
- “moștenire” - se creează un nou tip al unei clase deja existente.



## Moștenirea în C++

### Exemplu: compunere

```
class X { int i;  
public:  
    X() { i = 0; }  
    void set(int ii) { i = ii; }  
};  
class Y { int i;  
public:  
    X x; // Embedded object  
    Y() { i = 0; }  
    void f(int ii) { i = ii; }  
};  
int main() {  
    Y y;  
    y.f(47);  
    y.x.set(37); // Access the embedded object  
}
```



## Moștenirea în C++

C++ permite moștenirea ceea ce înseamnă că putem deriva o clasă din altă clasă de bază sau din mai multe clase.

Prin derivare se obțin clase noi, numite clase deriveate, care moștenesc proprietățile unei clase deja definite, numită clasă de bază.

Clasele deriveate conțin toți membrii clasei de bază, la care se adaugă noi membri, date și funcții membre.

Dintr-o clasă de bază se poate deriva o clasă care, la rândul său, să servească drept clasă de bază pentru derivarea altora. Prin această succesiune se obține o **ierarhie de clase**.

Se pot defini clase deriveate care au la bază mai multe clase, înglobând proprietățile tuturor claselor de bază, procedeu ce poartă denumirea de **moștenire multiplă**.



## **Moștenirea în C++**

C++ permite moștenirea ceea ce înseamnă că putem deriva o clasă din altă clasă de bază sau din mai multe clase.

Sintaxa:

**class Clasa\_Derivata : [modificatori de acces] Clasa\_de\_Baza { .... } ;**

sau

**class Clasa\_Derivata : [modificatori de acces] Clasa\_de\_Baza1, [modificatori de acces] Clasa\_de\_Baza2, [modificatori de acces] Clasa\_de\_Baza3 .....**

Clasa de bază se mai numește clasă părinte sau superclasă, iar clasa derivată se mai numește subclasă sau clasă copil.



## Moștenirea în C++

Exemplu: moștenire

```
class X { int i;  
public:  
    X() { i = 0; }  
    void set(int ii) { i = ii; }  
    int read() const { return i; }  
    int permute() { return i = i * 47; }  
};  
class Y : public X {  
    int i; // Different from X's I  
public:  
    Y() { i = 0; }  
    int change() {  
        i = permute(); // Different name call  
        return i;  
    }  
    void set(int ii) {  
        i = ii;  
        sX::set(ii); // Same-name function call  
    }  
};
```

```
int main() {  
    cout << sizeof(X) << sizeof(Y);  
  
    Y D;  
    D.change(); // X function interface comes through:  
    D.read();  
    D.permute(); // Redefined functions hide base versions:  
    D.set(12);  
}
```



## Moștenirea în C++

### Moștenire vs. Compunere

Moștenirea este asemănătoare cu procesul de includere a obiectelor în obiecte (procedeu ce poartă denumirea de compunere), dar există câteva elemente caracteristice moștenirii:

- codul poate fi comun mai multor clase;
- clasele pot fi extinse, fără a recompila clasele originare;
- funcțiile ce utilizează obiecte din clasa de bază pot utiliza și obiecte din clasele derivate din această clasă.



## Moștenirea în C++

Modifierii de acces la moștenire

```
class A : public B { /* declarații */};
```

```
class A : protected B { /* declarații */};
```

```
class A : private B { /* declarații */};
```

Dacă modifierul de acces la moștenire este **public**, membrii din clasa de bază își păstrează tipul de acces și în derivată.

Dacă modifierul de acces la moștenire este **private**, toți membrii din clasa de bază vor avea tipul de acces “private” în derivată, indiferent de tipul avut în bază.

Dacă modifierul de acces la moștenire este **protected**, membrii “publici” din clasa de bază devin “protected” în clasa derivată, restul nu se modifică.



## Moștenirea în C++

```
class Baza { int a;  
public:  
    void f() {cout<<a;} // a este privat dar accesibil in clasa  
private:  
    void g() {cout<<a;}  
};  
  
class Derivata1 : protected Baza{  
public:  
    void h() {cout<<a;} // a este privat, inaccesibil
```

```
int main(){  
    Baza ob1;  
    // cout<<ob1.a; /* a este privat deci am  
    acces la el doar din Baza */  
}
```



## Moștenirea în C++

```
class Baza {  
public: void f() {cout<<"B";}  
};
```

```
class Derivata : public Baza{ };
```

```
int main()  
{  
    Derivata ob1;  
    ob1.f();  
}
```

Obs. Funcția f( ) este accesibilă din derivată;  
-modifierul de acces la moștenire este “public”, deci f( ) ramane “public” și în Derivata, deci este accesibil la apelul din main()

```
class Baza {  
public: void f() {cout<<"B";}  
};
```

```
class Derivata : private sau protected Baza{ };
```

```
int main()  
{  
    Derivata ob1;  
    ob1.f(); // inaccesibil  
}
```

-Dacă modifierul de acces la moștenire este “private”, f( ) devine private în Derivata, deci inaccesibilă în main.  
-Dacă modifierul de acces la moștenire este “protected”, f( ) devine protected1 în Derivata, deci inaccesibilă în main.



## Moștenirea în C++

### Moștenirea cu specificatorul “public”

```
class Base {  
protected:  
    int i;  
public:  
    Base() : i(7) {}  
};
```

```
class Derived1 : public Base {};  
  
class Derived2 : public Derived1 {  
public:  
    void set(int x) { i = x; }  
};
```

```
• int main() {  
    •     Derived2;  
    •     d.set(10);  
    • }
```

- dacă în baza avem zone “protected” ele sunt transmise și în derivată1,2 tot ca protected, deci i este accesibil în funcția set()



## Moștenirea în C++

### Moștenirea cu specificatorul “private”

- inclusă în limbaj pentru completitudine;
- este mai bine a se utiliza compunerea în locul moștenirii private;
- toți membrii private din clasa de bază sunt ascunși în clasa derivată, deci inaccesibili;
- toți membrii public și protected devin private, dar sunt accesibile în clasa derivată;
- un obiect obținut printr-o astfel de derivare se tratează diferit față de cel din clasa de bază, e similar cu definirea unui obiect de tip bază în interiorul clasei noi (fără moștenire).
- dacă în clasa de bază o componentă era public, iar moștenirea se face cu specificatorul private, se poate reveni la public utilizând:
  - *using Baza::nume\_componentă*



## 1. Moștenirea în C++

### Moștenirea cu specificatorul “private”

```
class Base {  
protected:  
    int i;  
  
public:  
    Base() : i(7) {}  
};
```

```
class Derived1 : private Base { };  
  
class Derived2 : public Derived1 {  
public:  
    void set(int x) { i = x; }  
};
```

```
• int main() {  
    • Derived2;  
    • d.set(10);  
    • }
```

- moștenirea derivată1 din baza (private) atunci zonele protected devin private în derivată1 și neaccesibile în derivată2.



## Moștenirea în C++

### Moștenirea cu specificatorul “private”

```
class Pet {  
public:  
    char eat() const { return 'a'; }  
    int speak() const { return 2; }  
    float sleep() const { return 3.0; }  
    float sleep( int ) const { return 4.0; }  
};
```

```
class Goldfish : Pet { // Private inheritance  
public:  
    using Pet::eat; // Name publicizes member  
    using Pet::sleep; // Both overloaded members exposed  
};
```

```
int main() {  
    Goldfish bob;  
    bob.eat();  
    bob.sleep();  
    bob.sleep(1);  
    //! bob.speak(); // Error: private member  
    //! funcțion  
}
```



## 1. Moștenirea în C++

### Moștenirea cu specificatorul “protected”

- secțiuni definite ca protected sunt similare ca definire cu private (sunt ascunse de restul programului), cu excepția claselor derivate;
- good practice: cel mai bine este ca variabilele de instanță să fie PRIVATE și funcții care le modifică să fie protected;
- Sintaxă: ***class derivată: protected baza {...};***
- toți membrii publici și protected din baza devin protected în derivată;
- nu se prea folosește, inclusă în limbaj pentru completitudine.



## 1. Moștenirea în C++

### Moștenirea cu specificatorul “protected”

```
class Base {  
protected:  
    int i;  
public:  
    Base() : i(7) {}  
};
```

```
class Derived1 : protected Base { };
```

```
class Derived2 : public Derived1 {  
public:  
    void set(int x) { i = x; }  
};
```

```
int main() {  
    Derived2 d;  
    d.set(10);  
}
```

- dacă în baza avem zone “protected” ele sunt transmise și în derivată1,2 tot ca protected, deci i este accesibil în funcția set()



## Moștenirea în C++

### *Inițializare de obiecte*

Foarte important în C++: garantarea inițializării corecte => trebuie să fie asigurată și la compunere și moștenire.

La crearea unui obiect, compilatorul trebuie să garanteze apelul TUTUROR sub-obiectelor.

**Problemă:** - cazul sub-obiectelor care nu au constructori implicați sau schimbarea valorii unui argument default în constructor.

- Initializarea constantelor?

**De ce?** - constructorul noii clase nu are permisiunea să acceseze datele **private** ale sub-obiectelor, deci nu le pot inițializa direct.

**Rezolvare:** - o sintaxă specială: **lista de inițializare pentru constructori**.



## Moștenirea în C++

Exemplu: lista de inițializare pentru constructori

```
class Bar {  
    int x;  
public:  
    Bar(int i) {x = i;}  
};
```

```
class MyType: public Bar {  
public:  
    MyType(int);  
};
```

MyType :: MyType (int i) : Bar (i) { ... }



## Moștenirea în C++

Exemplu 2: lista de inițializare pentru constructori

```
class Alta_clasa { int a;  
    public:  
        Alta_clasa(int i) {a = i;}  
};  
  
class Bar { int x;  
    public:  
        Bar(int i) {x = i;}  
};  
  
class MyType2: public Bar {  
    Alta_clasa m; // obiect m = subobiect in cadrul clasei MyType2  
    public:  
        MyType2(int);  
};
```

MyType2 :: MyType2 (int i) : Bar (i), m(i+1) { ... }



## Moștenirea în C++

### *Chestiune specială: “pseudo - constructori” pentru tipuri de bază*

- membrii de tipuri predefinite nu au constructori;
- soluție: C++ permite tratarea tipurilor predefinite asemănător unei clase cu o singură dată membră și care are un constructor parametrizat.



## Moștenirea în C++

```
class X {  
    int i;  
    float f;  
    char c;  
    char* s;  
public:  
    X() : i(7), f(1.4), c('x'), s("howdy") {}  
};
```

```
int main() {  
    X x;  
    int i(100); // Applied to ordinary definition  
    int* ip = new int(47);  
}
```



## Moștenirea în C++

*Exemplu: compunere și moștenire*

```
class A { int i;  
public:  
    A(int ii) : i(ii) {}  
    ~A() {}  
    void f() const {}  
};
```

```
class B { int i;  
public:  
    B(int ii) : i(ii) {}  
    ~B() {}  
    void f() const {}  
};
```

```
class C : public B {  
    A a;  
public:  
    C(int ii) : B(ii), a(ii) {}  
    ~C() {} // Calls ~A() and ~B()  
    void f() const  
    { // Redefinition  
        a.f();  
        B::f();  
    }  
};  
int main()  
{  
    C c(47);  
}
```



## Moștenirea în C++

### Constructorii clasei deriveate

Pentru crearea unui obiect al unei clase deriveate, **se creează inițial un obiect al clasei de bază prin apelul constructorului acesteia**, apoi se adaugă elementele specifice clasei deriveate prin apelul constructorului clasei deriveate.

Declarația obiectului derivat trebuie să conțină valorile de inițializare, **âtât pentru elementele specifice, cât și pentru obiectul clasei de bază**.

Această specificare se atașează la antetul funcției constructor a clasei deriveate.

În situația în care clasa de bază are definit un **constructor implicit** sau **constructor cu parametri implicați**, nu se impune specificarea parametrilor care se transferă către obiectul clasei de bază.



## Moștenirea în C++

Constructorii clasei deriveate

*Constructorul parametrizat*

```
class Forma {  
protected:  
    int h;  
public:  
    Forma(int a = 0) { h = a; }  
};
```

```
class Cerc: public Forma {  
protected:  
    float raza;  
public:  
    Cerc(int h=0, float r=0) : Forma(h) { raza = r; }  
};
```



## **Moștenirea în C++**

### **Constructorii clasei deriveate**

#### ***Constructorul de copiere***

Se pot distinge mai multe situații.

- 1) Dacă ambele clase, atât clasa derivată cât și clasa de bază, nu au definit un constructor de copiere, se apelează constructorul implicit creat de compilator. Copierea se face membru cu membru.
- 2) Dacă clasa de bază are constructorul de copiere definit, dar clasa derivată nu, pentru clasa derivată compilatorul creează un constructor implicit care apelează constructorul de copiere al clasei de bază.
- 3) Dacă se definește constructor de copiere pentru clasa derivată, acestuia **îi revine în totalitate sarcina transferării** valorilor corespunzătoare membrilor ce aparțin clasei de bază.



## Moștenirea în C++

### Constructorii clasei deriveate

#### *Constructorul de copiere*

```
class Forma {  
protected:    int h;  
public:  
    Forma(const Forma& ob)    {      h = ob.h;    }  
};
```

```
class Cerc: public Forma {  
protected:  
    float raza;  
public:  
    Cerc(const Cerc&ob):Forma(ob)    {      raza = ob.raza;    }  
};
```



## Moștenirea în C++

### Ordinea apelării constructorilor și destructorilor

- constructorii sunt apelați în ordinea definirii obiectelor ca membri ai clasei și în ordinea moștenirii:
- la fiecare nivel se apelează:
  - **întâi constructorul de la moștenire,**
  - apoi **constructorii din obiectele membru** în clasa respectivă (care sunt apelați în ordinea definirii)
  - și la final se merge pe următorul nivel în ordinea moștenirii;
- destructorii sunt executați în ordinea inversă a constructorilor



## Moștenirea în C++

### Ordinea apelării constructorilor și destructorilor

```
class A{  
public:  
    A(){cout<<"A ";}  
    ~A(){cout<<"~A ";}  
};
```

```
class C{  
public:  
    C(){cout<<"C ";}  
    ~C(){cout<<"~C ";}  
};
```

**Ordine:** *C B A D ~D ~A ~B ~C*

```
class B{  
public:  
    C ob;  
  
public:  
    B(){cout<<"B ";}  
    ~B(){cout<<"~B ";}  
};  
  
class D: public B{  
public:  
    A ob;  
  
public:  
    D(){cout<<"D ";}  
    ~D(){cout<<"~D ";}  
};  
  
int main() {  
    D s;  
}
```



## Moștenirea în C++

### Ordinea chemării constructorilor și destructorilor

```
#define CLASS(ID) class ID { \
public: \
ID(int) \
{ cout << #ID " constructor\n"; } \
~ID() \
{ cout << #ID " destructor\n"; } \
};
```

```
CLASS(Base1);  
CLASS(Member1);  
CLASS(Member2);  
CLASS(Member3);  
CLASS(Member4);
```

```
class Derived1 : public Base1 {  
    Member1 m1;  
    Member2 m2;  
public:  
    Derived1(int) : m2(1), m1(2), Base1(3)  
    { cout << "Derived1 constructor\n"; }  
    ~Derived1() { cout << "Derived1 destructor\n"; }  
};  
  
class Derived2 : public Derived1 {  
    Member3 m3;  
    Member4 m4;  
public:  
    Derived2() : m3(1), Derived1(2), m4(3)  
    { cout << "Derived2 constructor\n"; }  
    ~Derived2() { cout << "Derived2 destructor\n"; }  
};  
int main() { Derived2 d2; }
```



## Moștenirea în C++

### Ordinea chemării constructorilor și destructorilor

```
#define CLASS(ID) class ID { \
public: \
    ID(int)
    { cout << #ID " constructor\n"; } \
    ~ID()
    { cout << #ID " destructor\n"; } \
};

CLASS(Base1);
CLASS(Member1);
CLASS(Member2);
CLASS(Member3);
CLASS(Member4);
```

Se va afisa:

Base1 constructor  
Member1 constructor  
Member2 constructor  
Derived1 constructor  
Member3 constructor  
Member4 constructor  
Derived2 constructor  
Derived2 destructor  
Member4 destructor  
Member3 destructor  
Derived1 destructor  
Member2 destructor  
Member1 destructor  
Base1 destructor



## Moștenirea în C++

### *Operatorul =*

```
class Forma {  
protected:  
    int h;  
public:  
    Forma& operator=(const Forma& ob);  
};  
  
class Cerc: public Forma {  
protected:  
    float raza;  
public:  
    Cerc& operator=(const Cerc& ob);  
};
```

```
Forma& Forma::operator=(const Forma& ob) {  
    if (this != &ob) { h = ob.h; }  
    return *this;  
}  
  
Cerc& Cerc::operator=(const Cerc& ob) {  
    if (this != &ob)  
    {  
        this->Forma::operator=(ob);  
    }  
    return *this;  
}
```



## Moștenirea în C++

### *Redefinirea funcțiilor membre*

```
class Forma {  
protected:  
    int h;  
public:  
    void afis() { cout<<h<<" "; }  
};
```

Clasa derivată are acces la toți membrii cu acces **protected** sau **public** ai clasei de bază.

```
class Cerc: public Forma {  
protected:  
    float raza;  
public:  
    void afis() {  
        Forma::afis();  
        cout<<raza; }  
};
```

Este permisă **supradefinirea** funcțiilor membre clasei de bază cu funcții membre ale clasei deriveate.



## Moștenirea în C++

### *Compatibilitatea între o clasă derivată și clasa de bază. Conversii de tip*

Deoarece clasa derivată moștenește proprietățile clasei de bază, între tipul clasă derivată și tipul clasă de bază se admite o anumită compatibilitate.

Compatibilitatea este valabilă numai pentru clase **derivate cu acces public** la clasa de bază și numai în sensul de la clasa derivată spre cea de bază, nu și invers.

Compatibilitatea se manifestă sub forma unor **conversii implicate de tip**:

- dintr-un obiect derivat într-un obiect de bază;
- dintr-un pointer sau referință la un obiect din clasa derivată într-un pointer sau referință la un obiect al clasei de bază.



# **Perspective**

## **Cursul 7:**

1. Proiectarea descendenta a claselor.
  - Redefinirea membrilor unei clase de bază într-o clasă derivată.
  - Mostenirea și funcțiile statice
  - Declarații de acces.
2. Parametrizarea metodelor (polimorfism la execuție).
  - Funcții virtuale în C++. Clase abstracte.
  - Destructori virtuali.



# **Programare orientată pe obiecte**

## **- suport de curs -**

**Andrei Păun**  
**Anca Dobrovăț**

**An universitar 2021 – 2022**  
**Semestrul II**  
**Seriile 13, 14 și 15**

**Curs 7**



## Agenda cursului

1. Proiectarea descendentală a claselor. Moștenirea în C++ (recapitulare și completări la cursul 6)
  - Controlul accesului la clasa de bază.
  - Constructori, destrutori și moștenire.
  - Redefinirea membrilor unei clase de bază într-o clasă derivată.
  - Declarații de acces.
2. Polimorfism la execuție prin funcții virtuale în C++.
  - Parametrizarea metodelor (polimorfism la execuție).
  - Funcții virtuale în C++.
  - Clase abstracte.
  - Overloading pe funcții virtuale
  - Destructori și virtualizare



## 1. Moștenirea în C++

- important în C++ - reutilizare de cod;
- reutilizare de cod prin creare de noi clase (nu se dorește crearea de clase de la zero);
- 2 modalități (compunere și moștenire);
- “compunere” - noua clasă “este compusă” din obiecte reprezentând instanțe ale claselor deja create;
- “moștenire” - se creează un nou tip al unei clase deja existente.



## 1. Moștenirea în C++

*Exemple: compozиie și moștenire*

```
class A { int i;  
public:  
    A(int ii) : i(ii) {}  
    ~A() {}  
    void f() const {}  
};  
  
class B { int i;  
public:  
    B(int ii) : i(ii) {}  
    ~B() {}  
    void f() const {}  
};
```

```
class C : public B {  
    A a;  
public:  
    C(int ii) : B(ii), a(ii) {}  
    ~C() {} // Calls ~A() and ~B()  
    void f() const  
    { // Redefinition  
        a.f();  
        B::f();  
    }  
};  
int main()  
{  
    C c(47);  
}
```



## 1. Moștenirea în C++

### *Inițializare de obiecte*

Foarte important în C++: garantarea inițializării corecte => trebuie să fie asigurată și la compozиție și moștenire.

La crearea unui obiect, compilatorul trebuie să garanteze apelul TUTUROR subobiectelor.

**Problema:** - cazul subobiectelor care nu au constructori implicați sau schimbarea valorii unui argument default în constructor.

**De ce?** - constructorul noii clase nu are permisiunea să acceseze datele private ale subobiectelor, deci nu le pot inițializa direct.

**Rezolvare:** - o sintaxă specială: *listă de inițializare pentru constructori*.



## 1. Moștenirea în C++

*Exemplu: lista de initializare pentru constructori*

```
class Alta_clasa { int a;  
    public:  
        Alta_clasa(int i) {a = i;}  
};  
class Bar { int x;  
    public:  
        Bar(int i) {x = i;}  
};  
class MyType2: public Bar {  
    Alta_clasa m; // obiect m = subobiect în cadrul clasei MyType2  
    public:  
        MyType2(int);  
};
```

MyType2 :: MyType2 (int i) : Bar (i), m(i+1) { ... }



## 1. Moștenirea în C++

### ***Constructorii clasei derivate***

Pentru crearea unui obiect al unei clase derivate, se creează inițial un **obiect al clasei de bază** prin apelul constructorului acesteia, apoi se adaugă elementele specifice clasei derivate prin apelul constructorului clasei derivate.

Declarația obiectului derivat trebuie să conțină valorile de inițializare, **âtât pentru elementele specifice, cât și pentru obiectul clasei de bază**.

Această specificare se atașează la antetul funcției constructor a clasei derivate.

În situația în care clasele de bază au definit **constructor implicit** sau **constructor cu parametri implicați**, nu se impune specificarea parametrilor care se transferă către obiectul clasei de bază.



## 1. Moștenirea în C++

*Constructorii clasei derivate*

*Constructorul de copiere*

Se pot distinge mai multe situații.

- 1) Dacă ambele clase, atât clasa derivată cât și clasa de bază, nu au definit constructor de copiere, se apelează constructorul implicit creat de compilator. Copierea se face membru cu membru.
- 2) Dacă clasa de bază are constructorul de copiere definit, dar clasa derivată nu, pentru clasa derivată compilatorul creează un constructor implicit care apelează constructorul de copiere al clasei de bază. (poate fi considerata un caz particular al primei situații, deoarece și partea de bază poate fi privită ca un fel de membru, iar la copiere se apelează cc pentru fiecare membru).
- 3) Dacă se definește constructor de copiere pentru clasa derivată, acestuia îi revine în totalitate sarcina transferării valorilor corespunzătoare membrilor ce aparțin clasei de bază.



## 1. Moștenirea în C++

### *Redefinirea funcțiilor membre*

Clasa derivată are acces la toți membrii cu acces **protected** sau **public** ai clasei de bază.

Este permisă supraredefinirea funcțiilor membre clasei de bază cu funcții membre ale clasei derivate.

- 2 modalități de a redefini o funcție membră:

- **cu același antet ca în clasa de bază** (“redefining” - în cazul funcțiilor oarecare / “overloading” - în cazul funcțiilor virtuale);
- **cu schimbarea listei de argumente sau a tipului returnat.**



## 1. Moștenirea în C++

### *Redefinirea funcțiilor membre*

**Exemplu:** - pastrarea antetului/tipului returnat

```
class Baza {  
public:  
    void afis() { cout<<"Baza\n"; }  
};
```

```
class Derivata : public Baza {  
public:  
    void afis() { Baza::afis(); cout<<" si Derivata\n"; }  
};
```

```
int main() {  
    Derivata d;  
    d.afis(); // se afiseaza "Baza si Derivata"  
}
```



## 1. Moștenirea în C++

### *Redefinirea funcțiilor membre*

**Exemplu:** - nepastrarea antetului/tipului returnat

```
class Baza {  
public:  
    void afis() { cout<<"Baza\n"; }  
};
```

```
class Derivata : public Baza {  
public:  
    void afis (int x) {  
        Baza::afis();  
        cout<<"si Derivata\n"; }  
};
```

```
int main() {  
    Derivata d;  
    d.afis(); //nu există Derivata::afis()  
    d.afis(3); }
```

*Obs: la redefinirea unei funcții din clasa de bază, toate celelalte versiuni sunt automat ascunse!*



## 1. Moștenirea în C++

### *Redefinirea funcțiilor membre*

Care este efectul codului urmator?

```
class Base {  
public:  
    int f() const { cout << "Base::f()\n"; return 1; }  
    int f(string) const { return 1; }  
    void g() {}  
};
```

```
class Derived1 : public Base {  
public:    void g() const {}  
};
```

```
class Derived2 : public Base {  
public:  
    // Redefinition:  
    int f() const { cout << "Derived2::f()\n"; return 2; }  
};
```

```
int main() {  
    string s("hello");
```

```
    Derived1 d1;  
    int x = d1.f();  
    cout << d1.f(s);
```

```
    Derived2 d2;  
    x = d2.f();  
    //! d2.f(s); // string version hidden
```



## 1. Moștenirea în C++

### *Redefinirea funcțiilor membre*

Care este efectul codului urmator?

```
class Base {  
public:  
    int f() const { cout << "Base::f()\n"; return 1; }  
    int f(string) const { return 1; }  
    void g() {}  
};  
class Derived3 : public Base {  
public:  
    // Change return type:  
    void f() const { cout << "Derived3::f()\n"; }  
};  
class Derived4 : public Base {  
public:  
    // Change argument list:  
    int f(int) const {  
        cout << "Derived4::f()\n";  
        return 4;  
    }  
};
```

```
int main() {  
  
    Derived3 d3;  
    //! x = d3.f(); // return int version hidden  
  
    Derived4 d4;  
    //! x = d4.f(); // f() version hidden  
    x = d4.f(1);  
}
```



## 1. Moștenirea în C++

### *Redefinirea funcțiilor membre*

**Obs:**

Schimbarea interfeței clasei de bază prin modificarea tipului returnat sau a signaturii unei funcții, înseamnă, de fapt, utilizarea clasei în alt mod.

Scopul principal al moștenirii: polimorfismul.

Schimbarea signaturii sau a tipului returnat = schimbarea interfeței = contravine exact polimorfismului (un aspect esențial este păstrarea interfeței clasei de bază).



## 1. Moștenirea în C++

*Redefinirea funcțiilor membre*

*Particularități la funcții*

constructorii și destructorii nu sunt moșteniți (se redefiniesc noi constr. și destr. pentru clasa derivată)

similar operatorul = (un fel de constructor)



## 1. Moștenirea în C++

*Funcții care nu se moștenesc automat*

**Operatorul =**

```
class Forma {  
protected:    int h;  
public:  
    Forma& operator=(const Forma& ob)  {  
        if (this!=&ob)      {          h = ob.h;      }  
        return *this;    }  
};
```

```
class Cerc: public Forma {  
protected:  
    float raza;  
public:  
    Cerc& operator=(const Cerc& ob)  {  
        if (this != &ob)      {          this->Forma::operator=(ob);      }  
        return *this;    }  
};
```



## 1. Moștenirea în C++

### *Moștenirea și funcțiile statice*

Funcțiile membre statice se comportă exact ca și funcțiile nemembre:  
Se moștenesc în clasa derivată.

Redefinirea unei funcții membre statice duce la ascunderea celorlalte supraîncărcări.

Schimbarea signaturii unei funcții din clasa de bază duce la ascunderea celorlalte versiuni ale funcției.

**Dar: O funcție membră statică nu poate fi virtuală.**



## 1. Moștenirea în C++

### *Moștenirea și funcțiile statice*

```
class Base {  
public:  
    static void f() { cout << "Base::f()\n"; }  
    static void g() { cout << "Base::f()\n"; }  
};  
  
class Derived : public Base {  
public: // Change argument list:  
    static void f(int x) { cout << "Derived::f(x)\n"; }  
};  
  
int main() {  
    int x;  
    Derived::f(); // ascunsa de supradefinirea f(x)  
    Derived::f(x);  
    Derived::g();  
}
```



## 1. Moștenirea în C++

### *Modificatorii de acces la moștenire*

class A : **public** B { /\* declarații \*/};

class A : **protected** B { /\* declarații \*/};

class A : **private** B { /\* declarații \*/};

Dacă modificadorul de acces la moștenire este **public**, membrii din clasa de bază își păstrează tipul de acces și în derivată.

Dacă modificadorul de acces la moștenire este **private**, toți membrii din clasa de bază vor avea tipul de acces “private” în derivată, indiferent de tipul avut în bază.

Dacă modificadorul de acces la moștenire este **protected**, membrii “publici” din clasa de bază devin “protected” în clasa derivată, restul nu se modifică.



## 1. Moștenirea în C++

### *Moștenire multiplă (MM)*

- putine limbaje au MM;
- moștenirea multiplă e complicată: ambiguitate LA MOSTENIREA IN ROMB / IN DIAMANT;
- nu e nevoie de MM (se simulează cu moștenire simplă);
- se moșteneste în același timp din mai multe clase;

*Sintaxa:*

*class Clasa\_Derivată : [modificatori de acces] Clasa\_de\_Bază1,  
[modificatori de acces] Clasa\_de\_Bază2, [modificatori de acces]  
Clasa\_de\_Bază3 .....*



## 1. Moștenirea în C++

### *Moștenire multiplă (MM)*

*Exemplu:*

```
class Imprimanta { };
class Scaner { };
class Multifuncționala: public Imprimanta, public Scaner { };
```

*Ce ar putea crea probleme în cazul urmator?*

```
class Baza{ };
class Derivata_1: public Baza{ };
class Derivata_2: public Baza{ };
class Derivata_3: public Derivata_1, public Derivata_2 { };
```

In **Derivata\_3** avem de două ori variabilele din **Baza**!!



## 1. Moștenirea în C++

### *Moștenire multiplă : ambiguități (problema diamantului)*

```
class base { public:    int i; };
class derived1 : public base { public:    int j; };
class derived2 : public base { public:    int k; };
class derived3 : public derived1, public derived2 {public:    int sum; };

int main() {
    derived3 ob;
    ob.i = 10; // this is ambiguous, which i      // expl ob.derived1::i
    ob.j = 20;
    ob.k = 30;
    ob.sum = ob.i + ob.j + ob.k; // i ambiguous here, too
    cout << ob.i << " "; // also ambiguous, which i?
    cout << ob.j << " " << ob.k << " ";
    cout << ob.sum;
    return 0;
}
```



## 1. Moștenirea în C++

### *Moștenire multiplă (MM)*

- dar dacă avem nevoie doar de o copie lui i?
- nu vrem să consumăm spațiu în memorie;
- **folosim moștenire virtuală:**

```
class base { public: int i; };
class derived1 : virtual public base { public: int j; };
class derived2 : virtual public base { public: int k; };
class derived3 : public derived1, public derived2 {public: int sum; };
```

- Dacă avem moștenire de două sau mai multe ori dintr-o clasă de bază (fiecare moștenire trebuie să fie virtuală) atunci compilatorul alocă spațiu pentru o singură copie;
- În clasele derived1 și 2 moștenirea e la fel ca mai înainte (niciun efect pentru virtual în acel caz)



## 2. Polimorfismul la execuție prin funcții virtuale

### *Funcții virtuale*

Funcțiile virtuale și felul lor de folosire: componentă **IMPORTANTĂ** a limbajului OOP.

Folosit pentru polimorfism la execuție ---> cod mai bine organizat cu polimorfism.

Codul poate “crește” fără schimbări semnificative: programe extensibile.

Funcțiile virtuale sunt definite în bază și redefinite în clasa derivată.

Pointer de tip bază care arată către obiect de tip derivat și cheamă o funcție virtuală în bază și redefinită în clasa derivată execută ***Funcția din clasa derivată***.

Poate fi vazuta ca exemplu de separare dintre interfata si implementare.



## 2. Polimorfismul la execuție prin funcții virtuale

### *Decuplare în privința tipurilor*

**Upcasting** - Tipul derivat poate lua locul tipului de bază (foarte important pentru procesarea mai multor tipuri prin același cod).

Funcții virtuale: ne lasă să chemăm funcțiile pentru tipul derivat.

Problemă: apel la funcție prin pointer (tipul pointerului ne da funcția apelată).



## 2. Polimorfismul la execuție prin funcții virtuale

```
enum note { middleC, Csharp, Eflat }; // Etc.
```

```
class Instrument { public:  
    void play(note) const {  
        cout << "Instrument::play" << endl; }  
};
```

```
class Wind : public Instrument {  
public: // Redefine interface function:  
    void play(note) const {  
        cout << "Wind::play" << endl; }  
};
```

```
void tune(Instrument& i) { i.play(middleC); }  
  
int main() {  
    Wind flute;  
    tune(flute); // Upcasting ===> se afiseaza Instrument::play  
}
```



## 2. Polimorfismul la execuție prin funcții virtuale

In C ---> early binding la apel de funcții - se face la compilare.

In C++ ---> putem defini late binding prin funcții virtuale (late, dynamic, runtime binding) - se face apel de funcție bazat pe tipul obiectului, la rulare (nu se poate face la compilare).

***Late binding ==> prin pointeri!***

Late binding pentru o funcție: se scrie virtual înainte de definirea funcției.

Pentru clasa de bază: nu se schimbă nimic!

Pentru clasa derivată: late binding însemenă că un obiect derivat folosit în locul obiectului de bază își va folosi funcția sa, nu cea din bază (din cauză de late binding).

***Utilitate: putem extinde codul precedent fără schimbări în codul deja scris.***



## 2. Polimorfismul la execuție prin funcții virtuale

```
enum note { middleC, Csharp, Eflat }; // Etc.
```

```
class Instrument { public:  
    virtual void play(note) const {  
        cout << "Instrument::play" << endl; }  
};
```

```
class Wind : public Instrument {  
public: // Redefine interface funcțion:  
    void play(note) const {  
        cout << "Wind::play" << endl; }  
};
```

```
void tune(Instrument& i) { i.play(middleC); }
```

```
int main() {  
    Wind flute;  
    tune(flute); // se afiseaza Wind::play  
}
```



```
#include <iostream>
using namespace std;
enum note { middleC, Csharp, Eflat }; // Etc.

class Instrument {
public:
    virtual void play(note) const {
        cout << "Instrument::play" << endl;
    }
};

// Wind objects are Instruments
// because they have the same interface:
class Wind : public Instrument {
public:
    // Override interface function:
    void play(note) const {
        cout << "Wind::play" << endl;
    }
};

void tune(Instrument& i) {
    // ...
    i.play(middleC);
}
```

```
class Percussion : public Instrument {
public:
    void play(note) const {
        cout << "Percussion::play" << endl; } };
class Stringed : public Instrument {
public:
    void play(note) const {
        cout << "Stringed::play" << endl; } };
class Brass : public Wind {
public:
    void play(note) const {
        cout << "Brass::play" << endl; } };
class Woodwind : public Wind {
public:
    void play(note) const {
        cout << "Woodwind::play" << endl; } };

int main() {
    Wind flute;
    Percussion drum;
    Stringed violin;
    Brass flugelhorn;
    Woodwind recorder;
    tune(flute); tune(flugehorn); tune(violin);
}
```



## 2. Polimorfismul la execuție prin funcții virtuale

### *Cum se face late binding*

Tipul obiectului este ținut în obiect pentru clasele cu funcții virtuale.

Late binding se face (uzual) cu o tabelă de pointeri: vptr către funcții.

În tabelă sunt adresele funcțiilor clasei respective (funcțiile virtuale sunt din clasa, celelalte pot fi moștenite, etc.).

Fiecare obiect din clasă are pointerul acesta în componentă.

La apel de funcție membru se merge la obiect, se apelează funcția prin vptr.

Vptr este inițializat în constructor (automat).



## 2. Polimorfismul la execuție prin funcții virtuale

### *Cum se face late binding*

```
class NoVirtual { int a;  
public:  
    void x() const {}  
    int i() const { return 1; } };
```

```
class OneVirtual { int a;  
public:  
    virtual void x() const {}  
    int i() const { return 1; } };
```

```
class TwoVirtuals { int a;  
public:  
    virtual void x() const {}  
    virtual int i() const { return 1; } };
```

```
int main()  
{  
    cout << "int: " << sizeof(int) << endl;  
    cout << "NoVirtual: "  
        << sizeof(NoVirtual) << endl;  
    cout << "void* : " << sizeof(void*) << endl;  
    cout << "OneVirtual: "  
        << sizeof(OneVirtual) << endl;  
    cout << "TwoVirtuals: "  
        << sizeof(TwoVirtuals) << endl;  
}
```



## 2. Polimorfismul la execuție prin funcții virtuale

### *Cum se face late binding*

```
class Pet { public:  
    virtual string speak() const { return " "; } };
```

```
class Dog : public Pet { public:  
    string speak() const { return "Bark!"; } };
```

```
int main()  
{  
    Dog ralph;  
    Pet* p1 = &ralph;  
    Pet& p2 = ralph;  
    Pet p3;  
    // Late binding for both:  
    cout << "p1->speak() = " << p1->speak() << endl;  
    cout << "p2.speak() = " << p2.speak() << endl;  
    // Early binding (probably):  
    cout << "p3.speak() = " << p3.speak() << endl;  
}
```



## 2. Polimorfismul la execuție prin funcții virtuale

Dacă funcțiile virtuale sunt aşa de importante de ce nu sunt toate funcțiile definite virtuale (din oficiu)?

Deoarece “costă” în viteza programului.

În Java sunt “default”, dar Java e mai lent.

Nu mai putem avea funcții inline (ne trebuie adresa funcției pentru **vptr**).



## 2. Polimorfismul la execuție prin funcții virtuale

### *Clase abstracte și funcții virtuale pure*

Clasă abstractă = clasă care are cel puțin o funcție virtuală PURĂ

Necesitate: clase care dau doar interfață (nu vrem obiecte din clasă abstractă ci upcasting la ea).

Eroare la instantierea unei clase abstracte (nu se pot defini obiecte de tipul respectiv).

Permisă utilizarea de pointeri și referințe către clasă abstractă (pentru upcasting).

Nu pot fi trimise către funcții (prin valoare).



## 2. Polimorfismul la execuție prin funcții virtuale

### *Funcții virtuale pure*

Sintaxa: **virtual tip\_returnat nume\_funcție(lista\_parametri) =0;**

Ex: **virtual int pura(int i)=0;**

Obs: La moștenire, dacă în clasa derivată nu se definește funcția pură, clasa derivată este și ea clasă abstractă ---> nu trebuie definită funcție care nu se execută niciodată

**UTILIZARE IMPORTANTĂ:** prevenirea “object slicing”.



## 2. Polimorfismul la execuție prin funcții virtuale

### *Clase abstracte și funcții virtuale pure*

```
class Pet { string pname;
```

```
public:
```

```
Pet(const string& name) : pname(name) {}
```

```
virtual string name() const { return pname; }
```

```
virtual string description() const {
```

```
    return "This is " + pname;
```

```
}
```

```
};
```

```
class Dog : public Pet { string favoriteActivity;
```

```
public:
```

```
Dog(const string& name, const string& activity)
```

```
    : Pet(name), favoriteActivity(activity) {}
```

```
string description() const {
```

```
    return Pet::name() + " likes to " + favoriteActivity;
```

```
}
```

```
};
```

```
void describe(Pet p) { // Slicing  
    cout << p.description() << endl;  
}
```

```
int main()
```

```
Pet p("Alfred");
```

```
Dog d("Fluffy", "sleep");
```

```
describe(p);
```

```
describe(d);
```

```
}
```



## 2. Polimorfismul la execuție prin funcții virtuale

### *Overload pe funcții virtuale*

Obs. Nu e posibil overload prin schimbarea tipului param. de întoarcere  
(e posibil pentru ne-virtuale)

De ce. Pentru că se vrea să se garanteze că se poate chama baza prin apelul respectiv.

Excepție: pointer către bază întors în bază, pointer către derivată în derivată



## 2. Polimorfismul la execuție prin funcții virtuale

### *Overload pe funcții virtuale*

```
class Base {  
public:  
    virtual int f() const {  
        cout << "Base::f()\n"; return 1; }  
    virtual void f(string) const {}  
    virtual void g() const {}  
};
```

```
class Derived1 : public Base {public:  
    void g() const {}  
};
```

```
class Derived2 : public Base {public:  
    // Overriding a virtual function:  
    int f() const { cout << "Derived2::f()\n";  
        return 2; }  
};
```

```
int main() {  
    string s("hello");  
    Derived1 d1;  
    int x = d1.f();  
    d1.f(s);  
    Derived2 d2;  
    x = d2.f();  
    //! d2.f(s); // string version hidden  
}
```



## 2. Polimorfismul la execuție prin funcții virtuale

### *Overload pe funcții virtuale*

```
class Base {  
public:  
    virtual int f() const {  
        cout << "Base::f()\n"; return 1; }  
    virtual void f(string) const {}  
    virtual void g() const {}  
};
```

```
class Derived3 : public Base {public:  
//! void f() const{ cout << "Derived3::f()\n";}};
```

```
class Derived4 : public Base {public:  
// Change argument list:  
    int f(int) const  
    { cout << "Derived4::f()\n"; return 4; }  
};
```

```
int main() {  
    string s("hello");  
    Derived4 d4;  
    x = d4.f(1);  
    //! x = d4.f(); // f() version hidden  
    //! d4.f(s); // string version hidden  
    Base& br = d4; // Upcast  
    //! br.f(1); // Derived version  
    unavailable  
    br.f();  
    br.f(s); // Base version available  
    return 0;  
}
```



## 2. Polimorfismul la execuție prin funcții virtuale

### *Constructori și virtualizare*

**Obs.** NU putem avea constructori virtuali.

În general pentru funcțiile virtuale se utilizează late binding, dar în utilizarea funcțiilor virtuale în constructori, varianta locală este folosită (early binding)

De ce?

Pentru că funcția virtuală din clasa derivată ar putea crede că obiectul e inițializat deja

Pentru că la nivel de compilator în acel moment doar VPTR local este cunoscut



## 2. Polimorfismul la execuție prin funcții virtuale

### *Destructori și virtualizare*

Este ușual să se întâlnească.

Se cheamă în ordine inversă decât constructorii.

*Dacă vrem să eliminăm porțiuni alocate dinamic și pentru clasa derivată dar facem upcasting trebuie să folosim destructori virtuali.*



## 2. Polimorfismul la execuție prin funcții virtuale

### *Destructori și virtualizare*

```
class Base1 {public: ~Base1() { cout << "~Base1()\n"; } };
```

```
class Derived1 : public Base1 {public: ~Derived1() { cout << "~Derived1()\n"; } };
```

```
class Base2 {public:  
    virtual ~Base2() { cout << "~Base2()\n"; }  
};
```

```
class Derived2 : public Base2 {public: ~Derived2() { cout << "~Derived2()\n"; } };
```

```
int main()  
{  
    Base1* bp = new Derived1;  
    delete bp; // Afis: ~Base1()  
    Base2* b2p = new Derived2;  
    delete b2p; // Afis: ~Derived2() ~Base2()  
}
```



## 2. Polimorfismul la execuție prin funcții virtuale

### *Destructori virtuali puri*

**Utilizare:** recomandat să fie utilizat dacă mai sunt și alte funcții virtuale.

**Restricție:** trebuie definiți în clasă (chiar dacă este abstractă).

La moștenire nu mai trebuie redefiniti (se construiește un destructor din oficiu)

De ce? Pentru a preveni instantierea clasei.

**Obs.** Nu are nici un efect dacă nu se face upcasting.

```
class AbstractBase {  
public:  
    virtual ~AbstractBase() = 0;  
};
```

```
AbstractBase::~AbstractBase()
```

```
class Derived : public AbstractBase {};  
// No overriding of destructor necessary?  
int main() { Derived d; }
```



## 2. Polimorfismul la execuție prin funcții virtuale

### *Funcții virtuale în destructori*

La apel de funcție virtuală din funcții normale se apelează conform VPTR

În destructori se face early binding! (apeluri locale)

De ce? Pentru că acel apel poate să se bazeze pe porțiuni deja distruse din obiect

**class** Base { **public:**

**virtual** ~Base() { cout << "~Base1()\n"; this->f(); }

**virtual void** f() { cout << "Base::f()\n"; }

};

**class** Derived : **public** Base { **public:**

    ~Derived() { cout << "~Derived()\n"; }

**void** f() { cout << "Derived::f()\n"; }

};

**int** main() {

    Base\* bp = **new** Derived;

**delete** bp; // Afis: ~Derived() ~Base1() Base::f()

}



## 2. Polimorfismul la execuție prin funcții virtuale

### *Downcasting*

Folosit în ierarhii polimorfice (cu funcții virtuale).

**Problema:** upcasting e sigur pentru că respectivele funcții trebuie să fie definite în bază, downcasting e problematic.

Explicit cast prin: **dynamic\_cast**

*Dacă știm cu siguranță tipul obiectului putem folosi “static\_cast”.*

**Static\_cast** întoarce pointer către obiectul care satisfac cerințele sau 0.

Folosește tabelele VTABLE pentru determinarea tipului.



## 2. Polimorfismul la execuție prin funcții virtuale

### *Downcasting*

```
class Pet { public: virtual ~Pet(){};  
class Dog : public Pet {};  
class Cat : public Pet {};  
  
int main() {  
    Pet* b = new Cat; // Upcast  
    Dog* d1 = dynamic_cast<Dog*>(b); // Afis - 0; Try to cast it to Dog*:  
    Cat* d2 = dynamic_cast<Cat*>(b); // Try to cast it to Cat*:  
    // b si d2 retin aceeasi adresa  
    cout << "d1 = " << d1 << endl;  
    cout << "d2 = " << d2 << endl;  
    cout << "b = " << b << endl;  
}
```



## 2. Polimorfismul la execuție prin funcții virtuale

### Downcasting

```
class Shape { public: virtual ~Shape() {}; }
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
    Circle c;
    Shape* s = &c; // Upcast: normal and OK
    // More explicit but unnecessary:
    s = static_cast<Shape*>(&c);
    // (Since upcasting is such a safe and common
    // operation, the cast becomes cluttering)
    Circle* cp = 0;
    Square* sp = 0;
```

```
// Static Navigation of class hierarchies
// requires extra type information:
if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
if(cp != 0)
    cout << "It's a circle!" << endl;
if(sp != 0)
    cout << "It's a square!" << endl;
// Static navigation is ONLY an efficiency
// hack;
// dynamic_cast is always safer. However:
// Other* op = static_cast<Other*>(s);
// Conveniently gives an error message,
// while
Other* op2 = (Other*)s;
// does not
}
```



## Perspective

Cursul 8:

Operatorii de cast

Downcasting si Upcasting

Tratarea exceptiilor



# Programare orientată pe obiecte

## - suport de curs -

Click to add text

**Andrei Păun**  
**Anca Dobrovăț**

**An universitar 2021 – 2022**  
**Semestrul II**  
**Seriile 13, 14 și 15**

**Curs 8**



## **Agenda cursului**

1. Recapitulare – Polimorfism la executie
2. Operatorii de cast
3. Downcasting si upcasting
4. Tratarea exceptiilor



## Polimorfismul la execuție prin funcții virtuale

### *Funcții virtuale*

Funcțiile virtuale și felul lor de folosire: componentă **IMPORTANTĂ** a limbajului OOP.

Folosit pentru polimorfism la execuție ---> cod mai bine organizat cu polimorfism.

Codul poate “crește” fără schimbări semnificative: programe extensibile.

Funcțiile virtuale sunt definite în bază și redefinite în clasa derivată.

Pointer de tip bază care arată către obiect de tip derivat și cheamă o funcție virtuală în bază și redefinită în clasa derivată execută ***Funcția din clasa derivată***.

Poate fi vazuta ca exemplu de separare dintre interfata si implementare.



## Polimorfismul la execuție prin funcții virtuale

In C ---> early binding la apel de funcții - se face la compilare.

In C++ ---> putem defini late binding prin funcții virtuale (late, dynamic, runtime binding) - se face apel de funcție bazat pe tipul obiectului, la rulare (nu se poate face la compilare).

***Late binding ==> prin pointeri!***

Late binding pentru o funcție: se scrie virtual înainte de definirea funcției.

Pentru clasa de bază: nu se schimbă nimic!

Pentru clasa derivată: late binding însemenă că un obiect derivat folosit în locul obiectului de bază își va folosi funcția sa, nu cea din bază (din cauză de late binding).

***Utilitate: putem extinde codul precedent fără schimbări în codul deja scris.***



## Polimorfismul la execuție prin funcții virtuale

### *Cum se face late binding*

Tipul obiectului este ținut în obiect pentru clasele cu funcții virtuale.

Late binding se face (uzual) cu o tabelă de pointeri: vptr către funcții.

În tabelă sunt adresele funcțiilor clasei respective (funcțiile virtuale sunt din clasa, celelalte pot fi moștenite, etc.).

Fiecare obiect din clasă are pointerul acesta în componentă.

La apel de funcție membru se merge la obiect, se apelează funcția prin vptr.

Vptr este inițializat în constructor (automat).



## Polimorfismul la execuție prin funcții virtuale

### *Cum se face late binding*

```
class NoVirtual { int a;  
public:
```

```
    void x() const {}  
    int i() const { return 1; } };
```

```
class OneVirtual { int a;  
public:
```

```
    virtual void x() const {}  
    int i() const { return 1; } };
```

```
class TwoVirtuals { int a;  
public:
```

```
    virtual void x() const {}  
    virtual int i() const { return 1; } };
```

```
int main() {  
    cout << "int: " << sizeof(int) << endl;  
    cout << "NoVirtual: "  
        << sizeof(NoVirtual) << endl;  
    cout << "void* : " << sizeof(void*) << endl;  
    cout << "OneVirtual: "  
        << sizeof(OneVirtual) << endl;  
    cout << "TwoVirtuals: "  
        << sizeof(TwoVirtuals) << endl;  
}
```



## Polimorfismul la execuție prin funcții virtuale

### *Cum se face late binding*

```
class Pet { public:  
    virtual string speak() const { return " "; } };  
  
class Dog : public Pet { public:  
    string speak() const { return "Bark!"; } };  
  
int main()  
{  
    Dog ralph;  
    Pet* p1 = &ralph;  
    Pet& p2 = ralph;  
    Pet p3;  
    // Late binding for both:  
    cout << "p1->speak() = " << p1->speak() << endl;  
    cout << "p2.speak() = " << p2.speak() << endl;  
    // Early binding (probably):  
    cout << "p3.speak() = " << p3.speak() << endl;  
}
```



## Polimorfismul la execuție prin funcții virtuale

### *Clase abstracte și funcții virtuale pure*

Clasă abstractă = clasă care are cel puțin o funcție virtuală PURĂ

Necesitate: clase care dau doar interfață (nu vrem obiecte din clasă abstractă ci upcasting la ea).

Eroare la instantierea unei clase abstracte (nu se pot defini obiecte de tipul respectiv).

Permisă utilizarea de pointeri și referințe către clasă abstractă (pentru upcasting).

Nu pot fi trimise către funcții (prin valoare).



## Polimorfismul la execuție prin funcții virtuale

### *Funcții virtuale pure*

Sintaxa: **virtual tip\_returnat nume\_funcție(lista\_parametri) =0;**

Ex: **virtual int pura(int i)=0;**

Obs: La moștenire, dacă în clasa derivată nu se definește funcția pură, clasa derivată este și ea clasă abstractă ---> nu trebuie definită funcție care nu se execută niciodată

UTILIZARE IMPORTANTĂ: prevenirea “object slicing”.



## Polimorfismul la execuție prin funcții virtuale

### *Constructori și virtualizare*

**Obs.** NU putem avea constructori virtuali.

În general pentru funcțiile virtuale se utilizează late binding, dar în utilizarea funcțiilor virtuale în constructori, varianta locală este folosită (early binding)

De ce?

Pentru că funcția virtuală din clasa derivată ar putea crede că obiectul e inițializat deja

Pentru că la nivel de compilator în acel moment doar VPTR local este cunoscut



## Polimorfismul la execuție prin funcții virtuale

### *Destructori și virtualizare*

Este ușual să se întâlnească.

Se cheamă în ordine inversă decât constructorii.

*Dacă vrem să eliminăm porțiuni alocate dinamic și pentru clasa derivată dar facem upcasting trebuie să folosim destructori virtuali.*



## Polimorfismul la execuție prin funcții virtuale

### *Destructori și virtualizare*

```
class Base1 {public: ~Base1() { cout << "~Base1()\n"; } };
```

```
class Derived1 : public Base1 {public: ~Derived1() { cout << "~Derived1()\n"; } };
```

```
class Base2 {public:  
    virtual ~Base2() { cout << "~Base2()\n"; }  
};
```

```
class Derived2 : public Base2 {public: ~Derived2() { cout << "~Derived2()\n"; } };
```

```
int main()  
{  
    Base1* bp = new Derived1;  
    delete bp; // Afis: ~Base1()  
    Base2* b2p = new Derived2;  
    delete b2p; // Afis: ~Derived2() ~Base2()  
}
```



## Polimorfismul la execuție prin funcții virtuale

### *Destructori virtuali puri*

**Utilizare:** recomandat să fie utilizat dacă mai sunt și alte funcții virtuale.

**Restricție:** trebuie definiți în clasă (chiar dacă este abstractă).

La moștenire nu mai trebuie redefiniti (se construiește un destructor din oficiu)

De ce? Pentru a preveni instantierea clasei.

**Obs.** Nu are nici un efect dacă nu se face upcasting.

```
class AbstractBase {  
public:  
    virtual ~AbstractBase() = 0;  
};
```

```
AbstractBase::~AbstractBase()
```

```
class Derived : public AbstractBase {};  
// No overriding of destructor necessary?  
int main() { Derived d; }
```



## Polimorfismul la execuție prin funcții virtuale

### *Funcții virtuale în destructori*

La apel de funcție virtuală din funcții normale se apelează conform VPTR

În destructori se face early binding! (apeluri locale)

De ce? Pentru că acel apel poate să se bazeze pe porțiuni deja distruse din obiect

**class** Base { **public:**

**virtual** ~Base() { cout << "~Base1()\n"; this->f(); }

**virtual void** f() { cout << "Base::f()\n"; }

};

**class** Derived : **public** Base { **public:**

    ~Derived() { cout << "~Derived()\n"; }

**void** f() { cout << "Derived::f()\n"; }

};

**int** main() {

    Base\* bp = **new** Derived;

**delete** bp; // Afis: ~Derived() ~Base1() Base::f()

}



## Polimorfismul la execuție prin funcții virtuale

### *Decuplare în privința tipurilor*

**Upcasting** - Tipul derivat poate lua locul tipului de bază (foarte important pentru procesarea mai multor tipuri prin același cod).

Funcții virtuale: ne lasă să chemăm funcțiile pentru tipul derivat.

Problemă: apel la funcție prin pointer (tipul pointerului ne da funcția apelată).



## Polimorfismul la execuție prin funcții virtuale

### *Downcasting*

Folosit în ierarhii polimorfice (cu funcții virtuale).

**Problema:** upcasting e sigur pentru că respectivele funcții trebuie să fie definite în bază, downcasting e problematic.

Explicit cast prin: **dynamic\_cast**

*Dacă știm cu siguranță tipul obiectului putem folosi “static\_cast”.*

**Static\_cast** întoarce pointer către obiectul care satisfac cerințele sau 0.

Folosește tabelele VTABLE pentru determinarea tipului.



## Polimorfismul la execuție prin funcții virtuale

### *Downcasting*

```
class Pet { public: virtual ~Pet(){};  
class Dog : public Pet {};  
class Cat : public Pet {};  
  
int main() {  
    Pet* b = new Cat; // Upcast  
    Dog* d1 = dynamic_cast<Dog*>(b); // Afis - 0; Try to cast it to Dog*:  
    Cat* d2 = dynamic_cast<Cat*>(b); // Try to cast it to Cat*:  
    // b si d2 retin aceeasi adresa  
    cout << "d1 = " << d1 << endl;  
    cout << "d2 = " << d2 << endl;  
    cout << "b = " << b << endl;  
}
```



## Polimorfismul la execuție prin funcții virtuale

### Downcasting

```
class Shape { public: virtual ~Shape() {}; }
class Circle : public Shape {};
class Square : public Shape {};
class Other {};

int main() {
    Circle c;
    Shape* s = &c; // Upcast: normal and OK
    // More explicit but unnecessary:
    s = static_cast<Shape*>(&c);
    // (Since upcasting is such a safe and common
    // operation, the cast becomes cluttering)
    Circle* cp = 0;
    Square* sp = 0;
```

```
// Static Navigation of class hierarchies
// requires extra type information:
if(typeid(s) == typeid(cp)) // C++ RTTI
    cp = static_cast<Circle*>(s);
if(typeid(s) == typeid(sp))
    sp = static_cast<Square*>(s);
if(cp != 0)
    cout << "It's a circle!" << endl;
if(sp != 0)
    cout << "It's a square!" << endl;
// Static navigation is ONLY an efficiency
// hack;
// dynamic_cast is always safer. However:
// Other* op = static_cast<Other*>(s);
// Conveniently gives an error message,
// while
Other* op2 = (Other*)s;
// does not
}
```



## Operatorii de cast

C++ are 5 operatori de cast:

1) operatorul traditional mostenit din C;

2) **dynamic\_cast**;

3) **static\_cast**;

4) **const\_cast**;

5) **reinterpret\_cast**.



## Operatorii de cast

### ***Dynamic\_cast***

- daca vrem sa schimbam tipul unui obiect la executie;
- **se verifica daca un downcast este posibil (si deci valid);**
- daca e valid, atunci se poate schimba tipul, altfel eroare.

Sintaxa:

**dynamic\_cast <target-type> (expr)**

- target-type trebuie sa fie un pointer sau o referinta;

Dynamic\_cast **schimba tipul unui pointer/referinte** intr-un alt tip **pointer/referinta**.



## Operatorii de cast

### ***Dynamic\_cast***

Scop: cast pe tipuri polimorfice;

Exemplu:

```
class B{virtual ...};  
class D:B {...};
```

- un pointer D\* poate fi transformat oricand într-un pointer B\* (pentru că un pointer către baza poate oricand retine adresa unei derivate);
- invers este necesar operatorul dynamic\_cast.

*In general, dynamic\_cast reușește dacă pointerul (sau referinta) de transformat este un pointer (referinta) către un obiect de tip target-type, sau derivat din aceasta, altfel, încercarea de cast esuează (dynamic\_cast se evaluează cu null în cazul pointerilor și cu bad\_cast exception în cazul referinelor).*



## Operatorii de cast

### *Dynamic\_cast*

```
Base *bp, b_ob;
```

```
Derived *dp, d_ob;
```

```
bp = &d_ob; // base pointer points to Derived object
```

```
dp = dynamic_cast<Derived *> (bp); // cast to derived pointer OK
```

```
bp = &b_ob; // base pointer points to Base object
```

```
dp = dynamic_cast<Derived *> (bp); // error
```



## Operatorii de cast

### *Dynamic\_cast*

```
class Base { public:  
    virtual void f()  
    {  
        cout << "Inside Base\n";  
    }  
};  
  
class Derived : public Base {  
public:  
    void f()  
    {  
        cout << "Inside  
Derived\n";  
    }  
};
```

```
int main() {  
    Base *bp, b_ob;  
    Derived *dp, d_ob;  
  
    dp = dynamic_cast<Derived *> (&d_ob);  
    if(dp) { cout << "from Derived * to Derived* OK.\n";  
             dp->f(); }  
    else cout << "Error\n";  
  
    bp = dynamic_cast<Base *> (&d_ob);  
    if(bp) { cout << "from Derived * to Base * OK.\n";  
             bp->f(); }  
    else cout << "Error\n";  
  
    bp = dynamic_cast<Base *> (&b_ob);  
    if(bp) { cout << "from Base * to Base * OK.\n";  
             bp->f(); }  
    else cout << "Error\n";
```



## Operatorii de cast

### Dynamic\_cast

```
class Base { public:  
    virtual void f()  
    {  
        cout << "Inside Base\n";  
    }  
};  
  
class Derived : public Base {  
public:  
    void f()  
    {  
        cout << "Inside  
Derived\n";  
    }  
};
```

```
/*  Base *bp, b_ob;  
Derived *dp, d_ob; */  
  
dp = dynamic_cast<Derived *> (&b_ob);  
if(dp) cout << "Error\n";  
else  
cout << "Cast from Base* to Derived* not OK.\n";  
  
bp = &d_ob; // bp points to Derived object  
dp = dynamic_cast<Derived *> (bp);  
if(dp) {  
    cout << "Casting bp to a Derived * OK\n" <<  
    "because bp is really pointing\n" <<  
    "to a Derived object.\n";  
    dp->f();  
}  
else cout << "Error\n";
```



## Operatorii de cast

### Dynamic\_cast

```
class Base { public:  
    virtual void f()  
    {  
        cout << "Inside Base\n";  
    }  
};  
class Derived : public Base {  
public:  
    void f()  
    {  
        cout << "Inside  
Derived\n";  
    }  
};
```

```
/* Base *bp, b_ob;  
   Derived *dp, d_ob; */  
  
    bp = &b_ob; // bp points to Base object  
    dp = dynamic_cast<Derived *> (bp);  
    if(dp) cout << "Error";  
    else { cout << "Now casting bp to a Derived *\n"  
           is not OK because bp is really \n pointing to a Base  
           object.\n"; }  
  
    dp = &d_ob; // dp points to Derived object  
    bp = dynamic_cast<Base *> (dp);  
    if(bp) { cout << "Casting dp to a Base * is OK.\n";  
             bp->f();}  
    else cout << "Error\n";  
    return 0;  
}
```



## Operatorii de cast

### ***Dynamic\_cast*** Afisare:

Cast from Derived \* to Derived \* OK.

Inside Derived

Cast from Derived \* to Base \* OK.

Inside Derived

Cast from Base \* to Base \* OK.

Inside Base

Cast from Base \* to Derived \* not OK.

Casting bp to a Derived \* OK

because bp is really pointing

to a Derived object.

Inside Derived

Now casting bp to a Derived \*

is not OK because bp is really

pointing to a Base object.

Casting dp to a Base \* is OK.

Inside Derived



## Operatorii de cast

### ***Static\_cast***

- este un substitut pentru operatorul de cast clasic;
- lucreaza pe tipuri nepolimorfice;
- poate fi folosit pentru orice conversie standard;
- ne se fac verificari la executie (run-time);

### **Sintaxa: static\_cast <type> (expr)**

Expl:

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    for(i=0; i<10; i++)
        cout << static_cast<double> (i) / 3 << " ";
    return 0;
}
```



## Operatorii de cast

### ***Const\_cast***

- folosit pentru a rescrie, explicit, proprietatea de const sau volatile intr-un cast (elimina proprietatea de a fi constant);
- tipul destinatie trebuie sa fie acelasi cu tipul sursa, cu exceptia atributelor const / volatile.

Sintaxa: `const_cast <type> (expr)`



## Operatorii de cast

**Const\_cast** Exemplu - pointer

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

void sqrval(const int *val) {
    int *p;
// cast away const-ness.
p = const_cast<int *>(val);
    *p = *val **val; // now, modify object through v
}
int main()
{
    int x = 10;
    cout << "x before call: " << x << endl;
    sqrval(&x);
    cout << "x after call: " << x << endl;
    return 0;
}
```



## Operatorii de cast

***Const\_cast*** Exemplu - referinta

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

void sqrval(const int &val) {
    // cast away const on val
const_cast<int &>(val) = val * val;
}

int main()
{
    int x = 10;
    cout << "x before call: " << x << endl;
    sqrval(x);
    cout << "x after call: " << x << endl;
    return 0;
}
```



## Operatorii de cast

### *Reinterpret\_cast*

- converteste un tip într-un alt tip fundamental diferit;

Sintaxa: reinterpret\_cast <type> (expr)

Expl:

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

```
int main()
{
    int* p = new int(65);
    char* ch = reinterpret_cast<char*>(p);
    cout << *p << endl;
    cout << *ch << endl;
    cout << p << endl;
    cout << ch << endl;
    return 0;
}
```



## Tratarea exceptiilor în C++

O excepție este o problemă care apare în timpul execuției unui program.

O excepție C++ este un răspuns la o circumstanță excepțională care apare în timpul rulării unui program, (probleme la alocare, încercare de împărțire la zero, etc.)

- automatizarea procesării erorilor
- try, catch, throw
- block try aruncă excepție cu throw care este prinsă cu catch
- după ce este prinsă se termină execuția din blocul catch și se dă controlul “mai sus”, nu se revine la locul unde s-a făcut throw (nu e apel de funcție).



## Tratarea exceptiilor în C++

```
try {  
    // try block  
}  
  
catch (type1 arg) {  
    // catch block  
}  
  
catch (type2 arg) {  
    // catch block  
}  
  
catch (type3 arg) {  
    // catch block  
}...  
  
catch (typeN arg) {  
    // catch block  
}
```

tipul argumentului arg din catch arată care bloc catch este executat

dacă nu este generată excepție, nu se execută nici un bloc catch

instrucțiunile catch sunt verificate în ordinea în care sunt scrise, primul de tipul erorii este folosit



## Tratarea exceptiilor în C++

### *Observații:*

- dacă se face `throw` și nu există un bloc `try` din care a fost aruncată excepția sau o funcție apelată dintr-un bloc `try`: eroare
- dacă nu există un `catch` care să fie asociat cu `throw`-ul respectiv (tipuri de date egale) atunci programul se termină prin `terminate()`
- `terminate()` poate să fie redefinită să facă altceva



## Tratarea exceptiilor în C++

**Exemplu - Semnalarea unei posibile erori la alocarea de memorie: bad\_alloc**

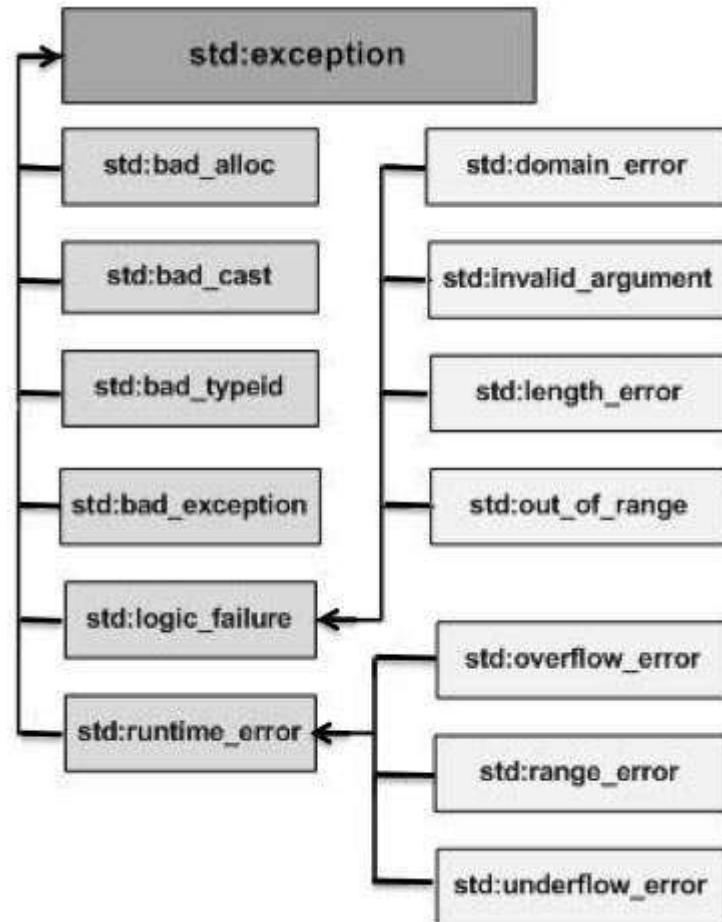
```
class TestTry {
    int *v, n;
public:
    TestTry(int a) {
        try {
            v = new int[a];
        }
        catch (bad_alloc Nume_Var) {
            cout << "Allocation Failure\n";
            exit(EXIT_FAILURE);
        }
        n = a;
    }
};

int main() { TestTry T(4); }
```



## Tratarea exceptiilor în C++

### *Exceptii standard de biblioteca <exception>*



Sursa: [https://www.tutorialspoint.com/cplusplus/cpp\\_exceptions\\_handling.htm](https://www.tutorialspoint.com/cplusplus/cpp_exceptions_handling.htm)



## Tratarea exceptiilor în C++

*Tipul aruncat coincide cu tipul parametrului blocului catch*

```
void Test_Throw_ok () {  
    try {  
        throw 10;  
    }  
    catch (int x) {  
        cout << "Exceptie 10\n";  
    }  
}
```

```
int main() {  
    Test_Throw_ok();  
}
```

Exceptia este prinsă; se afișează expresia din blocul catch

```
void Test_Throw_ok () {  
    try {  
        throw 10;  
    }  
    catch (char x) {  
        cout << "Exceptie 10\n";  
    }  
}
```

```
int main() {  
    Test_Throw_ok();  
}
```

Exceptia nu este prinsă



## Tratarea exceptiilor în C++

### Aruncarea unei exceptii dintr-o functie (throw în functie)

```
class TestTry {
public:
    void Test_Throw_Functie() {
        try {
            Test(5);
            Test(200);
            Test(-300);
            Test(22);
        }
        catch (int x) {
            cout << "Exceptie pe valoarea " << x << "\n";
        }
    }
};
```

```
void Test(int x)
{
    cout << "In functie x = " << x << "\n";
    if (x < 0) throw x;
}
```

```
int main() {
    TestTry T;
    T.Test_Throw_Functie();
}
```

In functie x = 5  
In functie x = 200  
In functie x = -300  
Exceptie pe valoarea -300



## Tratarea exceptiilor în C++

*Try-catch local, în funcție, se continuă execuția programului*

```
class TestTry {
public:
    void Test_Try_Local()
    {
        int x;
        x = -25;
        Try_in_functie(x);
        x = 13;
        Try_in_functie(x);
        n = x;
        cout << n;
    }
};
```

```
void Try_in_functie(int x)
{
    try
    {
        if (x < 0) throw x;
    }
    catch(int x)
    {
        cout << "Exceptie pe valoarea " << x << "\n";
    }
}
```

Exceptie pe valoarea -25  
13

```
int main()
{
    TestTry T;
    T.Test_Try_Local();
}
```



## Tratarea exceptiilor în C++

### *Excepții multiple; catch (...)*

```
void Exceptii_multiple(int x) {
    try{
        if (x < 0) throw x; //int
        if (x == 0) throw 'A'; //char
        if (x > 0) throw 12.34; //double
    }
    Catch(int){...}
}
catch(...){
    cout << "Catch macar una!\n";
}
int main(){
    Exceptii_multiple(-52);
    Exceptii_multiple(0);
    Exceptii_multiple(34);
}
```



## Tratarea exceptiilor în C++

- aruncarea de erori din clase de bază și derivate
- un catch pentru tipul de bază va fi executat pentru un obiect aruncat de tipul derivat
- să se pună catch-ul pe tipul derivat primul și apoi catchul pe tipul de bază

```
class B {};
class D: public B {};
int main()
{
    D derived;
    try { throw derived; }

    catch(B b) { cout << "Caught a base class.\n"; }

    catch(D d) { cout << "This won't execute.\n"; }

    return 0;
}
```



## Tratarea exceptiilor în C++

La definiția unei funcții (metode), se poate preciza lista tipurilor de exceptii care pot fi generate în cadrul funcției.

***void Functie (int test) throw(int, char)***

- se poate specifica ce exceptii aruncă o funcție
- se restricționează tipurile de exceptii care se pot arunca din funcție
- un alt tip nespecificat termină programul:
  - apel la unexpected() care apelează abort()
  - se poate redefini



## Tratarea exceptiilor în C++

*Exemplu funcție care precizeaza lista tipurilor de exceptii*

```
void Functie(int x) throw (int,char)
{
    if (x < 0) throw x;
    if (x == 0) throw 'a';
    if (x > 0) throw 1.2;
}

int main()
{
    try
    {
        //          Functie(-1);
        //          Functie(0);
        Functie(1);
    }
    catch (int a){ cout << "int: " << a; }
    catch (char a){ cout << "char: " << a; }
    catch (double a){ cout << "double: " << a; }
    return 0;
}
```

*Observație: lista cu tipurile de exceptii poate fi nulă, caz în care nu se acceptă nici o eroare:*

Unele compilatoare pot da warninguri, altele termina executia abrupt:  
“terminate called after throwing an instance of ‘double’ ”



## Tratarea exceptiilor în C++

### Rearuncarea unei exceptii

- re-aruncarea unei exceptii: throw; // fără exceptie din catch

```
void Rearuncare_exceptie(int x)
{
    try{
        if (x < 0) throw x;
        cout<<"A\n";
    }
    catch(int x) {
        cout<<"B\n";
        throw;
    }
    cout<<"C\n";
}

int main()
{
    try
    {
        Rearuncare_exceptie(-1);
    }
    catch (int a){ cout << "D\n"; }
    cout << "E\n";
}
```

Se afiseaza:

B

D

E



## Tratarea exceptiilor în C++

XVIII. Spuneți dacă programul de mai jos este corect. În caz afirmativ, spuneți ce afișează pentru o valoare întreagă citită egală cu 7, în caz negativ spuneți de ce nu este corect.

```
#include <iostream.h>
float f(int y)
{ try
  { if (y%2) throw y/2;
  }
  catch (int i)
  { if (i%2) throw;
    cout<<"Numarul "<<
  }
  return y/2;
}
int main()
{ int x;
  try
  { cout<<"Da-mi un nu
    cin>>x;
    if (x) f(x);
    cout<<"Numarul "<<
  }
  catch (int i)
  { cout<<"Numarul "<<
  }
  return 0;
}
```

Da-mi un numar intreg: -2	Numarul -2 nu e bun!
Da-mi un numar intreg: -1	Numarul 0 nu e bun!
<b>Numarul -1 nu e bun!</b>	
Da-mi un numar intreg: 0	Numarul 0 nu e bun!
Da-mi un numar intreg: 1	Numarul 0 nu e bun!
<b>Numarul 1 nu e bun!</b>	
Da-mi un numar intreg: 2	Numarul 2 nu e bun!
Da-mi un numar intreg: 3	Numarul 1 e bun!
Da-mi un numar intreg: 4	Numarul 4 nu e bun!
Da-mi un numar intreg: 5	Numarul 2 nu e bun!
<b>Numarul 5 nu e bun!</b>	
Da-mi un numar intreg: 6	Numarul 6 nu e bun!
Da-mi un numar intreg: 7	Numarul 3 e bun!
Da-mi un numar intreg: 8	Numarul 8 nu e bun!
Da-mi un numar intreg: 9	Numarul 4 nu e bun!
<b>Numarul 9 nu e bun!</b>	
Da-mi un numar intreg: 10	Numarul 10 nu e bun!



## Tratarea exceptiilor în C++

*Implementarea unei ierarhii de clase de exceptii pornind de la std::exception*

Varianta C++98 – Detalii despre <exception> si toate functiile sale membre se pot gasi:  
<https://www.cplusplus.com/reference/exception/exception/>

```
1 class exception {
2 public:
3     exception () throw();
4     exception (const exception&) throw();
5     exception& operator= (const exception&) throw();
6     virtual ~exception() throw();
7     virtual const char* what() const throw();
8 }
```

```
class MyException : public exception {
public:
    const char * what () const throw () { return "Particularizat\n"; }

int main() {
    try {
        throw MyException();
    } catch(MyException& e) {
        cout << "Prins\n";
        cout << e.what() << "\n";
    } catch(std::exception& e) {
        cout << "Alte erori\n";
    }
    return 0;
}
```



## Tratarea exceptiilor în C++

*Implementarea unei ierarhii de clase de exceptii pornind de la std::exception*

```
class MyException : public exception {
    public:
        const char * what () const throw () { return "Exceptie\n"; }

    class Exceptie_matematica : public MyException
    {
    public:
        const char * what () const throw () { return "Exceptie matematica\n"; }
    };

int main() {
    try {
        //throw MyException();
        throw Exceptie_matematica();
    }
    catch(MyException& e) { cout << e.what() << "\n"; }
    catch(Exceptie_matematica& e) { cout << e.what() << "\n"; }
    return 0;
}
```

Se afiseaza Exceptie Matematica



## Tratarea exceptiilor în C++

### Functia terminate()

```
class A { };
class B { };

void f() { throw A(); }
void g() { throw B(); }

void my_terminate() {
    cout << "Terminate personalizat\n";
    abort();
}

void (*old_terminate) () = set_terminate(my_terminate);
```

```
int main()
{
    try
    {
        //f();
        g();
    }
    catch (A)
    {
        cout << "Exceptie A.\n";
    }
}
```

Se afiseaza Terminate personalizat si apoi se termina programul, incorect.



## Tratarea exceptiilor în C++

### *Functia unexpected()*

```
void g()
{
    throw "Surpriza.";
}

void f(int x) throw (int)
{
    if (x <= 0) throw 123;
    g();
}

void my_unexpected() {
    cout << "Exceptie neprevazuta.";
    exit(1);
}
```

```
int main()
{
    set_unexpected(my_unexpected);
    try {
        f(10);
    } catch (int) {
        cout << "Exceptie int" << endl;
    } catch (double) {
        cout << "Exceptie double" << endl;
    }
    return 0;
}
```

Vedeti ce se intampla sub compilatorul g++! Surpriza!



# Perspective

Cursul 9:

Templates/sabloane



# **Programare orientată pe obiecte**

## **- suport de curs -**

**Andrei Păun  
Anca Dobrovăț**

**An universitar 2021 – 2022  
Semestrul II  
Seriile 13, 14 și 15**

**Curs 9**



## Agenda cursului

Controlul tipului în timpul rulării

Şabloane în C++ (Templates)



## Controlul tipului în timpul rulării programului în C++

Facilitati C++ adaugate in cadrul polimorfismului la executie:

- 1) ***Run - Time Type Identification (RTTI)*** - permite identificarea tipului unui obiect in timpul executiei programului;
- 1) set aditional de 4 operatori de cast (***dynamic\_cast***, ***const\_cast***, ***reinterpret\_cast***, si ***static\_cast***) - pentru o modalitate mai sigura de cast:

Unul dintre operatori, ***dynamic\_cast***, este legat direct de mecanismul RTTI.



## Controlul tipului în timpul rulării programului în C++

### *Run - Time Type Identification (RTTI)*

- nu se regaseste in limbajele nepolimorfice (expl. C), intrucat nu e nevoie de informatie la executie, pentru ca tipul fiecarui obiect este cunoscut la compilare (expl. in timpul scrierii);
- in limbajele polimorfice (expl. C++) pot aparea situatii in care tipul unui obiect nu este cunoscut pana la executia programului;
- C++ implementeaza polimorfismul prin mostenire, functii virtuale si pointeri catre clasa de baza care pot fi utilizati pentru a arata catre obiecte din clase derivate, deci nu se poate sti a-priori tipul obiectului catre care se pointeaza.

**Determinarea se face la executie, folosind RTTI.**



## Controlul tipului în timpul rulării programului în C++

### *Run - Time Type Identification (RTTI)*

- **typeid** - pentru a obtine tipul obiectului;
- **#include <typeinfo>;**
- uzual **typeid(object);**
- tipul obiectului: predefinit sau definit de utilizator;
- typeid - returneaza o referinta catre un obiect de tip **type\_info** care descrie tipul obiectului;
- **typeid(NULL) genereaza exceptie: bad\_typeid**



## Controlul tipului în timpul rulării programului în C++

### *Run - Time Type Identification (RTTI)*

#### **Clasa type\_info**

Membri publici:

- **bool operator==(const type\_info &ob);**
- **bool operator!=(const type\_info &ob);**
  - pentru doua obiecte se poate verifica daca au sau nu acelasi tip;
- **bool before(const type\_info &ob);**
  - putem verifica daca un type\_info precede un alt type\_info:  
**if(typeid(obiect1).before(typeid(obiect2)))**
  - se foloseste in implementarea type\_info ca si chei pentru o structura;
- **const char \*name( );**
  - sirul exact intors depinde de compilator dar contine si tipul obiectului.



## Controlul tipului în timpul rulării programului în C++

### *Run - Time Type Identification (RTTI)*

#### Exemplu cu tipuri predefinite

```
#include <iostream>
#include <typeinfo>
using namespace std;
int main() {
    int a, b;    float c;    char *p;
    cout << "The type of a is: " << typeid(a).name() << endl;
    cout << "The type of c is: " << typeid(c).name() << endl;
    cout << "The type of p is: " << typeid(p).name() << endl;
    if(typeid(a) == typeid(b))
        cout << "The types of i and j are the same\n";
    if(typeid(a) != typeid(c))
        cout << "The types of i and f are not the same\n";
}

// Pe compilatorul personal s-au afisat: i (pt int), f(pentru float) si Pc(pentru char*)
```



## Controlul tipului în timpul rulării programului în C++

### *Run - Time Type Identification (RTTI)*

#### Exemplu cu tipuri definite de utilizator

```
#include <iostream>
#include <typeinfo>
using namespace std;
class myclass1{ ... };
class myclass2{ ... };
int main() {
    myclass1 ob1;
    myclass2 ob2;
    cout << "The type of ob1 is: " << typeid(ob1).name() << endl;
    cout << "The type of ob2 is: " << typeid(ob2).name() << endl;
    if(typeid(ob1) != typeid(ob2)) cout << "ob1 and ob2 are of differing types\n";
}

// Pe compilatorul personal s-au afisat: 8myclass1 (pt ob1), 8myclass2(pentru ob2);
```



## Controlul tipului în timpul rulării programului în C++

### *Run - Time Type Identification (RTTI)*

#### Cea mai importantă utilizare a **typeid** - tipuri polimorfice

```
class Baza {public: virtual void f () { } }; // tip polimorfic
class Derivata1: public Baza {};
class Derivata2: public Baza {};
int main() {    Baza *p, b;    Derivata1 d1;    Derivata2 d2;
    p = &b;
    cout << "p is pointing to an object of type " << typeid(*p).name() << endl;
    p = &d1;
    cout << "p is pointing to an object of type " << typeid(*p).name() << endl;
    p = &d2;
    cout << "p is pointing to an object of type " << typeid(*p).name() << endl;
    return 0;
}

// Pe compilatorul personal s-au afisat: 4Baza; 9Derivata1; 9Derivata2
```



## Controlul tipului în timpul rulării programului în C++

### *Run - Time Type Identification (RTTI)*

#### Demonstrarea typeid cu referinte

```
class Baza {public: virtual void f () { } };// tip polimorfic
class Derivata1: public Baza {};
class Derivata2: public Baza {};

void WhatType(Baza &ob){
    cout << "ob is referencing an object of type " << typeid(ob).name() << endl;

int main() {    Baza b;    Derivata1 d1;    Derivata2 d2;
    WhatType(b);
    WhatType(d1);
    WhatType(d2);

    return 0;
}
```



## Controlul tipului în timpul rulării programului în C++

### *Run - Time Type Identification (RTTI)*

Alta modalitate de utilizare **typeid**:

#### **typeid(type-name)**

Expl: **cout << typeid(int).name();**

Expl de utilizare:

```
class Baza {public: virtual void f () { } };// tip polimorfic
class Derivata1: public Baza { };
class Derivata2: public Baza { };
```

**void WhatType(Baza &ob)**

{

```
cout << "ob is referencing an object of type " << typeid(ob).name() << endl;
if(typeid(ob) == typeid(Baza)) cout << "Baza.\n";
if(typeid(ob) == typeid(Derivata1)) cout << "Derivata1.\n";
```

}



## Controlul tipului în timpul rulării programului în C++

### *Run - Time Type Identification (RTTI)*

Un exemplu cu o functie, denumita “**factory**” care produce obiecte de diferite tipuri (in general, o astfel de functie se numeste “**object factory**” - ne vom mai intalni cu acest concept la Design Patterns).

```
class Baza {public: virtual void f () { } }; // tip polimorfic
class Derivata1: public Baza { };
class Derivata2: public Baza { };
```

```
Baza *factory() {
    switch(rand()%2)  {
        case 0:
            return new Derivata1;
        case 1:
            return new Derivata2;
    }
    return 0;
}
```



## Controlul tipului în timpul rulării programului în C++

### *Run - Time Type Identification (RTTI)*

```
class Baza {public: virtual void f () { } };// tip polimorfic
class Derivata1: public Baza { };
class Derivata2: public Baza { };
Baza *factory() {}
int main()
{
    Baza *b;
    int nr1 = 0, nr2 = 0;
    for(int i=0; i<10; i++)  {
        b = factory(); // generate an object
        cout << "Object is " << typeid(*b).name() << endl;
        // count it
        if(typeid(*b) == typeid(Derivata1)) nr1++;
        if(typeid(*b) == typeid(Derivata2)) nr2++;  }
    cout<<nr1<<"\t"<<nr2;
    return 0;
}
```



## Controlul tipului în timpul rulării programului în C++

*Run - Time Type Identification (RTTI)*

**Nu functioneaza cu pointeri void, nu au informatie de tip**



## Controlul tipului în timpul rulării programului în C++

### *Operatorii de cast*

C++ are 5 operatori de cast:

1) operatorul traditional mostenit din C;

2) **dynamic\_cast**;

3) **static\_cast**;

4) **const\_cast**;

5) **reinterpret\_cast**.



## Controlul tipului în timpul rulării programului în C++

### *Dynamic\_cast*

- daca vrem sa schimbam tipul unui obiect la executie;
- **se verifica daca un downcast este posibil (si deci valid);**
- daca e valid, atunci se poate schimba tipul, altfel eroare.

Sintaxa:

**dynamic\_cast <target-type> (expr)**

- target-type trebuie sa fie un pointer sau o referinta;

Dynamic\_cast schimba tipul unui pointer/referinte intr-un alt tip pointer/referinta.



## Controlul tipului în timpul rulării programului în C++

### ***Dynamic\_cast***

Scop: cast pe tipuri polimorfice;

Exemplu:

```
class B{virtual ...};  
class D:B {...};
```

- un pointer D\* poate fi transformat oricand într-un pointer B\* (pentru că un pointer către baza poate oricând retine adresa unei derivate);
- invers este necesar operatorul dynamic\_cast.

*In general, dynamic\_cast reușește dacă pointerul (sau referinta) de transformat este un pointer (referinta) către un obiect de tip target-type, sau derivat din aceasta, altfel, încercarea de cast esuează (dynamic\_cast se evaluează cu null în cazul pointerilor și cu bad\_cast exception în cazul referinelor).*



## Controlul tipului în timpul rulării programului în C++

### *Dynamic\_cast*

```
Base *bp, b_ob;
```

```
Derived *dp, d_ob;
```

```
bp = &d_ob; // base pointer points to Derived object
```

```
dp = dynamic_cast<Derived *> (bp); // cast to derived pointer OK
```

```
bp = &b_ob; // base pointer points to Base object
```

```
dp = dynamic_cast<Derived *> (bp); // error
```



## Controlul tipului în timpul rulării programului în C++

### *Dynamic\_cast*

```
class Base { public:  
    virtual void f()  
    {  
        cout << "Inside Base\n";  
    }  
};  
  
class Derived : public Base {  
public:  
    void f()  
    {  
        cout << "Inside  
Derived\n";  
    }  
};
```

```
int main() {  
    Base *bp, b_ob;  
    Derived *dp, d_ob;  
  
    dp = dynamic_cast<Derived *> (&d_ob);  
    if(dp) { cout << "from Derived * to Derived* OK.\n";  
             dp->f(); }  
    else cout << "Error\n";  
  
    bp = dynamic_cast<Base *> (&d_ob);  
    if(bp) { cout << "from Derived * to Base * OK.\n";  
             bp->f(); }  
    else cout << "Error\n";  
  
    bp = dynamic_cast<Base *> (&b_ob);  
    if(bp) { cout << "from Base * to Base * OK.\n";  
             bp->f(); }  
    else cout << "Error\n";
```



## Controlul tipului în timpul rulării programului în C++

### Dynamic\_cast

```
class Base { public:  
    virtual void f()  
    {  
        cout << "Inside Base\n";  
    }  
};  
  
class Derived : public Base {  
public:  
    void f()  
    {  
        cout << "Inside  
Derived\n";  
    }  
};
```

```
/*  Base *bp, b_ob;  
    Derived *dp, d_ob; */  
  
dp = dynamic_cast<Derived *> (&b_ob);  
if(dp) cout << "Error\n";  
else  
    cout << "Cast from Base* to Derived* not OK.\n";  
  
bp = &d_ob; // bp points to Derived object  
dp = dynamic_cast<Derived *> (bp);  
if(dp) {  
    cout << "Casting bp to a Derived * OK\n" <<  
        "because bp is really pointing\n" <<  
        "to a Derived object.\n";  
    dp->f();  
}  
else cout << "Error\n";
```



## Controlul tipului în timpul rulării programului în C++

### Dynamic\_cast

```
class Base { public:  
    virtual void f()  
    {  
        cout << "Inside Base\n";  
    }  
};  
class Derived : public Base {  
public:  
    void f()  
    {  
        cout << "Inside  
Derived\n";  
    }  
};
```

```
/* Base *bp, b_ob;  
   Derived *dp, d_ob; */  
  
    bp = &b_ob; // bp points to Base object  
    dp = dynamic_cast<Derived *> (bp);  
    if(dp) cout << "Error";  
    else { cout << "Now casting bp to a Derived *\n"  
           is not OK because bp is really \n pointing to a Base  
           object.\n"; }  
  
    dp = &d_ob; // dp points to Derived object  
    bp = dynamic_cast<Base *> (dp);  
    if(bp) { cout << "Casting dp to a Base * is OK.\n";  
             bp->f();}  
    else cout << "Error\n";  
    return 0;  
}
```



## Controlul tipului în timpul rulării programului în C++

***Dynamic\_cast*** Afisare:

Cast from Derived \* to Derived \* OK.

Inside Derived

Cast from Derived \* to Base \* OK.

Inside Derived

Cast from Base \* to Base \* OK.

Inside Base

Cast from Base \* to Derived \* not OK.

Casting bp to a Derived \* OK

because bp is really pointing

to a Derived object.

Inside Derived

Now casting bp to a Derived \*

is not OK because bp is really

pointing to a Base object.

Casting dp to a Base \* is OK.

Inside Derived



## Controlul tipului în timpul rulării programului în C++

**Dynamic\_cast** înlocuiește **typeid**

Fie Base și Derived 2 clase polimorfice.

Base \*bp;

Derived \*dp;

```
// ... if(typeid(*bp) == typeid(Derived)) dp = (Derived *) bp;
```

- cast obisnuit;
- îl verifică valabilitatea operării de cast

**Cel mai indicat:**

```
dp = dynamic_cast <Derived *> (bp);
```



## Controlul tipului în timpul rulării programului în C++

**Dynamic\_cast** înlocuiește **typeid**

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Base {
public:
    virtual void f() { }
};

class Derived : public Base {
public:
void derivedOnly()  {
    cout << "Is a Derived Object.\n"; }
};

int main()
{
    Base *bp, b_ob;
    Derived *dp, d_ob;
```

// use typeid

```
bp = &b_ob;
if(typeid(*bp) == typeid(Derived))  {
    dp = (Derived *) bp;
    dp->derivedOnly();  }
else  cout << "Cast from Base to
Derived failed.\n";

bp = &d_ob;
if(typeid(*bp) == typeid(Derived))  {
    dp = (Derived *) bp;
    dp->derivedOnly();  }
else  cout << "Error, cast should
work!";

    return 0;
}
```



## Controlul tipului în timpul rulării programului în C++

**Dynamic\_cast** înlocuieste **typeid**

```
#include <iostream>
#include <typeinfo>
using namespace std;
class Base {
public:
    virtual void f() { }
};

class Derived : public Base {
public:
    void derivedOnly()  {
        cout << "Is a Derived Object.\n";
    }
int main()
{
    Base *bp, b_ob;
    Derived *dp, d_ob;
```

// use dynamic\_cast

```
bp = &b_ob;
dp = dynamic_cast<Derived *>(bp);
if(dp) dp->derivedOnly();
else cout << "Cast from Base to
Derived failed.\n";

bp = &d_ob;
dp = dynamic_cast<Derived *>(bp);
if(dp) dp->derivedOnly();
else cout << "Error, cast should
work!\n";
return 0;
}
```



## Controlul tipului în timpul rulării programului în C++

### **Static\_cast**

- este un substitut pentru operatorul de cast clasic;
- lucreaza pe tipuri nepolimorfice;
- poate fi folosit pentru orice conversie standard;
- ne se fac verificari la executie (run-time);

### **Sintaxa: static\_cast <type> (expr)**

Expl:

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    for(i=0; i<10; i++)
        cout << static_cast<double> (i) / 3 << " ";
    return 0;
}
```



## Controlul tipului în timpul rulării programului în C++

### Const\_cast

- folosit pentru a rescrie, explicit, proprietatea de const sau volatile intr-un cast (elimina proprietatea de a fi constant);
- tipul destinatie trebuie sa fie acelasi cu tipul sursa, cu exceptia atributelor const / volatile.

Sintaxa: `const_cast <type> (expr)`



## Controlul tipului în timpul rulării programului în C++

***Const\_cast*** Exemplu - pointer

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

void sqrval(const int *val) {
    int *p;
// cast away const-ness.
p = const_cast<int *>(val);
    *p = *val **val; // now, modify object through v
}
int main()
{
    int x = 10;
    cout << "x before call: " << x << endl;
    sqrval(&x);
    cout << "x after call: " << x << endl;
    return 0;
}
```



## Controlul tipului în timpul rulării programului în C++

***Const\_cast*** Exemplu - referinta

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

void sqrval(const int &val) {
// cast away const on val
const_cast<int &> (val) = val * val;
}

int main()
{
int x = 10;
cout << "x before call: " << x << endl;
sqrval(x);
cout << "x after call: " << x << endl;
return 0;
}
```



## Controlul tipului în timpul rulării programului în C++

### *Reinterpret\_cast*

- converteste un tip într-un alt tip fundamental diferit;

Sintaxa: reinterpret\_cast <type> (expr)

Expl:

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

```
int main()
{
    int* p = new int(65);
    char* ch = reinterpret_cast<char*>(p);
    cout << *p << endl;
    cout << *ch << endl;
    cout << p << endl;
    cout << ch << endl;
    return 0;
}
```



## Şabloane (Templates) în C++

### Funcții generice

Mulți algoritmi sunt generici (nu contează pe ce tip de date operează).

Înlăturăm bug-uri și mărim viteza implementării dacă reușim să refolosim aceeași implementare pentru un algoritm care trebuie folosit cu mai multe tipuri de date.

O singură implementare, mai multe folosiri.

O funcție generică face auto overload (pentru diverse tipuri de date).

**Sintaxa:**

```
template <class Ttype> tip_returnat nume_funcție(listă_de_argumete) {
// corpul funcției
}
```

Ttype este un nume pentru tipul de date folosit (încă indecis), compilatorul îl va înlocui cu tipul de date folosit.



## Şabloane (Templates) în C++

### Funcţii generice

**template <class Ttype>** // e ok şi **template <typename Ttype>**

```
Ttype maxim (Ttype V[ ], int n) {
    Ttype max = V[0];
    for (int i = 1; i < n; i++) {
        if (max < V[i]) max = V[i];
    }
    return max;
}
```

```
int main ()
{
    int VI[] = {1, 5, 3, 7, 3};
    float VF[] = {(float)1.1, (float)5.1, (float)3.1, (float)4.1};
    cout << "maxim (VI): " << maxim<int> (VI, sizeof (VI)/sizeof (int))<< endl;
    cout << "maxim (VF): " << maxim<float> (VF, sizeof (VF)/ sizeof (double)) << endl;
}
```



## Şabloane (Templates) în C++

### Funcții generice

**Specificația de template trebuie să fie imediat înaintea definiției funcției.**

```
template <class Ttype>
int i; // this is an error
void swapargs(Ttype &a, Ttype &b)
{
    Ttype temp;
    temp = a;
    a = b;
    b = temp;
}
```



## Şabloane (Templates) în C++

### Funcții generice

***Putem avea funcții cu mai mult de un tip generic.***

compilatorul creează atâtea funcții cu același nume câte sunt necesare (d.p.d.v. al parametrilor folosiți).

```
template <class type1, class type2>
void myfunc(type1 x, type2 y)
{
    cout << x << ' ' << y << '\n';
}

int main()
{
    myfunc(10, "I like C++");
    myfunc(98.6, 19L);
    return 0;
}
```



## Şabloane (Templates) în C++

### Funcții generice

#### *Overload pe şablon*

Şablon: overload implicit

Putem face overload explicit

Se numeşte “**specializare explicită**”

In cazul specializării explicate versiunea şablonului care s-ar fi format în cazul numărului şi tipurilor de parametrii respectivi nu se mai creează (se foloseşte versiunea explicită)



## Şabloane (Templates) în C++

### Funcţii generice

#### *Overload pe şabloane - Specializare explicită*

```
template <class T> T maxim( T a, T b)
```

```
{
```

```
    cout<<"template"<<endl;
    if (a>b) return a; // operatorul < trebuie să fie definit pentru tipul T
    return b;
```

```
}
```

```
template <> char * maxim ( char* a, char* b)
```

```
{
```

```
    cout<<"suprîncărcare neconst"<<endl;
    if (strcmp(a,b)>0) return a;
    return b;
```

```
}
```



## Şabloane (Templates) în C++

### Functii generice

#### *Overload pe şabloane - Specializare explicită*

```
template <class T> T maxim( T a, T b)
{
    cout<<"template"<<endl;
    if (a>b) return a; // operatorul < trebuie să fie definit pentru tipul T
    return b;
}
```

```
template <> const char * maxim(const char* a,const char* b)
{
    cout<<"supraîncărcare const"<<endl;
    if (strcmp(a,b)>0) return a;
    return b; }
```

```
template <> char * maxim ( char* a, char* b)
{
    cout<<"supraîncărcare neconst"<<endl;
    if (strcmp(a,b)>0) return a;
    return b; }
```



## Şabloane (Templates) în C++

### Funcții generice

#### *Overload pe şabloane - Specializare explicită*

```
// pot exista ambele variante
//dacă nu există template<> const char* -pt "ab" se alege şablonul general
//dacă nu există template<> char* -pt v1 se alege şablonul general
/* NU FACE CONVERSIA NICI (char *) --> (const char *) și nici (const char *) --> (char *) */

int main(int argc, char *argv[])
{
    char v1[10] = "abc", v2[10] = "bcd";
    cout << maxim("ab", "bc") << endl;
    cout << maxim(v1, v2) << endl;
    cout << maxim<char *>(v1, "ab") << endl;
    return 0;
}
```



## Şabloane (Templates) în C++

### Funcții generice

#### *Overload pe şabloane*

Diferită de specializare explicită

Similar cu overload pe funcții (doar că acum sunt funcții generice)

Simplu: la fel ca la funcțiile normale



## Şabloane (Templates) în C++

### Functii generice

#### Overload pe şabloane

```
// First version of f() template.  
template <class X> void f(X a) {  
    cout << "Inside f(X a)\n";  
}
```

```
// Second version of f() template.  
template <class X, class Y> void f(X a, Y b) {  
    cout << "Inside f(X a, Y b)\n";  
}
```

```
int main() {  
    f(10); // calls f(X)  
    f(10, 20); // calls f(X, Y)  
    return 0;  
}
```



## Şabloane (Templates) în C++

### Functii generice

***Overload pe şabloane - ce funcție se apelează (ordinea de alegere)***

pas 1 potrivire FĂRĂ CONVERSIE

prioritate varianta non-template,  
apoi template fără parametri,  
apoi template cu 1 parametru ,  
apoi template cu mai mulți parametrii

pas 2 dacă nu există potrivire exactă

- conversie DOAR la varianta non-template



## Şabloane (Templates) în C++

### Funcţii generice

**Overload pe şabloane - ce funcţie se apelează (ordinea de alegere)**

**template <class T> void f(T t){ ... }**

**template <> void f(float x){ ... }**

**void f(float x){ .. }**

**int main()**

{     **f(1);** // T = int ('a');

**f(2.5);** // T=double;

**float x; f(x);** //non-template float , priorită de template<>, priorită de template T

**f<>(x);** // template<> priorită de template general cu T=float

**f<float>(x);** // template<> priorită de template general cu T=float

....

}



## Şabloane (Templates) în C++

### Functii generice

#### *Utilizarea parametrilor standard într-un template*

```
const int TABWIDTH = 8;  
// Display data at specified tab position.  
template<class X> void tabOut(X data, int tab) {  
    for(; tab; tab--)  
        for(int i=0; i<TABWIDTH; i++) cout << ' ';  
    cout << data << "\n";  
}
```

```
int main()  
{  
    tabOut("This is a test", 0);  
    tabOut(100, 1);  
    tabOut('X', 2);  
    tabOut(10/3, 3);  
    return 0;  
}
```



## Şabloane (Templates) în C++

### Clase generice

Şabloane pentru clase nu pentru funcţii.

Clasa conţine toţi algoritmii necesari să lucreze pe un anumit tip de date.

Din nou algoritmii pot fi generalizaţi, şabloni.

Specificăm tipul de date pe care lucrăm când obiectele din clasa respectivă sunt create.

Funcţiile membru ale unei clase generice sunt şi ele generice (în mod automat).

Nu e necesar să le specificăm cu template daca sunt definite in clasa. este necesar să le specificăm cu template dacă le definim în afara clasei.



## Şabloane (Templates) în C++

### Clase generice

Cozi, stive, liste înlăntuite, arbori de sortare

```
template <class Ttype> class class-name {  
...  
}
```

```
class-name <type> ob;
```

Ttype este tipul de date parametrizat.

Ttype este precizat când clasa e instanțiată.

Putem avea mai multe tipuri (separate prin virgulă)



## Şabloane (Templates) în C++

### Clase generice

```
template <class T>
```

```
class vector
```

```
{
```

```
    int dim;
```

```
    T v[100];
```

```
public:
```

```
    void citire();
```

```
    void afisare();
```

```
};
```

```
template <class T>
```

```
void vector<T>::citire()
```

```
{
```

```
    cin>>dim;
```

```
    for(int i = 0; i<dim; i++)
```

```
        cin>>v[i];
```

```
}
```

```
template <class T>
```

```
void vector<T>::afisare()
```

```
{
```

```
    for(int i = 0; i<dim; i++)
```

```
        cout<<v[i]<<" ";
```

```
    cout<<"\n";
```

```
}
```

```
int main()
```

```
{
```

```
    vector<int> ob1;
```

```
    ob1.citire();
```

```
    ob1.afisare();
```

```
    vector<float> ob2;
```

```
    ob2.citire();
```

```
    ob2.afisare();
```

```
    return 0;
```

```
}
```



## Şabloane (Templates) în C++

### Clase generice

*Mai multe tipuri de date generice intr-o clasă*

```
template <class Type1, class Type2> class myclass {
    Type1 i;
    Type2 j;
public:
    myclass(Type1 a, Type2 b) { i = a; j = b; }
    void show() { cout << i << ' ' << j << '\n'; }
};

int main()
{
    myclass<int, double> ob1(10, 0.23);
    myclass<char, char *> ob2('X', "Templates add power.");
    ob1.show(); // show int, double
    ob2.show(); // show char, char *
    return 0;
}
```



## Şabloane (Templates) în C++

### Clase generice

*Şabloanele se folosesc cu operatorii suprascrisi*

```
class student {  
    string nume;  
    float vârstă;  
} x,y;
```

```
template <class T> void maxim(T a, T b) {  
    if (a > b) cout<<"Primul este mai mare\n";  
    else cout<<"Al doilea este mai mare\n";  
}
```

```
int main()  
{  
    int a = 3, b = 7;  
    maxim(a,b); // ok  
    maxim(x,y); // operatorul > ar trebui definit în clasa student  
}
```



## Şabloane (Templates) în C++

### Clase generice

***Şabloanele se folosesc cu operatorii suprascrisi***

Se pot specifica și argumente valori în definirea claselor generalizate.

După “template” dăm tipurile parametrizate cât și “parametri normali” (ca la funcții).

Acești “param. normali” pot fi int, pointeri sau referințe; trebuie să fie cunoscuți la compilare: tratați ca și constante.

**template <class tip1, class tip2, int i>**



## Şabloane (Templates) în C++

### Clase generice

#### *Specializări explicite pentru clase*

La fel ca la şabloanele pentru funcţii

Se foloseşte **template <>**



## Şabloane (Templates) în C++

### Clase generice

#### Specializare de clasă

**template <class T>** // sau template <typename T>

**class Nume {**

    T x;

**public:**

**void set\_x(T a)**{x = a;}

**void afis()**{cout<<x;}

**}**

**template <>**

**class Nume<unsigned> {**

**unsigned x;**

**public:**

**void set\_x(unsigned a)**{x = a;}

**void afis()**{cout<<"\nUnsigned "<<x;}

**}**

**int main()**  
{

    Nume<int> m;  
    m.set\_x(7);  
    m.afis();  
    Nume<unsigned> n;  
    n.set\_x(100);  
    n.afis();  
    **return 0;**

}



## Şabloane (Templates) în C++

### Clase generice

#### **Argumente default și şabloane**

Putem avea valori default pentru tipurile parametrizate.

```
template <class X=int> class myclass { //...
```

Dacă instanțiem myclass fără să precizăm un tip de date atunci int este tipul de date folosit pentru şablon.

Este posibil să avem valori default și pentru argumentele valori (nu tipuri).



## Şabloane (Templates) în C++

### Clase generice

#### Argumente default și şabloane

```
template <class ATYPE=int, int size=10>
```

```
class atype {  
    ATYPE a[size]; // size of array is passed in size  
public:  
    atype();  
};
```

```
template <class ATYPE, int size>
```

```
atype<ATYPE,size>::atype() {           for(int i=0; i<size; i++) a[i] = i; }
```

```
int main()
```

```
{  
    atype<int, 100> intarray; // integer array, size 100  
    atype<double> doublearray; // double array, default size 10  
    atype<> defarray; // default to int array of size 10  
    return 0;  
}
```



## Şabloane (Templates) în C++

### Typeid și clasele template

Tipul unui obiect care este o instanță a unei clase template este determinat, în parte, de tipul datelor utilizate în cadrul datelor generice când obiectul este instantiat.

2 instanțe de tipuri diferite au fost create cu date diferite.

```
template <class T> class myclass
{
    T a;
public:
    myclass(T i)
    {
        a = i;
    }
    // ...
};
```



## Şabloane (Templates) în C++

### Typeid și clasele template

```
template <class T> class myclass{ ... };
```

```
int main()
```

```
{
```

```
    myclass<int> o1(10), o2(9);  
    myclass<double> o3(7.2);
```

```
    cout << "Type of o1 is " << typeid(o1).name() << endl;  
    cout << "Type of o2 is " << typeid(o2).name() << endl;  
    cout << "Type of o3 is " << typeid(o3).name() << endl;
```

```
    if(typeid(o1) == typeid(o2)) cout << "o1 and o2 are the same type\n";  
    if(typeid(o1) == typeid(o3)) cout << "Error\n";  
    else cout << "o1 and o3 are different types\n";
```

```
    return 0;
```

```
}
```



## Şabloane (Templates) în C++

### *Typeid și clasele template*

//Afisare compilator personal:

Type of o1 is 7myclassliE

Type of o2 is 7myclassliE

Type of o3 is 7myclassIdE

o1 and o2 are the same type

o1 and o3 are different types



## Şabloane (Templates) în C++

### *dynamic\_cast și clasele template*

Tipul unui obiect care este o instanță a unei clase template este determinat, în parte, de tipul datelor utilizate în cadrul datelor generice când obiectul este instantiat.

2 instanțe de tipuri diferite au fost create cu date diferite.

**template <class T> class myclass{ ... };**

myclass<int> și myclass<double> sunt 2 instanțe diferite.

**Nu se poate folosi dynamic\_cast pentru a schimba tipul unui pointer dintr-o instanță într-un pointer dintr-o instanță diferită.**

Ideal, folosim templates pentru polimorfism la compilare. Dynamic\_cast este la runtime și este mai costisitor.



## Şabloane (Templates) în C++

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

template <class T>
class Num
{
protected:    T val;
public:       Num(T x) { val = x; }
             virtual T getval( ) { return val; }
};

template <class T>
class SqrNum : public Num<T>
{
public: SqrNum(T x) : Num<T>(x) { }
        T getval( ) { return val * val; }
};
```

```
int main()
{
    Num<int> *bp, nob(2);
    SqrNum<int> *dp, sob(3);
    Num<double> dob(3.3);

    bp = dynamic_cast<Num<int>*> (&sob);
    if(bp)
    {
        cout << "Cast from SqrNum<int>* to
Num<int>* OK.\n";
        cout << "Value is " << bp->getval() << endl;
    }
    else
        cout << "Error\n";
    return 0;
}
```



## Şabloane (Templates) în C++

### **dynamic\_cast și clasele template**

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

template <class T>
class Num {
protected: T val;
public: Num(T x) { val = x; }
    virtual T getval( ) { return val; }
};

template <class T>
class SqrNum : public Num<T> {
public: SqrNum(T x) : Num<T>(x) {}
    T getval( ) { return val * val; }
};
```

```
int main()
{
    Num<int> *bp, nob(2);
    SqrNum<int> *dp, sob(3);
    Num<double> dob(3.3);

    dp = dynamic_cast<SqrNum<int> *> (&nob);
    if(dp) cout << "Error\n";
    else
    {
        cout << "Cast from Num<int>* to SqrNum<int>* not OK.\n";
        cout << "Can't cast a pointer to a base object into\n";
        cout << "a pointer to a derived object.\n";
    }
    return 0;
}
```



## Şabloane (Templates) în C++

### **dynamic\_cast și clasele template**

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

```
template <class T>
class Num {
protected: T val;
public: Num(T x) { val = x; }
    virtual T getval( ) { return val; }
};
```

```
template <class T>
class SqrNum : public Num<T> {
public: SqrNum(T x) : Num<T>(x) {}
    T getval( ) { return val * val; }
};
```

```
int main()
{
    Num<int> *bp, nob(2);
    SqrNum<int> *dp, sob(3);
    Num<double> dob(3.3);

    bp = dynamic_cast<Num<int> *> (&dob);
    if(bp)
        cout << "Error\n";
    else
        cout << "Can't cast from Num<double>* to Num<int>*.\\n";
        cout << "These are two different types.\\n";
    return 0;
}
```



## Perspective

Cursul 10:

1. Pointeri, Const, static in C++

Controlul tipului în timpul rulării programului în C++.

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

b Moștenire multiplă și identificatori de tip (dynamic\_cast, typeid).



# **Programare orientată pe obiecte**

## **- suport de curs -**

**Andrei Păun**  
**Anca Dobrovăț**

**An universitar 2021 – 2022**  
**Semestrul II**  
**Seriile 13, 14 și 15**

**Curs 10 & 11**



# Agenda cursului

Pointeri și referințe

Const

Volatile

Static



## Pointerii în C

- Recapitulare
- &, \*, array
- Operații pe pointeri: =, ++, --, +int, -
- Pointeri către pointeri, pointeri către functii
- Alocare dinamică: malloc, free
- Diferențe cu C++



## Pointerii în 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;



## Operatori pe pointeri

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

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

```
j=&i;
```

```
*j=9;
```



```
#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 trebuie făcute cu schimbarea de tip



# Aritmetică pe pointeri

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



# pointeri și array-uri

- numele array-ului este pointer
- $\text{lista}[5]==*(\text{lista}+5)$
- array de pointeri, numele listei este un pointer către pointeri (dublă indirectare)
- $\text{int } **p;$  (dublă indirectare)



# alocare dinamică

- `void *malloc(size_t bytes);`
  - alocă în memorie dinamic bytes și întoarce pointer către zona respectivă

```
char *p;  
p=malloc(100);
```

întoarce null dacă alocarea nu s-a putut face  
pointer void\* este convertit AUTOMAT la orice tip



- diferența la C++: trebuie să se facă schimbare de tip dintre void\* în tip\*

```
p=(char *) malloc(100);
```

sizeof: a se folosi pentru portabilitate  
a se verifica dacă alocarea a fost fără eroare  
(dacă se întoarce null sau nu)

```
if (!p) ...
```



# eliberarea de memorie alocată dinamic

void free(void \*p);

unde p a fost alocat dinamic cu malloc()

a nu se folosi cu argumentul p invalid pentru că  
rezultă probleme cu lista de alocare dinamică



# C++: Array-uri de obiecte

- o clasă de un tip
- putem crea array-uri cu date de orice tip (inclusiv obiecte)
- se pot defini neinitializate sau initializate clasa lista[10];

sau

clasa lista[10]={1,2,3,4,5,6,7,8,9,0};

pentru cazul initializat dat avem nevoie de constructor care primește un parametru întreg.



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

class cl {
    int i;
public:
    cl(int j) { i=j; } // constructor
    int get_i() { return i; }
};

int main()
{
    cl ob[3] = {1, 2, 3}; // initializers
    int i;

    for(i=0; i<3; i++)
        cout << ob[i].get_i() << "\n";
    return 0;
}
```



- inițializare pentru constructori cu mai mulți parametri

```
clasa lista[3]={clasa(1,5), clasa(2,4), clasa(3,3)};
```



- pentru definirea listelor de obiecte neinitializate: constructor fără parametri
- dacă în program vrem și initializare și neinitializare: overload pe constructor (cu și fără parametri)



# pointeri către obiecte

- obiectele sunt în memorie
- putem avea pointeri către obiecte
- `&obiect;`
- accesarea membrilor unei clase:  
-> în loc de .



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

class cl {
    int i;
public:
    cl(int j) { i=j; }
    int get_i() { return i; }
};

int main()
{
    cl ob(88), *p;
    p = &ob; // get address of ob
    cout << p->get_i(); // use -> to call
    get_i()
    return 0;
}
```

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

class cl {
public:
    int i;
    cl(int j) { i=j; }
};

int main()
{
    cl ob(1);
    int *p;
    p = &ob.i; // get address of ob.i
    cout << *p; // access ob.i via p
    return 0;
}
```



- 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  
ieseim din verificările automate făcute de C++



# pointerul **this**

- orice funcție membru are pointerul **this** (definit ca argument implicit) care arată către obiectul asociat cu funcția respectivă
- (pointer către obiecte de tipul clasei)
- funcțiile prieten nu au pointerul this
- funcțiile statice nu au pointerul this



```
#include <iostream>
using namespace std;
class pwr {
    double b;
    int e;
    double val;
public:
    pwr(double base, int exp);
    double get_pwr() { return this->val; }
};
```

```
pwr::pwr(double base, int exp)
{
    this->b = base;
    this->e = exp;
    this->val = 1;
    if(exp==0) return;
    for( ; exp>0; exp--) this->val = this->val * this->b;
}
```

```
int main()
{
    pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
    cout << x.get_pwr() << " ";
    cout << y.get_pwr() << " ";
    cout << z.get_pwr() << "\n";
    return 0;
```





# pointeri către clase derivate

- clasa de bază B și clasa derivată D
- un pointer către B poate fi folosit și cu D;

B \*p, o(1);

D oo(2);

p=&o;

p=&oo;



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

class base {
    int i;
public:
    void set_i(int num) { i=num; }
    int get_i() { return i; }
};

class derived: public base {
    int j;
public:
    void set_j(int num) { j=num; }
    int get_j() { return j; }
};
```

```
int main()
{
    base *bp;
    derived d;
    bp = &d; // base pointer points to derived object
              // access derived object using base pointer
    bp->set_i(10);
    cout << bp->get_i() << " ";
    /* The following won't work. You can't access
       elements of a derived class using a base class
       pointer.

    bp->set_j(88); // error
    cout << bp->get_j(); // error
*/
    ((derived *)bp)->set_j(88);
    cout << ((derived *)bp)->get_j();

    return 0;
}
```



# pointeri către clase derivate

- de ce merge și pentru clase derivate?
  - pentru că acea clasă derivată funcționează ca și clasa de bază plus alte detalii
- aritmetică pe pointeri: nu funcționează dacă incrementăm un pointer către bază și suntem în clasa derivată
- se folosesc pentru polimorfism la execuție (funcții virtuale)



// This program contains an error.

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

class base {
    int i;
public:
    void set_i(int num) { i=num; }
    int get_i() { return i; }
};

class derived: public base {
    int j;
public:
    void set_j(int num) {j=num;}
    int get_j() {return j;}
};
```

```
int main()
{
    base *bp;
    derived d[2];
    bp = d;
    d[0].set_i(1);
    d[1].set_i(2);
    cout << bp->get_i() << " ";
    bp++; // relative to base, not derived
    cout << bp->get_i(); // garbage value
    displayed
    return 0;
}
```



# pointeri către membri în clase

- pointer către membru
- nu sunt pointeri normali (către un membru dintr-un obiect) ci specifică un offset în clasă
- nu putem să aplicăm . și ->
- se folosesc .\* și ->\*

# Facultatea de Matematică și Informatică

## Universitatea din București



```
#include <iostream>
using namespace std;
class cl { int val;
public:
    cl(int i) { val=i; }
    int val;
    int double_val() { return val+val; }
};

int main()
{
    int cl::*data; // data member pointer
    int (cl::*func)(); // function member pointer
    cl ob1(1), ob2(2); // create objects
    data = &cl::val; // get offset of val
    func = &cl::double_val; // get offset of
                           //double_val()
    cout << "Here are values: ";
    cout << ob1.*data << " " << ob2.*data << "\n";
    cout << "Here they are doubled: ";
    cout << (ob1.*func) () << " ";
    cout << (ob2.*func) () << "\n";
    return 0;
}
```

```
#include <iostream>
using namespace std;
class cl { int val;
public: cl(int i) { val=i; }
        int val;
        int double_val() { return val+val; }};

int main(){
    int cl::*data; // data member pointer
    int (cl::*func)(); // function member
                       //pointer
    cl ob1(1), ob2(2), *p1, *p2;
    p1 = &ob1; // access objects through a
               //pointer
    p2 = &ob2;
    data = &cl::val; // get offset of val
    func = &cl::double_val;
    cout << "Here are values: ";
    cout << p1->*data << " " << p2->*data;
    cout << "\n";
    cout << "Here they are doubled: ";
    cout << (p1->*func) () << " ";
    cout << (p2->*func) () << "\n";
    return 0;
}
```



```
int cl::*d;  
int *p;  
cl o;  
  
p = &o.val // this is address of a  
specific val  
d = &cl::val // this is offset of generic  
val
```

- pointeri la membri nu sunt folosiți decât rar  
în cazuri speciale



# parametri referință

- nou la C++
- la apel prin valoare se adaugă și apel prin referință la C++
- nu mai e nevoie să folosim pointeri pentru a simula apel prin referință, limbajul ne dă acest lucru
- sintaxa: în funcție & înaintea parametrului formal



```
// Manually: call-by-reference using a
pointer.
```

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

```
void neg(int *i);
```

```
int main()
{
```

```
    int x;
    x = 10;
    cout << x << " negated is ";
    neg(&x);
    cout << x << "\n";
    return 0;
}
```

```
void neg(int *i)
{
    *i = -*i;
}
```

```
// Use a reference parameter.
```

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

```
void neg(int &i); // i now a reference
```

```
int main()
{
```

```
    int x;
    x = 10;
    cout << x << " negated is ";
    neg(x); // no longer need the &
operator
    cout << x << "\n";
    return 0;
}
```

```
void neg(int &i)
```

```
{
    i = -i; // i is now a reference,
don't need *
}
```



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

void swap(int &i, int &j);

int main()
{
    int a, b, c, d;
    a = 1;           b = 2;           c = 3;           d = 4;
    cout << "a and b: " << a << " " << b << "\n";
    swap(a, b); // no & operator needed
    cout << "a and b: " << a << " " << b << "\n";
    cout << "c and d: " << c << " " << d << "\n";
    swap(c, d);
    cout << "c and d: " << c << " " << d << "\n";
    return 0;
}

void swap(int &i, int &j)
{
    int t;
    t = i; // no * operator needed
    i = j;
    j = t;
}
```



# referințe către obiecte

- dacă transmitem obiecte prin apel prin referință la funcții nu se mai creează noi obiecte temporare, se lucrează direct pe obiectul transmis ca parametru
- deci copy-constructorul și destructorul nu mai sunt apelate
- la fel și la întoarcerea din funcție a unei referințe



```
#include <iostream>
using namespace std;
class cl {
    int id;
public:
    int i;
    cl(int i);
    ~cl() { cout << "Destructing " << id << "\n"; }
    void neg(cl &o) { o.i = -o.i; } // no temporary created
};

cl::cl(int num)
{
    cout << "Constructing " << num << "\n";
    id = num;
}

int main()
{
    cl o(1);
    o.i = 10;
    o.neg(o);
    cout << o.i << "\n";
    return 0;
}
```

Constructing 1  
-10  
Destructing 1



# 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 atribuirii 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



# referințe independente

- nu e asociat cu apelurile de funcții
- se creează un alt nume pentru un obiect
- referințele independente trebuie initializate la definire pentru că ele nu se schimbă în timpul programului



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

int main()
{
    int a;
    int &ref = a; // independent reference
    a = 10;
    cout << a << " " << ref << "\n";
    ref = 100;                                10 10
    cout << a << " " << ref << "\n";          100 100
    int b = 19;                                19 19
    ref = b; // this puts b's value into a        18 18
    cout << a << " " << ref << "\n";
    ref--; // this decrements a
    // it does not affect what ref refers to
    cout << a << " " << ref << "\n";
    return 0;
}
```



# referințe către clase derivate

- putem avea referințe definite către clasa de bază și apelată funcția cu un obiect din clasa derivată
- exact ca la pointeri



# Alocare dinamică în C++

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



# operatorii new, delete

- new: alocă memorie și întoarce un pointer la începutul zonei respective
- delete: sterge zona respectivă de memorie

p= new tip;

delete p;

la eroare se “aruncă” excepția bad\_alloc din <new>



```
#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;
}
```



# alocare de array-uri

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

int main()
{
    int *p, i;
    try {
        p = new int [10]; // allocate 10 integer array
    } catch (bad_alloc xa) {
        cout << "Allocation Failure\n";
        return 1;
    }
    for(i=0; i<10; i++ )
        p[i] = i;
    for(i=0; i<10; i++)
        cout << p[i] << " ";
    delete [] p; // release the array
    return 0;
}
```



# alocare de obiecte

- cu new
- după creare, new întoarce un pointer către obiect
- după creare se execută constructorul obiectului
- când obiectul este șters din memorie (delete) se execută destructorul



# obiecte create dinamic cu constructori parametrizează

```
class balance { ... }

...

balance *p;
// this version uses an initializer
try {
    p = new balance (12387.87, "Ralph Wilson");
} catch (bad_alloc xa) {
    cout << "Allocation Failure\n";
    return 1;
}
```



- array-uri de obiecte alocate dinamic
  - nu se pot inițializa
  - trebuie să existe un constructor fără parametri
  - delete poate fi apelat pentru fiecare element din array



- new și delete sunt operatori
- pot fi suprascriși pentru o anumită clasă
- pentru argumente suplimentare există o formă specială
  - p\_var = new (lista\_argumente) tip;
- există forma nothrow pentru new: similar cu malloc:  
`p=new(nothrow) int[20]; // intoarce null la eroare`



# const și volatile

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



- `#define BUFSIZE 100` (tipic în C)
- erori subtile datorită substituirii de text
- BUFSIZE e mult mai bun decât “valori magice”
- nu are tip, se comportă ca o variabilă
- mai bine: `const int bufsize = 100;`



- acum compilatorul poate face calculele la început: “constant folding”: important pt. array: o expresie complicată e calculată la compilare
- `char buf[bufsize];`
- se poate face const pe: **char, int, float, double** și variantele lor
- se poate face const și pe obiecte



- const implică “*internal linkage*” adică e vizibilă numai în fișierul respectiv (la linkare)
- trebuie dată o valoare pentru elementul constant la declarare, singura excepție:  
**extern const int bufsize;**
- în mod normal compilatorul nu alocă spațiu pentru constante, dacă e declarat ca extern alocă spațiu (să poată fi accesat și din alte părți ale programului)



- pentru structuri complicate folosite cu const se alocă spațiu: nu se știe dacă se alocă sau nu spațiu și atunci const impune localizare (să nu existe coliziuni de nume)
- de aceea avem “*internal linkage*”



```
// 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 declarăm cu const
- dacă încercăm să o schimbăm primim eroare de compilare



- const poate elimina memorie și acces la memorie
- const pentru aggregate: aproape sigur compilatorul alocă memorie
- nu se pot folosi valorile la compilare



```
// Constants and aggregates
const int i[] = { 1, 2, 3, 4 };

//! float f[i[3]]; // Illegal

struct S { int i, j; };
const S s[] = { { 1, 2 }, { 3, 4 } };

//! double d[s[1].j]; // Illegal

int main() {}
```



# diferențe cu C

- const în C: o variabilă globală care nu se schimbă
- deci nu se poate considera ca valoare la compilare

```
const int bufsize = 100;  
char buf[bufsize];
```

eroare în C



- În C se poate declara cu  
**const int bufsize;**
- În C++ nu se poate aşa, trebuie extern:  
**extern const int bufsize;**
- diferența:
  - C external linkage
  - C++ internal linkage



- În C++ compilatorul încearcă să nu creeze spațiu pentru const-uri, dacă totuși se transmite către o funcție prin referință, extern etc. atunci se creează spațiu
- C++: const în afara tuturor funcțiilor: scopul ei este doar în fișierul respectiv: internal linkage,
- alți identificatori declarați în același loc (fara const)  
**EXTERNAL LINKAGE**



# pointeri const

- 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



# pointeri constanți

- 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+initializare



# const pointer catre const element

```
int d = 1;
```

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

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



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



- 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



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



# constante caractere

char\* cp = "howdy";

dacă se încearcă schimbarea caracterelor din  
“howdy” compilatorul ar trebui să genereze  
eroare; nu se întâmplă în mod ușual  
(compatibilitate cu C)

mai bine: char cp[] = "howdy";  
și atunci nu ar mai trebui să fie probleme



# 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



```
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  
}
```



```
// 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!
}
```



```
// Constant return by value
// Result cannot be used as an lvalue

class X {    int i;
public:    X(int ii = 0);
void modify();
};

X::X(int ii) { i = ii; }
void X::modify() { i++; }

X f5() {    return X(); }
const X f6() {    return X(); }

void f7(X& x) { // Pass by non-const
reference
    x.modify();
}

int main() {
    f5() = X(1); // OK -- non-const return
value
    f5().modify(); // OK
    // Causes compile-time errors:
    //! f7(f5());
    //! f6() = X(1);
    //! f6().modify();
    //! f7(f6());
} ///:~
```



- f7() creează obiecte temporare, de aceea nu compilează
- aceste obiecte au constructor și destructor dar pentru că nu putem să le “atingem” sunt definite sub forma de obiecte constante
- f7(f5()); se creează ob. temporar pentru rezultatul lui f5(); și apoi apel prin referință la f7
- ca să compileze (dar cu erori mai târziu) trebuie apel prin referință const



- $f5() = X(1);$
- $f5().modify();$
- compilează fără probleme, dar procesarea se face pe obiectul temporar (modificările se pierd imediat, deci aproape sigur este bug)



# parametrii de intrare și iesire: adrese

- e preferabil să fie definiți ca const
- în felul acesta pointerii și referințele nu pot fi modificați/modificate



```
// Constant pointer arg/return

void t(int* ) {}

void u(const int* cip) {
//!  *cip = 2; // Illegal -- modifies value
    int i = *cip; // OK -- copies value
//!  int* ip2 = cip; // Illegal: non-const
}

const char* v() {
    // Returns address of static character
    // array:
    return "result of function v()";
}

const int* const w() {
    static int i;
    return &i;
}
```

```
int main() {
    int x = 0;
    int* ip = &x;
    const int* cip = &x;
    t(ip); // OK
//!  t(cip); // Not OK
    u(ip); // OK
    u(cip); // Also OK
//!  char* cp = v(); // Not OK
    const char* ccp = v(); // OK
//!  int* ip2 = w(); // Not OK
    const int* const ccip = w(); // OK
    const int* cip2 = w(); // OK
//!  *w() = 1; // Not OK
} //://:~
```

- cip2 nu schimbă adresa întoarsă din w (pointerul constant care arată spre constantă) deci e ok; următoarea linie schimbă valoarea deci compilatorul intervine



# comparații cu C

- În C dacă vrem param. o adresa: se face pointer la pointer
- În C++ nu se încurajează acest lucru: const referință
- pentru apelant e la fel ca apel prin valoare
  - nici nu trebuie să se gândească la pointeri
  - trimiterea unei adrese e mult mai eficientă decât transmiterea obiectului prin stivă, se face const deci nici nu se modifică



# Ob. temporare sunt const

```
//: C08:ConstTemporary.cpp
// Temporaries are const

class X {};

X f() { return X(); } // Return by value

void g1(X&) {} // Pass by non-const reference
void g2(const X&) {} // Pass by const reference

int main() {
    // Error: const temporary created by f():
    //! g1(f());
    // OK: g2 takes a const reference:
    g2(f());
} // :~
```

- În C avem pointeri  
deci e OK



# Const în clase

- const pentru variabile de instanță și
  - funcții de instanță de tip const
- 
- să construim un vector pentru clasa respectivă, în C folosim #define
  - problemă în C: coliziune pe nume



- În C++: punem o variabilă de instanță const
- problemă: toate obiectele au această variabilă, și putem avea chiar valori diferite (depinde de inițializare)
- când se creează un const într-o clasă nu se poate inițializa (constructorul inițializează)
- În constructor trebuie să fie deja inițializat (altfel am putea să îl schimbăm în constructor)



- inițializare de variabile const în obiecte: lista de inițializare a constructorilor

```
// Initializing const in classes
#include <iostream>
using namespace std;

class Fred {
    const int size;
public:
    Fred(int sz);
    void print();
};

Fred::Fred(int sz) : size(sz) {}
void Fred::print() { cout << size << endl; }

int main() {
    Fred a(1), b(2), c(3);
    a.print(), b.print(), c.print();
} // :~
```



# rezolvarea problemei inițiale

- cu static
  - inseamnă că nu e decât un singur asemenea element în clasă
  - îl facem static const și devine similar ca un const la compilare
  - static const trebuie inițializat la declarare (nu în constructor)

# Facultatea de Matematică și Informatică

## Universitatea din București



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

class StringStack {
    static const int size = 100;
    const string* stack[size];
    int index;
public:
    StringStack();
    void push(const string* s);
    const string* pop();
};

StringStack::StringStack() : index(0) {
    memset(stack, 0, size * sizeof(string*));
}

void StringStack::push(const string* s) {
    if(index < size)
        stack[index++] = s;
}

const string* StringStack::pop() {
    if(index > 0) {
        const string* rv = stack[--index];
        stack[index] = 0;
        return rv;
    }
    return 0;
}
```

```
string iceCream[] = {
    "pralines & cream",
    "fudge ripple",
    "jamocha almond fudge",
    "wild mountain blackberry",
    "raspberry sorbet",
    "lemon swirl",
    "rocky road",
    "deep chocolate fudge"
};

const int iCsz =
    sizeof iceCream / sizeof *iceCream;

int main() {
    StringStack ss;
    for(int i = 0; i < iCsz; i++)
        ss.push(&iceCream[i]);
    const string* cp;
    while((cp = ss.pop()) != 0)
        cout << *cp << endl;
}
```



# enum hack

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

class Bunch {
    enum { size = 1000 };
    int i[size];
};

int main() {
    cout << "sizeof(Bunch) = " << sizeof(Bunch)
        << ", sizeof(i[1000]) = "
        << sizeof(int[1000]) << endl;
}
```

- în cod vechi
- a nu se folosi cu C++ modern
- static const int size=1000;



# obiecte const și funcții membru const

- obiecte const: nu se schimbă
- pentru a se asigura că starea obiectului nu se schimbă funcțiile de instanță apelabile trebuie definite cu const
- declararea unei funcții cu const nu garantează că nu modifică starea obiectului!



# functii membru const

- compilatorul și linkerul cunosc faptul că funcția este const
- se verifică acest lucru la compilare
- nu se pot modifica partii ale obiectului în aceste funcții
- nu se pot apela funcții non-const



```
//: C08:ConstMember.cpp
class X {
    int i;
public:
    X(int ii);
    int f() const;
};

X::X(int ii) : i(ii) {}
int X::f() const { return i; }

int main() {
    X x1(10);
    const X x2(20);
    x1.f();
    x2.f();
} ///:~
```



- toate funcțiile care nu modifică date să fie declarate cu const
- ar trebui ca “default” funcțiile membru să fie de tip const

# Facultatea de Matematică și Informatică

## Universitatea din București



```
#include <iostream>
#include <cstdlib> // Random number generator
#include <ctime> // To seed random generator

using namespace std;

class Quoter {
    int lastquote;
public:
    Quoter();
    int lastQuote() const;
    const char* quote();
};

Quoter::Quoter() {    lastquote = -1;
    srand(time(0)); // Seed random number
generator
}

int Quoter::lastQuote() const {    return
lastquote;}
```

```
const char* Quoter::quote() {
    static const char* quotes[] = {
        "Are we having fun yet?",
        "Doctors always know best",
        "Is it ... Atomic?",
        "Fear is obscene",
        "There is no scientific evidence "
        "to support the idea "
        "that life is serious",
        "Things that make us happy, make us wise",
    };
    const int qsize = sizeof quotes/sizeof *quotes;
    int qnum = rand() % qsize;
    while(lastquote >= 0 && qnum == lastquote)
        qnum = rand() % qsize;
    return quotes[lastquote = qnum];
}

int main() {
    Quoter q;
    const Quoter cq;
    cq.lastQuote(); // OK
//! cq.quote(); // Not OK; non const function
    for(int i = 0; i < 20; i++)
        cout << q.quote() << endl;
} // :~
```



# schimbări în obiect din funcții const

- “casting away constness”
- se face castare a pointerului `this` la pointer către tipul de obiect
- pentru că în funcții const este de tip clasa `const*`
- după această schimbare de tip se modifica prin pointerul `this`



```
// "Casting away" constness

class Y {
    int i;
public:
    Y();
    void f() const;
};

Y::Y() { i = 0; }

void Y::f() const {
//! i++; // Error -- const member function
((Y*)this)>>i++; // OK: cast away const-
ness
// Better: use C++ explicit cast syntax:
(const_cast<Y*>(this))>>i++;
}

int main() {
    const Y yy;
    yy.f(); // Actually changes it!
} // :~
```



- apare în cod vechi
- nu e ok pentru că funcția modifică și noi credem că nu modifică
- o metodă mai bună: în continuare



```
// The "mutable" keyword

class Z {
    int i;
    mutable int j;
public:
    Z();
    void f() const;
};

Z::Z() : i(0), j(0) {}

void Z::f() const {
    //! i++; // Error -- const member function
    j++; // OK: mutable
}

int main() {
    const Z zz;
    zz.f(); // Actually changes it!
} // :~
```



# volatile

- e similar cu const
- obiectul se poate schimba din afara programului
- multitasking, multithreading, întreruperi
- nu se fac optimizări de cod
- avem obiecte volatile, funcții volatile, etc.



# static

- ceva care își tine poziția neschimbătă
- alocare statică pentru variabile
- vizibilitate locală a unui nume



- variabile locale statice
- își mențin valorile intre apelări
- inițializare la primul apel



```
#include "../require.h"
#include <iostream>
using namespace std;

char oneChar(const char* charArray = 0) {
    static const char* s;
    if(charArray) {
        s = charArray;
        return *s;
    }
    else
        require(s, "un-initialized s");
    if(*s == '\0')
        return 0;
    return *s++;
}

char* a = "abcdefghijklmnopqrstuvwxyz";

int main() {
    // oneChar(); // require() fails
    oneChar(a); // Initializes s to a
    char c;
    while((c = oneChar()) != 0)
        cout << c << endl;
} ///:~
```



# obiecte statice

- la fel ca la tipurile predefinite
- avem nevoie de constructorul predefinit



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

class X {
    int i;
public:
    X(int ii = 0) : i(ii) {} // Default
    ~X() { cout << "X::~X()" << endl; }
};

void f() {
    static X x1(47);
    static X x2; // Default constructor
    required
}

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



# destructori statici

- când se termină main se distrug obiectele
- De obicei se cheamă exit() la ieșirea din main
- dacă se cheamă exit() din destructor e posibil să avem ciclu infinit de apeluri la exit()
- destructorii statici nu sunt execuți dacă se iese prin abort()



- dacă avem o funcție cu obiect local static
- și funcția nu a fost apelată, nu vrem să apelam destructorul pentru obiect neconstruit
- C++ ține minte care obiecte au fost construite și care nu

# Facultatea de Matematică și Informatică

## Universitatea din București



```
#include <fstream>
using namespace std;
ofstream out("statdest.out"); // Trace file

class Obj {
    char c; // Identifier
public:
    Obj(char cc) : c(cc) {
        out << "Obj::Obj() for " << c << endl;
    }
    ~Obj() {
        out << "Obj::~Obj() for " << c << endl;
    }
};

Obj a('a'); // Global (static storage)
// Constructor & destructor always called

void f() {
    static Obj b('b');
}

void g() {
    static Obj c('c');
}

int main() {
    out << "inside main()" << endl;
    f(); // Calls static constructor for b
    // g() not called
    out << "leaving main()" << endl;
} //://:~
```

Obj::Obj() for a  
inside main()  
Obj::Obj() for b  
leaving main()  
Obj::~Obj() for b  
Obj::~Obj() for a



# static pentru nume (la linkare)

- orice nume care nu este într-o clasă sau funcție este vizibil în celealte parti ale programului (external linkage)
- dacă e definit ca static are internal linkage: vizibil doar în fișierul respectiv
- linkarea e valabilă pentru elemente care au adresă (clase, var. locale nu au)



- int a=0;
- în afara claselor, funcțiilor: este var globală, vizibilă pretutindeni
- similar cu: extern int a=0;
- static int a=0; // internal linkage
- nu mai e vizibilă pretutindeni, doar local în fișierul respectiv



```
//{L} LocalExtern2
#include <iostream>

int main() {
    extern int i;
    std::cout << i;
} ///:~

//: C10:LocalExtern2.cpp {O}
int i = 5;
///:~
```



# funcții extern și static

- schimbă doar vizibilitatea
- void f(); similar cu extern void f();
- restrictiv:
  - static void f();



- alți specificatori:
  - auto: aproape nefolosit; spune ca e var. locală
  - register: să se pună într-un registru



# variabile de instanță statice

- când vrem să avem valori comune pentru toate obiectele
- static

```
class A {  
    static int i;  
public:  
    //...  
};  
int A::i = 1;
```

- int A::i = 1;
- se face o singura dată
- e obligatoriu să fie făcut de creatorul clasei, deci e ok



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

int x = 100;

class WithStatic {
    static int x;
    static int y;
public:
    void print() const {
        cout << "WithStatic::x = " << x << endl;
        cout << "WithStatic::y = " << y << endl;
    }
};

int WithStatic::x = 1;
int WithStatic::y = x + 1;
// WithStatic::x NOT ::x

int main() {
    WithStatic ws;
    ws.print();
}
```



```
// Static members & local classes
#include <iostream>
using namespace std;

// Nested class CAN have static data members:
class Outer {
    class Inner {
        static int i; // OK
    }
}

int Outer::Inner::i = 47;

// Local class cannot have static data members:
void f() {
    class Local {
        public:
        //! static int i; // Error
        // (How would you define i?)
    } x;
}

int main() { Outer x; f(); } // :~
```



# funcții membru statice

- nu sunt asociate cu un obiect, nu au this

```
class X {  
public:  
    static void f(){};  
};  
  
int main() {  
    X::f();  
} // :~
```



# Despre examen

- Descrieți pe scurt funcțiile şablon (template).

```
#include <iostream.h>
class problema
{ int i;
public: problema(int j=5){i=j;}
    void schimba(){i++;}
    void afiseaza(){cout<<"starea
currenta "<<i<<"\n";}
};
problema mister1() { return problema(6);}
void mister2(problema &o)
{ o.afiseaza();
  o.schimba();
  o.afiseaza();
}
int main()
{ mister2(mister1());
  return 0;
}
```



```
#include<iostream.h>
class B
{ int i;
public: B() { i=1; }
         virtual int get_i() { return i; }
};
class D: virtual public B
{ int j;
public: D() { j=2; }
         int get_i() {return B::get_i()+j; }
};
class D2: virtual public B
{ int j2;
public: D2() { j2=3; }
         int get_i() {return B::get_i()+j2; } };

class MM: public D, public D2
{ int x;
public: MM() { x=D::get_i()+D2::get_i(); }
         int get_i() {return x; } };

int main()
{ B *o= new MM();
cout<<o->get_i()<<"\n";
MM *p= dynamic_cast<MM*>(o);
if (p) cout<<p->get_i()<<"\n";
D *p2= dynamic_cast<D*>(o);
if (p2) cout<<p2->get_i()<<"\n";
return 0;
}
```



```
#include <iostream.h>
#include <typeinfo>
class B
{ int i;
public: B() { i=1; }
    int get_i() { return i; }
};

class D: B
{ int j;
public: D() { j=2; }
    int get_j() {return j; }
};

int main()
{ B *p=new D;
cout<<p->get_i();
if (typeid((B*)p).name()=="D*")
cout<<((D*)p)->get_j();
return 0;
}
```



```
#include<iostream.h>
template<class T, class U>
T f(T x, U y)
{ return x+y;
}
int f(int x, int y)
{ return x-y;
}
int main()
{ int *a=new int(3), b(23);
cout<<*f(a,b);
return 0;
}
```



```
#include<iostream.h>
class A
{ int x;
public: A(int i=0) { x=i; }
        A operator+(const A& a) { return x+a.x; }
        template <class T> ostream& operator<<(ostream&); }
template <class T>
ostream& A::operator<<(ostream& o) { o<<x; return o; }
int main()
{ A a1(33), a2(-21);
cout<<a1+a2;
return 0;
}
```



## Perspective

Cursul 12:

Biblioteca Standard Template Library - STL

- Containere, iteratori și algoritmi.
- Clasele vector, list, map / multimap.
- Elemente avansate



# **Programare orientată pe obiecte**

## **- suport de curs -**

**Andrei Păun**  
**Anca Dobrovăț**

**An universitar 2021 – 2022**  
**Semestrul II**  
**Seriile 13, 14 și 15**

**Curs 12**



# Agenda cursului

Biblioteca Standard Template Library - STL

- Containere, iteratori și algoritmi.
- Clasele vector, list, map / multimap.
- Elemente avansate



## Standard Template Library (STL)

-bibliotecă de clase C++, parte din Standard Library

### Oferă:

- structuri de date și algoritmi fundamentali → dezvoltarea de programe în C++;
- componente generice, parametrizabile. Aproape toate clasele din STL sunt parametrizate (Template).

Componentele STL se pot compune cu usurință fără a sacrifica performanță (generic programming)

STL conține clase pentru:

- **containere**
- **iteratori**
- **algoritmi**
- functori (function objects)
- allocators



## Standard Template Library (STL)

### Container

**“Containers are the STL objects that actually store data” (H. Schildt)**

- grupare de date în care se pot adauga (insera) și din care se pot sterge (extragă) obiecte;
- gestionează memoria necesară stocării elementelor, oferă metode de acces la elemente (direct și prin iteratori);
- **funcționalități (metode):**
  - accesare elemente (ex.: [ ])
  - gestiune capacitate (ex.: size())
  - modificare elemente (ex.: insert, clear)
  - iterator (begin(), end())
  - alte operații (ie: find)



## Standard Template Library (STL)

### Container

#### Tipuri de containtere:

- **de tip secventa** (in terminologia STL, o secventa este o lista liniara):
  - vector
  - deque
  - list
- **asociativi** (permit regasirea eficienta a informatiilor bazandu-se pe chei)
  - set
  - multiset
  - map (permite accesul la informatii cu cheie unica)
  - multimap
- **adaptor de containere**
  - stack
  - queue
  - priority\_queue



## Standard Template Library (STL)

### Adaugarea si stergerea elementelor din containtere:

De **tip secenta** (vector, deque, list) si - **asociativi** (set, multiset, map, multimap):

- insert()
- erase()

De **tip secenta** (vector, deque, list) permit si:

- push\_back()
- pop\_back()

### List , Deque

- pop\_front()
- push\_front()

### Containerele asociative:

- find()

Accesarea uzuala: prin iteratori.



## Standard Template Library (STL)

### Containere de tip secvențial

#### Vector (Dynamic Array)

Vector, Deque, List sunt containere de tip secvențial, folosesc reprezentări interne diferite, astfel operațiile uzuale au complexități diferite.

***template <class T, class Allocator = allocator <T> > class vector***

T = tipul de date utilizat

Allocator = tipul de alocator utilizat (in general cel standard).

- elementele sunt stocate secvențial în zone continue de memorie.

Vector are performanțe bune la:

- Acesare elemente individuale de pe o poziție dată (constant time).
- Iterare elemente în orice ordine (linear time).
- Adăugare/Ștergere elemente de la sfârșit (constant amortized time).



## Standard Template Library (STL)

### Containere de tip secentă

#### Vector (Dynamic Array)

Constructori:

**explicit vector(const Allocator &a = Allocator( ));**

// expl. **vector<int> iv** - vector de int cu zero elemente;

**explicit vector(size\_type num, const T &val = T( ), const Allocator &a = Allocator( ));**

// expl. **vector<int> iv(10)** - vector de int cu 10 elemente;

// expl. **vector<int> iv(10,7)** - vector de int cu 10 elemente, fiecare egal cu 7;

**vector(const vector<T,Allocator> &ob);**

// expl. **vector<int> iv2(iv)** - vector de int reprezentând copia lui iv;

**template <class InIter>**

**vector(InIter start, InIter end, const Allocator &a = Allocator( ));**



## Standard Template Library (STL)

### Containere de tip secentă Vector (Dynamic Array)

#### Functii membre uzuale (sursa H. Schildt)

- **back( )**: returneaza o referinta catre ultimul element;
- **front( )**: returneaza o referinta catre primul element;
- **begin( )**: returneaza un iterator catre primul element;
- **end( )**: returneaza un iterator catre zona de memorie de dupa ultimul element;
- **clear( )**: elimina toate elementele din vector.
- **empty( )**: “true(false)” daca vectorul e (sau nu) gol.
- **erase(iterator i)**: stergerea elementului pointat de i; returneaza un iterator catre elementul de dupa cel sters.
- **erase(iterator start, iterator end)**: stergerea elementelor intre start si end.



## Standard Template Library (STL)

### Containere de tip secentă Vector (Dynamic Array)

#### Functii membre uzuale (sursa H. Schildt)

- **insert(iterator i, const T &val)**: insereaza val inaintea elementului pointat;
- **insert(iterator i, size\_type num, const T & val)** : insereaza un numar de “num” elemente de valoare “val” inaintea elementului pointat de i.
- **insert(iterator i, Iterator start, Iterator end)**: insereaza o secentă start-end inaintea elementului pointat de i.
- **operator[ ](size\_type i)** : returneaza o referinta la elementul specificat de i;
- **pop\_back()**: sterge ultimul element.
- **push\_back(const T &val)**: adauga la final valoarea “val”
- **size()** : dimensiunea vectorului.



## Standard Template Library (STL)

### Containere de tip secentă

#### Vector (Dynamic Array)

**Exemplu citirea unui vector cu accesarea elementelor prin iterator si prin operatorul [ ]**

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v(3);
    v[0] = 12;
    v[1] = 34;
    v[2] = 56;
    //v.resize(5); // necesar pentru reactualizarea dimensiunii vectorului
    v[3] = 78;
    v[4] = 90;
    for(int i = 0; i<v.size() ; i++)
        cout<<v[i]<<" ";
}
```



## Standard Template Library (STL)

### Containere de tip secentă

#### Vector (Dynamic Array)

**Exemplu citirea unui vector cu accesarea elementelor prin iterator si prin operatorul [ ]**

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<int> v(3);
    v.push_back(12);
    v.push_back(34);
    v.push_back(56);
    v.push_back(78);
    v.push_back(90);
    for(int i = 0; i<v.size(); i++)
        cout<<v[i]<<" "; // Afisare 8 valori, resize automat
}
```



## Standard Template Library (STL)

### Containere de tip secentă Vector (Dynamic Array)

#### Exemplu

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<char> v(10);
    unsigned int i;
    cout << "Size = " << v.size() << endl;
    for(i=0; i<10; i++) v[i] = i + 'a';
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;
    for(i=0; i<10; i++) v.push_back(i + 10 + 'a');
    cout << "Size now = " << v.size() << endl;
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
    cout << endl;
    for(i=0; i<v.size(); i++) v[i] = toupper(v[i]);
    for(i=0; i<v.size(); i++) cout << v[i] << " ";
}
```

```
Size = 10
a b c d e f g h i j
Size now = 20
a b c d e f g h i j k l m n o p q r s t
A B C D E F G H I J K L M N O P Q R S T
```



## Standard Template Library (STL)

### Containere de tip secentă

#### Vector (Dynamic Array)

### Exemplu - accesarea unui vector cu iterator

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    vector<char> v(10); //unsigned int i = 0;
    vector<char>::iterator p;

    for( p = v.begin(); p != v.end(); p++, i++)
        *p = i + 'a';

    for( p = v.begin(); p != v.end(); p++)
        cout << *p << " ";

    for( p = v.begin(); p != v.end(); p++)
        *p = toupper(*p);
}
```



## Standard Template Library (STL)

### Containere de tip secentă

#### Vector (Dynamic Array)

### Exemplu - inserarea si stergerea elementelor intr-un vector

```
vector<int> v(3);
v[0] = 12;      v[1] = 34;      v[2] = 56; // 12,34,56
v.resize(4);
v[3] = 78;
v.push_back(90); // 12, 34, 56, 78, 90

vector<int>::iterator p,pp;
p = v.begin();
p++;
v.insert(p,3,100); // 12, 100, 100, 100, 34, 56, 78, 90

p = v.begin();
v.erase(p); // 100, 100, 100, 34, 56, 78, 90

p = v.begin() + 3;
v.erase(p,p+2); // 100, 100, 100, 78, 90
```



## Standard Template Library (STL)

### Containere de tip secentă

#### Vector (Dynamic Array)

### Exemplu - inserarea elementelor de tip definit de utilizator

```
class Test { int i;
public:
    Test(int x = 0) :i(x) {cout<<"C ";}
    Test(const Test& x) {i = x.i; cout<<"CC ";}
    ~Test() {cout<<"D ";}
};

int main() {
    vector<Test> v;
    v.push_back(10); cout<<endl;      /// C CC D
    v.push_back(20); cout<<endl;      /// C CC CC D D
    Test ob(30);
    v.push_back(ob); cout<<endl;      /// C CC CC CC D D
    Test& ob2 = ob;
    v.push_back(ob2);     /// CC D D D D
}
```



## Standard Template Library (STL)

### Containere de tip secentă List

***template <class T, class Allocator = allocator <T> > class list***

T = tipul de date utilizat

Allocator = tipul de alocator utilizat (in general cel standard).

- implementat ca și listă dublu înlănțuită

List are performanțe bune la:

- Ștergere/adăugare de elemente pe orice poziție (constant time).
- Mutarea de elemente sau secvențe de elemente în liste sau chiar și intre liste diferite(constant time).
- Iterare de elemente in ordine (linear time)



## Standard Template Library (STL)

### Containere de tip secenta List

Constructori:

```
explicit list(const Allocator &a = Allocator( ));  
// expl. list<int> iv
```

```
explicit list(size_type num, const T &val = T( ), const Allocator &a =  
Allocator());  
// expl. list<int> iv(10);  
// expl. list<int> iv(10,7);
```

```
list(const list <T,Allocator> &ob);  
// expl. list<int> iv2(iv);
```

```
template <class InIter>  
list(InIter start, InIter end, const Allocator &a = Allocator( ));
```



## Standard Template Library (STL)

### Containere de tip secenta List

#### Functii membre uzuale (sursa H. Schildt)

- **back( )**: returneaza o referinta catre ultimul element;
- **front( )**: returneaza o referinta catre primul element;
- **begin( )**: returneaza un iterator catre primul element;
- **end( )**: returneaza un iterator catre zona de memorie de dupa ultimul element;
- **clear( )**: elimina toate elementele din vector.
- **empty( )**: “true(false)” daca vectorul e (sau nu) gol.
- **erase(iterator i)**: stergerea elementului pointat de i; returneaza un iterator catre elementul de dupa cel sters.
- **erase(iterator start, iterator end)**: stergerea elementelor intre start si end.



## Standard Template Library (STL)

### Containere de tip secenta List

#### Functii membre uzuale (sursa H. Schildt)

- **insert(iterator i, const T &val)**: insereaza val inaintea elementului pointat;
- **insert(iterator i, size\_type num, const T & val)** : insereaza un numar de “num” elemente de valoare “val” inaintea elementului pointat de i.
- insert(iterator i, InIter start, InIter end)**: insereaza o secenta start-end inaintea elementului pointat de i.
- **merge(list <T, Allocator> &ob)**: concateneaza lista din ob; aceasta din urma devine vida.
- **merge(list &ob, Comp cmpfn)**: concateneaza si sorteaza;



## Standard Template Library (STL)

### Containere de tip secenta List

#### Functii membre uzuale (sursa H. Schildt)

- **pop\_back( )**: sterge ultimul element.
- **pop\_front( )**: sterge primul element.
- **push\_back(const T &val)**: adauga la final valoarea “val”
- **push\_front(const T &val)**: adauga la inceput valoarea “val”
- **remove(const T &val)**: elimina toate valorile “val” din lista;
- **reverse( )**: inverseaza lista.
- **size( )** : numarul de elemente.
- **sort( )**: ordoneaza crescator
- **sort(Comp cmpfn)**: - sorteaza cu o functie de comparatie.



## Standard Template Library (STL)

### Containere de tip secentă List

#### Exemplu

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<int> lst; // create an empty list
    int i;
    for(i=0; i<10; i++) lst.push_back(i);
    cout << "Size = " << lst.size() << endl;
    list<int>::iterator p;
    for(p = lst.begin(); p != lst.end(); p++)
        cout << *p << " ";
    for(p = lst.begin(); p != lst.end(); p++)
        *p = *p + 100;
    for(p = lst.begin(); p != lst.end(); p++)
        cout << *p << " ";
    return 0;
}
```



## Standard Template Library (STL)

### Containere de tip secentă List

#### Exemplu

```
list<int> lst; // lista vida
for(int i=0; i<10; i++) lst.push_back(i); // insereaza 0 .. 9

// Lista afisata in ordine inversa - mod 1"
list<int>::iterator p;
p = lst.end();
while(p != lst.begin()) {
    p--; // decrement pointer before using
cout << *p << " ";
}
```

```
// Lista afisata in ordine inversa - mod 2"
list<int>::reverse_iterator q;
for(q = lst.rbegin(); q!=lst.rend();q++)
cout<<*q<<" ";
```



## Standard Template Library (STL)

### Containere de tip secentă List

#### Exemplu - push\_front, push\_back și sort

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list lst1, lst2;
    int i;
    for(i=0; i<10; i++) lst1.push_back(i);
    for(i=0; i<10; i++) lst2.push_front(i);

    list::iterator p;
    for( p = lst1.begin(); p != lst1.end(); p++) cout << *p << " ";
    for( p = lst2.begin(); p != lst2.end(); p++) cout << *p << " ";

    // sort the list
    lst1.sort();
}
```



## Standard Template Library (STL)

### Containere de tip secentă

#### List

### Exemplu - ordonare crescătoare și descrescătoare

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

bool compare(int a, int b) {return a>b;}

int main() {
    list<int> l;
    l.push_back(5); l.push_back(3); l.push_front(2);
    list<int>::iterator p;
    l.sort(); //crescator
    for(p=l.begin(); p!=l.end();p++) cout<<*p<<" ";
}

l.sort(compare); // descrescator prin functia comparator
for(p=l.begin(); p!=l.end();p++) cout<<*p<<" ";
}
```



## Standard Template Library (STL)

### Containere de tip secentă List

#### Exemplu - concatenarea a 2 liste

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
    list<int> lst1, lst2; /*...creare liste ...*/
    list<int>::iterator p;
// concatenare
    lst1.merge(lst2);
    if(lst2.empty())
        cout << "lst2 vida\n";
    cout << "Contents of lst1 after merge:\n";
    for(p = lst1.begin(); p != lst1.end(); p++) cout << *p << " ";
    return 0;
}
```



## Standard Template Library (STL)

### Containere de tip secenta List

#### Exemplu - stocarea obiectelor in list

```
class Test { int i;
public:
    Test(int x = 0) : i(x) { cout<<"C " ; }
    Test(const Test& x) { i = x.i; cout<<"CC " ; }
    ~Test() { cout<<"D " ; } }

int main() {
    list<Test> v;
    v.push_back(10); cout<<endl;      /// C CC D
    v.push_back(20); cout<<endl;      /// C CC D
    Test ob(30);
    v.push_back(ob); cout<<endl;      /// C CC
    Test& ob2 = ob;
    v.push_back(ob2);     /// CC D D D D D
}
```



## Standard Template Library (STL)

### Containere de tip secenta List

#### Exemplu - stocarea obiectelor in list

```
#include <iostream>
#include <list>
#include <cstring>
using namespace std;
class myclass
{
    int a, b, sum;
public:
    myclass() {a = b = 0;}
    myclass(int i, int j) {a = i; b = j; sum = a + b;}
    int getsum() {return sum;}
    friend bool operator<(myclass &o1, myclass &o2) {return o1.sum < o2.sum;}
    friend bool operator>(myclass &o1, myclass &o2) {return o1.sum > o2.sum;}
    friend bool operator==(myclass &o1, myclass &o2) {return o1.sum == o2.sum;}
    friend bool operator!=(myclass &o1, myclass &o2) {return o1.sum != o2.sum;}
};
```



## Standard Template Library (STL)

### Containere de tip secentă List

#### Exemplu - stocarea obiectelor în list

```
int main() {
    int i;
    list<myclass> lst1;
    list<myclass>::iterator p;

    for(i=0; i<10; i++) lst1.push_back(myclass(i, i));
    for (p = lst1.begin(); p != lst1.end(); p++) cout<< p->getsum()<< " ";
    cout << endl;
// create a second list
    list<myclass> lst2;
    for(i=0; i<10; i++) lst2.push_back(myclass(i*2, i*3));
    for (p = lst2.begin(); p != lst2.end(); p++) cout<< p->getsum()<< " ";
    cout << endl;
// now, merge lst1 and lst2
    lst1.merge(lst2);
    return 0;
}
```



## Standard Template Library (STL)

### Containere de tip secenta

**Deque (double ended queue)** - Coadă dublă (completă)

- elementele sunt stocate în blocuri de memorie (chunks of storage)
- elementele se pot adăuga/șterge eficient de la ambele capete

### Vector vs Deque

- Accesul la elemente de pe orice poziție este mai eficient la vector
- Inserare/ștergerea elementelor de pe orice poziție este mai eficient la Deque (dar nu e timp constant)
- Pentru liste mari Vector aloca zone mari de memorie, deque aloca multe zone mai mici de memorie – Deque este mai eficient în gestiunea memoriei



## Standard Template Library (STL)

### Adaptor de container

- încapsulează un container de tip secvență, și folosesc acest obiect pentru a oferi funcționalități specifice containerului (stivă, coadă, coadă cu priorități).

**Stack**: strategia LIFO (last in first out) pentru adaugare/ștergere elemente  
Operații: empty(), push(), pop(), top().

**template < class T, class Container = deque<T> > class stack;**

**Queue**: strategia FIFO (first in first out)  
Operații: empty(), front(), back(), push(), pop(), size();

**template < class T, class Container = deque<T> > class queue;**

**Priority\_queue**: se extrag elemente pe baza priorităților  
Operații: empty(), top(), push(), pop(), size();

**template < class T, class Container = vector<T>, class Compare = less<typename Container::value\_type> > class priority\_queue;**



## Standard Template Library (STL)

### Adaptor de container

#### Exemplu stiva

```
#include <stack>
#include<iostream>
using namespace std;
void sampleStack() {
    stack<int> s;
    s.push(3);
    s.push(4);
    s.push(1);
    s.push(2);

    while (!s.empty())
        {
        cout << s.top() << " ";
        s.pop();
    }
}

int main()
{
sampleStack(); // 2, 1, 4, 3
}
```



## Standard Template Library (STL)

### Adaptor de container

#### Exemplu stiva

```
#include <stack>
#include <vector>
#include<iostream>
using namespace std;
void sampleStack() {
    stack<int,vector<int>> s; // primul parametru = tipul
    // elementelor, al doilea parametru, stilul de stocare
    s.push(3);
    s.push(4);
    s.push(1);
    s.push(2);

    while (!s.empty())
        cout << s.top() << " ";
    s.pop();
}

int main()
{
    sampleStack(); // 2, 1, 4, 3
}
```



## Standard Template Library (STL)

### Adaptor de container

#### Exemplu stiva - influența celui de-al doilea parametru (eventual)

```
class Test { int i;
public:
Test(int x = 0):i(x){cout<<"C ";};
Test(const Test& x){i = x.i; cout<<"CC ";
~Test(){cout<<"D ";}};

int main()
{
    stack<Test> s;
    s.push(1); // C CC D
    cout<<endl;
    s.push(3); // C CC D
    cout<<endl;
    s.push(5); // C CC D urmat de D D D (distrugerea listei)
}
```



## Standard Template Library (STL)

### Adaptor de container

#### Exemplu stiva - influența celui de-al doilea parametru (eventual)

```
class Test { int i;
public:
Test(int x = 0):i(x){cout<<"C ";};
Test(const Test& x){i = x.i; cout<<"CC ";}
~Test(){cout<<"D ";};

int main()
{
    stack<Test, vector<Test> > s;
    s.push(1); // C CC D
    cout<<endl;
    s.push(3); // C CC CC D D
    cout<<endl;
    s.push(5); // C CC CC CC D D D urmat de D D D
}
```



## Standard Template Library (STL)

### Adaptor de container

#### Exemplu stiva - influența celui de-al doilea parametru (eventual)

```
class Test { int i;
public:
Test(int x = 0):i(x){cout<<"C ";};
Test(const Test& x){i = x.i; cout<<"CC ";}
~Test(){cout<<"D ";};

int main()
{
    stack<Test, list<Test> > s;
    s.push(1); // C CC D
    cout<<endl;
    s.push(3); // C CC D
    cout<<endl;
    s.push(5); // C CC D urmat de D D D (distrugerea listei)
}
```



## Standard Template Library (STL)

### Adaptor de container

#### Exemplu coada

```
#include <queue>
#include<iostream>
using namespace std;
void sampleQueue() {
    queue<int> s;
    //queue<int, deque<int> >
    //queue<int, list<int> >

    s.push(3);
    s.push(4);
    s.push(1);
    s.push(2);

    while (!s.empty())
    {
        cout << s.top() << " ";
        s.pop();
    }
}

int main()
{
    sampleQueue(); // 3, 4, 1, 2
}
```



## Standard Template Library (STL)

### Adaptor de container

#### Exemplu coada cu prioritate

```
#include <queue>
#include<iostream>
using namespace std;
void samplePriorQueue() {
    priority_queue<int> s;
    //priority_queue<int, deque<int> >
    //priority_queue<int, list<int> >

    s.push(3);
    s.push(4);
    s.push(1);
    s.push(2);

    while (!s.empty())
        cout << s.top() << " ";
    s.pop();
}

int main() {
    samplePriorQueue(); // 4, 3, 2, 1
}
```



## Standard Template Library (STL)

### Containere asociative

eficiente în accesare elementelor folosind chei (nu folosind poziții ca și în cazul containerelor de tip secvență).

- **Set**

- mulțime - stochează elemente distințte.
- se folosește arbore binar de căutare ca și reprezentare internă

- **Map**

- dictionar - stochează elemente formate din cheie și valoare
- nu putem avea chei duplicate
- **multimap** poate avea chei duplicate

- **Bitset**

- container special pentru a stoca biti.



## Standard Template Library (STL)

### Containere asociative Map

- in sens general, map = lista de perechi cheie - valoare

***template <class Key, class T, class Comp = less<Key>, class Allocator = allocator <pair<const Key,T> > class map***

Key = tipul cheilor

T = tipul valorilor

Comp = functie care compara 2 chei

Allocator = tipul de alocator utilizat (in general cel standard).

***Inserarea se face ordonat după chei.***



## Standard Template Library (STL)

### Containere asociative Map

Constructori:

***explicit map(const Comp &cmpfn = Comp( ), const Allocator &a = Allocator( ) );***

**// expl. map<char,int> m;** - map cu zero elemente;

**map(const map<Key,T,Comp,Allocator> &ob);**

**template <class InIter>  
map(InIter start, InIter end, const Comp &cmpfn = Comp( ), const Allocator &a = Allocator( ));**



## Standard Template Library (STL)

### Containere asociative Map

#### Functii membre uzuale (sursa H. Schildt)

Member	Description
iterator begin( ); const_iterator begin( ) const;	returneaza un iterator catre primul element;
iterator end( ); const_iterator end( ) const; element;	returneaza un iterator catre ultimul
void clear( );	elimina toate elementele din map.
size_type count(const key_type &k) const;	- numarul de aparitii ale lui k
bool empty( ) const;	“true(false)” daca vectorul e (sau nu) gol.



## Standard Template Library (STL)

### Containere asociative Map

#### Functii membre uzuale (sursa H. Schildt)

Member	Description
iterator erase(iterator i);	stergerea elementului pointat de i; returneaza un iterator catre elementul de dupa cel sters.
iterator erase(iterator start, iterator end);	stergerea elementelor intre start si end.
size_type erase(const key_type &k)	- stergerea tuturor valorilor k
iterator find(const key_type &k);	
const_iterator find(const key_type &k) const;	- returneaza un iterator catre valoarea cautata k sau catre sfarsitul map daca nu exista.



## Standard Template Library (STL)

### Containere asociative Map

#### Functii membre uzuale (sursa H. Schildt)

##### Member

size\_type size( ) const;

##### Description

iterator insert(iterator i, const value\_type &val); - insereaza val pe pozitia pointata de i sau dupa;

template <class InIter>

void insert(InIter start, InIter end); - insereaza o secventa start-end.

pair <iterator,bool>

insert(const value\_type &val);

mapped\_type & operator[ ](const key\_type &i) - returneaza o referinta la elementul cheie i, daca nu exista, atunci se insereaza.



## Standard Template Library (STL)

### Containere asociative

#### Map

#### Exemplu

```
#include <iostream>
#include <map>
using namespace std;
int main() {
    map<int, int> m;
    m.insert(pair<int, int>(1,100));
    m.insert(pair<int, int>(7,300));
    m.insert(pair<int, int>(3,200));
    m.insert(pair<int, int>(5,400));
    m.insert(pair<int, int>(2,500));

    map<int, int>::iterator p;
    for(p = m.begin(); p!= m.end(); p++)
        cout<<p->first<<" "<<p->second<<endl;
```

```
//copierea intr-un alt map
map<int,int>m2(m.begin(),m.end());
```



## Standard Template Library (STL)

### Containere asociative

#### Map

#### Exemplu

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

int main() {
    map<char, int> m;
    for(int i=0; i<26; i++)
        m.insert(pair<char, int>('A'+i, 65+i));

    char ch;      cout << "Enter key: ";      cin >> ch;
    map<char, int>::iterator p;
    // find value given key
    p = m.find(ch);
    if(p != m.end())
        cout << "Its ASCII value is " << p->second;
    else cout << "Key not in map.\n";
    return 0;
}
```



## Standard Template Library (STL)

### Containere asociative Multimap

Exemplu - catalog studenti (cheia = nume si valoarea = numar matricol)

```
#include <iostream>
#include <map>
using namespace std;
int main( ) {
    multimap<string, int> m;
    m.insert(m.end(), make_pair("Ionescu",100));
    // echivalent cu m.insert(pair<string,int>("Ionescu",100));

    m.insert(m.end(), make_pair("Popescu",355));
    m.insert(m.end(), make_pair("Ionescu",234));

    map<string, int>::iterator p;
    for(p = m.begin(); p!= m.end(); p++)
        cout<<p->first<<" " <<p->second<<endl;
```



## Standard Template Library (STL)

### Containere asociative Multimap

#### Exemplu - catalog studenti (cheia = nume si valoarea = numar matricol)

```
//Afisarea elementelor cu un anumit nume
string nume;
cin>>nume;
for(p = m.begin(); p!= m.end(); p++)
if (p->first == nume)
    cout<<p->first<<" " <<p->second<<endl;

//stergerea primei aparitii a unui nume
cin>>nume;
map<string, int>::iterator f;
f = m.find(nume);
if (f!=m.end()) m.erase(f);
for(p = m.begin(); p!= m.end(); p++)
    cout<<p->first<<" " <<p->second<<endl;
```



## Standard Template Library (STL)

### Containere asociative Multimap

**Exemplu - catalog studenti (cheia = nume si valoarea = numar matricol)**

```
//Stergerea tuturor aparitiilor unui nume
cin>>nume;

m.erase (nume);
```



## Standard Template Library (STL)

### Iterator

- un concept fundamental în STL, este elementul central pentru algoritmi oferiti de STL;
- obiect care gestionează o poziție (curentă) din containerul asociat;
- suport pentru traversare (++,-), dereferențiere (\*it);
- permite decuplarea între algoritmi și containere.

### *Tipuri de iteratori:*

- iterator input/output (`istream_iterator`, `ostream_iterator`)
- forward iterators, iterator bidirectional, iterator random access
- reverse iterators.

**Adaptoarele de containere (container adaptors) - stack, queue, priority\_queue - nu oferă iterator!**



## Standard Template Library (STL)

### Iterator

Functii uzuale:

- **begin()** - returneaza pozitia de inceput a containerului
- **end()** - returneaza pozitia de dupa terminarea containerului
- **advance()** - incrementeaza pozitia iteratorului
- **next()** - returneaza noul iterator dupa avansarea cu o pozitie
- **prev()** - returneaza noul iterator dupa decrementarea cu o pozitie
- **inserter()** – insereaza elemente pe orice pozitie



## Standard Template Library (STL)

### Iterator

#### Exemplu - begin, end, advance, next, prev:

```
#include<iostream>
#include<iterator>
#include <vector>

using namespace std;
int main()
{
    vector<int> v(5);
    for(int i = 0; i<5; i++) v[i] = (i+1)*10;

    vector<int>::iterator p;
    for (p = v.begin(); p < v.end(); p++)
        cout << *p << " ";
    cout<<endl;
    p = v.begin();
    advance(p, 2);
    vector<int>::iterator n = next(p);
    vector<int>::iterator d = prev(p);
    for (; p < v.end(); p++)
        cout << *p << " ";
    cout<<endl<<*n<<" "<<*d;
}
```

10 20 30 40 50

30 40 50

40 20



## Standard Template Library (STL)

### Iterator

#### Exemplu - inserter:

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v = { 1, 2, 3, 4, 5 };
    vector<int> v2 = { 10, 20, 30 };
    vector<int>::iterator p= v.begin();
    advance(p, 2);
    copy(v2.begin(), v2.end(), inserter(v, p));
    for (p = v.begin(); p < v.end(); p++)
        cout << *p << " ";
    return 0;
}
```

1 2 10 20 30 3 4 5



## Standard Template Library (STL)

### Reverse iterator

#### Exemplu

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> v = { 1, 2, 3, 4, 5 };

    vector<int>::iterator p;
    for (p = v.begin(); p < v.end(); p++)
        cout << *p << " ";
    cout<<endl;
    vector<int>::reverse_iterator r;
    for (r = v.rbegin(); r < v.rend(); r++)
        cout << *r << " ";
    return 0;
}
```



## Standard Template Library (STL)

### Algoritm

- colecție de funcții template care pot fi folosite cu iteratori. Funcțiile operează pe un domeniu (range) definit folosind iteratori;
- domeniu (range) este o secvențe de obiecte care pot fi accesate folosind iteratori sau pointeri.
- headere: **<algorithm>**, **<numeric>**



## Standard Template Library (STL)

### Algoritm

- **Operații pe secvențe**
  - care nu modifică sursa: accumulate, count, find, count\_if, etc
  - care modifică : copy, transform, swap, reverse, random\_shuffle, etc.
- **Sortări:** sort, stable\_sort, etc.
- **Pe secvențe de obiecte ordonate**
  - **Căutare binară** : binary\_search, etc
  - **Interclasare (Merge)**: merge, set\_union, set\_intersect, etc.
- **Min/max:** min, max, min\_element, etc.
- **Heap:** make\_heap, sort\_heap, etc.



## Standard Template Library (STL)

### Algoritm

**Exemplu: – accumulate : calculează suma elementelor**

```
vector<int> v;
v.push_back(3);
v.push_back(4);
v.push_back(2);
v.push_back(7);
v.push_back(17);

//compute the sum of all elements in the vector
cout << accumulate(v.begin(), v.end(), 0) << endl;

//compute the sum of elements from 1 inclusive, 4 exclusive [1,4)
vector<int>::iterator start = v.begin() + 1;
vector<int>::iterator end = v.begin() + 4;
cout << accumulate(start, end, 0) << endl;
```



## Standard Template Library (STL)

### Algoritm

#### Exemplu: – copy

```
vector<int> v;  
v.push_back(3);  
v.push_back(4);  
v.push_back(2);  
v.push_back(7);  
v.push_back(17);  
  
//make sure there are enough space in the destination  
//allocate space for 5 elements  
vector<int> v2(5);  
  
//copy all from v to v2  
copy(v.begin(), v.end(), v2.begin());
```



## Standard Template Library (STL)

### Algoritm

#### Exemplu: – sort

Sorteaza elementele din intervalul [first,last) ordine crescătoare.

- Elementele se compară folosind **operator <**

```
vector<int> v;  
v.push_back(3);  
v.push_back(4);  
v.push_back(2);  
v.push_back(7);  
v.push_back(17);  
  
sort(v.begin(), v.end());
```



## Standard Template Library (STL)

### Algoritm

Exemplu: – sort cu o functie de comparare

```
bool asc(int i, int j) {  
    return (i < j);  
}  
  
vector<int> v;  
v.push_back(3);  
v.push_back(4);  
v.push_back(2);  
v.push_back(7);  
v.push_back(17);  
  
sort(v.begin(), v.end(), asc);
```



## Standard Template Library (STL)

### Algoritm

#### Exemplu: – for\_each

```
void print(int elem) {  
    cout << elem << " "  
}  
  
void testForEach() {  
    vector<int> v;  
    v.push_back(3);  
    v.push_back(4);  
    v.push_back(2);  
    v.push_back(7);  
    v.push_back(17);  
  
    for_each(v.begin(), v.end(), print);  
    cout << endl;  
}
```



## Standard Template Library (STL)

### Algoritm

#### Exemplu: – transform

- aplică funcția pentru fiecare element din secvență ([first1,last1])
- rezultatul se depune în secvența rezultat

```
int multiply3(int el) {
    return 3 * el;
}

void testTransform() {
    vector<int> v;
    v.push_back(3); v.push_back(4);
    v.push_back(2); v.push_back(7);
    v.push_back(17);

    vector<int> v2(5);
    transform(v.begin(), v.end(), v2.begin(), multiply3);
    //print the elements
    for_each(v2.begin(), v2.end(), print);
}
```



## Standard Template Library (STL)

### C++11

#### Lambda Expressions

- permite definirea functiei local, la momentul apelarii;

**Sintaxa:** *[capture](parameters) mutable exception -> return-type {body}*

**capture** - partea introductiva, care spune compilatorului ca urmeaza o expresie lambda; aici se specifica si ce variabile si in ce mod (valoare sau referinta) se copiaza din blocul in care expresia lambda este definita;

**parameters** - parametrii expresiei lambda;

**mutable (optional)** – permite modificarea parametrilor transmisi prin valoare

**exception (optional)** – a se utiliza **noexcept** daca nu se arunca nicio expresie

**return – type (optional)** - tipul la care se evaluateaza expresia lambda; aceasta parte este optionala, cel mai adesea compilatorul putand deduce implicit care este tipul expresiei.

**body** – corpul expresiei



## Standard Template Library (STL)

C++11

### Lambda Expressions

#### Blocul *capture* [ ]

- Expresiile lambda pot “captura” variabilele din program sau poate introduce variabile locale (incepand cu C++14).
- Variabilele transmise cu & sunt accesate prin referinta, iar celelalte prin valoare.
- Setul [ ] fara parametri semnifica faptul ca lambda nu acceseaza variabile din acelasi bloc de program.

```
int a = 0;                                // Define an integer variable
auto f = []() { return a*9; }; // Error: 'a' cannot be accessed
auto f = [a]() { return a*9; }; // OK, 'a' is "captured" by value
auto f = [&a]() { return a++; }; // OK, 'a' is "captured" by reference
                                         // Note: It is the responsibility of the programmer
                                         //       to ensure that a is not destroyed before the
                                         //       lambda is called.
auto b = f();                               // Call the lambda function. a is taken from the capture
```



## Standard Template Library (STL)

C++11

### Lambda Expressions

#### Blocul **capture [ ]** – observatii / restrictii

- Daca [ ] contine & by default, atunci nu se poate declara un argument particular tot cu &
- Daca [ ] contine = by default, atunci nu se poate declara un argument particular tot prin valoare

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};      // OK
    [&, &i]{};     // ERROR: i preceded by & when & is the default
    [=, this]{};   // ERROR: this when = is the default
    [=, *this]{};  // OK: captures this by value. See below.
    [i, i]{};      // ERROR: i repeated
}
```



## Standard Template Library (STL)

C++11

### Lambda Expressions – exemplu

```
#include <bits/stdc++.h>
#include <vector>

using namespace std;

int main() {
    vector<int> v={1,2,1,3,1,1};
    //afisare prin lambda expresie
    for_each(v.begin(), v.end(), [ ](int i){ cout << i << ' '; });
    cout << '\n';

    // contorizare prin lambda expresie
    int cont = 0; //modificat prin lambda expresie
    for_each(v.begin(), v.end(), [&cont](int i){
        if (i == 1)
            cont++;
    });
    cout << cont << " de 1 ";
    return 0;
}
```



## Standard Template Library (STL)

C++11

### Lambda Expressions – exemplu

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    auto sum = [] (int a, int b) {
        return a + b;
    };

    cout << "Sum of two integers:" << sum(5, 6) << endl;

    return 0;
}
```



## Standard Template Library (STL)

C++14

Lambda Expressions – exemplu cu tipuri de date generalizate (incepând cu C++14)

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    // generalized lambda
    auto sum = [] (auto a, auto b) {
        return a + b;
    };

    cout << "Sum(5,6) = " << sum(5, 6) << endl; // sum of two integers
    cout << "Sum(2.0,6.5) = " << sum(2.0, 6.5) << endl; // sum of two float
    cout << "Sum(string(\"SoftwareTesting\"), string(\"help.com\")) = "
        << sum(string("SoftwareTesting"), string("help.com")) << endl; // s
    return 0;
}
```



## Standard Template Library (STL)

C++14

Lambda Expressions – exemplu cu tipuri de date generalizate (incepând cu C++14)

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    // generalized lambda
    auto sum = [] (auto a, auto b) {
        return a + b;
    };

    cout << "Sum(5,6) = " << sum(5, 6) << endl; // sum of two integers
    cout << "Sum(2.0,6.5) = " << sum(2.0, 6.5) << endl; // sum of two float
    cout << "Sum(string(\"SoftwareTesting\"), string(\"help.com\")) = "
        << sum(string("SoftwareTesting"), string("help.com")) << endl; // s
    return 0;
}
```



## Standard Template Library (STL)

### C++11

#### Deducerea tipului in mod automat (**auto** si **decltype**)

In C++03, fiecare variabila trebuie sa aiba un tip de date;

Cuvantul cheie "auto"

In C++11, acesta poate fi dedus din initializarea variabilei respective;

In cazul functiilor, daca tipul returnat este **auto** , este evaluata de expresia returnata la runtime;

Variabilele **auto** TREBUIE initializate, altfel eroare.



## Standard Template Library (STL)

### C++11

#### Deducerea tipului in mod automat (auto si decltype)

```
#include <bits/stdc++.h>
using namespace std;
class Test{ };
int main() {
    auto x = 1; // x e int
    float z;
    auto y = z;// y e float
    auto t = 3.37; // t e double
    auto p = &x; // pointer catre int

    Test ob;
    auto A = ob;
    auto B = &ob;

    cout << typeid(x).name() << endl << typeid(y).name() << endl
        << typeid(t).name() << endl << typeid(p).name() << endl
        << typeid(A).name() << endl << typeid(B).name() << endl;

    return 0;
}
```



## Standard Template Library (STL)

C++11

### Deducerea tipului in mod automat (auto si decltype)

```
#include <bits/stdc++.h>
#include <vector>
using namespace std;
int f(){}
int& g(){}

int main() {
    vector<int> v = {1,2,4};

    auto x = v.begin();
    cout << typeid(x).name() << endl;
    for(auto p = v.begin(); p!=v.end(); p++)
        cout<<*p<<" ";
    cout<<endl;
    auto y = f();
    auto z = g();//    auto& z = g();
    cout << typeid(y).name() << endl << typeid(z).name() << endl;
    return 0;
}
```



## Standard Template Library (STL)

C++11

### Deducerea tipului in mod automat (auto si decltype)

```
int f(){}
float g(){}
// atentie: daca float& g(){}, eroare la decltype(g()) z, intrucat trebuie initializat

int main()
{
    decltype(f()) y;
    decltype(g()) z;
    cout << typeid(y).name() << endl;
    cout << typeid(z).name() << endl;

    return 0;
}
```

***auto – permite declararea unei variabile cu un tip specific, iar decltype "ghiceste" tipul***



## Standard Template Library (STL)

C++11

### Deducerea tipului in mod automat (auto si decltype)

```
#include <bits/stdc++.h>
using namespace std;

template <class T1, class T2>
auto findMin(T1 a, T2 b) -> decltype(a < b ? a : b)
{
    return (a < b) ? a : b;
}

int main()
{
    auto x = findMin(4, 3.44);
    cout << typeid(x).name() << endl;
    decltype(findMin(5.4, 3)) y;
    cout << typeid(y).name() << endl;

    return 0;
}
```



# Perspective

**Cursul 12:**

Design Patterns



# **Programare orientată pe obiecte**

## **- suport de curs -**

**Andrei Păun**  
**Anca Dobrovăț**

**An universitar 2021 – 2022**  
**Semestrul II**  
**Seriile 13, 14 și 15**

**Curs 13**



## Agenda cursului

### Şabloane de proiectare (Design Patterns)

- Definiție și clasificare.
- Exemple de şabloane de proiectare (Singleton, Abstract Object Factory, Observer, Strategy Pattern).

### Obs: Prezentare bazata pe GoF

*(Erich Gamma, Richard Helm, Ralph Johnson si John Vlissides – Design Patterns, Elements of Reusable Object-Oriented Software (cunoscută și sub numele “Gang of Four”), 1994)*



## Sabloane de proiectare (Design patterns)

### Principiile proiectării de clase (S.O.L.I.D) – Robert C. Martin

#### Single-responsibility principle

A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

#### Open–closed principle

"Software entities ... should be open for extension, but closed for modification."

#### Liskov substitution principle

"Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."

#### Interface segregation principle

"Many client-specific interfaces are better than one general-purpose interface."

#### Dependency inversion principle

One should "depend upon abstractions, [not] concretions."



## Sabloane de proiectare (Design patterns)

### Principiile proiectarii de clase

#### *Principiul “inchis-deschis”*

“Entitatile software (module, clase, functii etc.) trebuie sa fie deschise la extensii si inchise la modificare” (Bertrand Meyer, 1988).

“deschis la extensii” = comportarea modulului poate fi extinsa pentru a satisface noile cerinte.

“inchis la modificare” = nu este permisa modificarea codului sursa.



## Sabloane de proiectare (Design patterns)

### Principiile proiectarii de clase

#### *Principiul substituirii*

Functiile care utilizeaza pointeri sau referinte la clasa de baza trebuie sa fie apte sa utilizeze obiecte ale claselor derivate fara sa le cunoasca.

#### *Principiul de inversare a dependentelor*

- A. “Modulele de nivel inalt nu trebuie sa depinda de modulele de nivel jos. Ambele trebuie sa depinda de abstractii.”
- B. “Abstractiile nu trebuie sa depinda de detalii. Detaliile trebuie sa depinda de abstractii.”
  - programele OO bine proiectate inverseaza dependenta structurala de la metoda procedurala traditionala
  - metoda procedurala: o procedura de nivel inalt apeleaza o procedura de nivel jos, deci depinde de ea



## Sabloane de proiectare (Design patterns)

### Definitie si clasificare

Aplicarea principiilor pentru a crea arhitecturi OO → se ajunge repetat la aceleasi structuri, cunoscute sub numele de **sabloane de proiectare (design patterns)**.

Un **sablon de proiectare** descrie

- o problema care se intalneste in mod repetat in proiectarea programelor
- solutia generala pentru problema respectiva

Solutia este exprimata folosind **clase si obiecte**.

**Cand si unde a aparut ideea?**

- Arhitectura
- 1977: “A pattern language: Towns, Buildings, Construction”
- Christopher Alexander

[https://en.wikipedia.org/wiki/Pattern\\_language](https://en.wikipedia.org/wiki/Pattern_language)



## Sabloane de proiectare (Design patterns)

### Definitie si clasificare

#### *Clasificarea şabloanelor după scop:*

- **creaționale** (creational patterns) privesc modul de creare al obiectelor.
- **structurale** (structural patterns) se referă la compoziția claselor sau a obiectelor.
- **comportamentale** (behavioral patterns) caracterizează modul în care obiectele și clasele interacționează și își distribuie responsabilitățile.

#### *Clasificarea şabloanelor după domeniu de aplicare:*

- sabloanele claselor** se referă la relații dintre clase, relații stabilite prin moștenire și care sunt statice (fixate la compilare).
- sabloanele obiectelor** se referă la relațiile dintre obiecte, relații care au un caracter dinamic .



## Sabloane de proiectare (Design patterns)

### Definitie si clasificare

In general, un sablon are 4 elemente esentiale:

- 1. nume**
- 2. descrierea problemei.** (contextul in care apare, cand trebuie aplicat sablonul).
- 3. descrierea solutiei.** (elementele care compun proiectul, relatiile dintre ele, responsabilitatile si colaborarile).
- 4. consecintele si compromisuri aplicarii sablonului.**

Un sablon de proiectare descrie de asemenea problemele de implementare ale sablonului si un exemplu de implementare a sablonului in unul sau mai multe limbaje de programare.



## Sabloane de proiectare (Design patterns)

### Structura unui sablon

În cartea de referință (GoF), descrierea unui sablon este alcătuită din urmatoarele secțiuni:

**Numele sablonului și clasificarea**

**Intentia**

**Alte nume prin care este cunoscut**, dacă există.

**Motivatia** - scenariu care ilustrează o problema de proiectare și rezolvarea ;

**Aplicabilitatea**

**Structura** - reprezentată grafic prin diagrame de clase și de interacțiune (UML) ;

**Participanți** - clasele și obiectele și responsabilitatile lor;

**Colaborari**

**Consecințe** - compromisurile și rezultatele utilizării sablonului.

**Implementare** - tehnici de implementare, aspectele dependente de limbaj

**Exemplu de cod**

**Utilizari cunoscute**

**Sabloane corelate**



## Sabloane de proiectare (Design patterns)

Unele dintre sabloanele de proiectare cele mai folosite sunt descrise in cele ce urmeaza:

### **1. Abstract Server**

Cand un client depinde direct de server, este incalcat principiul de inversare a dependentelor. Modificarile facute in server se vor propaga in client, iar clientul nu va fi capabil sa foloseasca alte servere similare cu acela pentru care a fost construit.

Situatia de mai sus se poate imbunatati prin inserarea unei interfete abstracte intre client si server,

Client -> Interfata <- Manager



## Sabloane de proiectare (Design patterns)

### 2. Adapter

Cand inserarea unei interfete abstracte nu este posibila deoarece serverul este produs de o alta companie (third party ISV) sau are foarte multe dependente de intrare care-l fac greu de modificat, se poate folosi un ADAPTER pentru a lega interfata abstracta de server.

Client -> Interfata <- Adapter ->Manager.



## Sabloane de proiectare (Design patterns)

### 3. Singleton (*clase cu o singura instanta*)

#### Intentia

proiectarea unei clase cu un singur obiect (o singura instanță)

#### Motivatia

într-un sistem de operare:

există un sistem de fișiere

există un singur manager de ferestre

**Aplicabilitate** când trebuie să existe exact o instanta: clientii clasei trebuie să aibă acces la instanta din orice punct bine definit.



## Sabloane de proiectare (Design patterns)

### ***Singleton (clase cu o singura instantă)***

#### **Consecințe**

- acces controlat la instantă unică;
- reducerea spațiului de nume (eliminarea variab. globale);
- permite rafinarea operațiilor și reprezentării;
- permite un număr variabil de instanțe;
- mai flexibilă decât operațiile la nivel de clasă (statice).



## Sabloane de proiectare (Design patterns)

### *Singleton (clase cu o singura instanta) - exemplu cu referinte*

```
class Singleton
{
public:
    static Singleton& instance()
    {
        return uniqueInstance;
    }
    int getValue() { return data; }
    void setValue(int value) { data = value; }

private:
    static Singleton uniqueInstance;
    int data;
    Singleton(int d = 0):data(d) { }
    Singleton & operator=(Singleton & ob) {
        if (this != &ob) data = ob.data; return *this; }
    Singleton(const Singleton & ob) { data = ob.data; }
};

Singleton Singleton::uniqueInstance (0);
```

```
int main()
{
    Singleton& s1 = Singleton::instance();
    cout << s1.getValue() << endl;
    Singleton& s2 = Singleton::instance();
    s2.setValue(9);
    cout << s1.getValue() << endl;
    return 0;
}
```



## Sabloane de proiectare (Design patterns)

### *Singleton (clase cu o singura instanta) - exemplu cu pointeri*

```
class Singleton
{
public:
    static Singleton* instance()
    {
        if (uniqueInstance == NULL)
            uniqueInstance = new Singleton();
        return uniqueInstance;
    }
    int getValue() { return data; }
    void setValue(int value) { data = value; }

private:
    static Singleton* uniqueInstance;
    int data;
    Singleton(int d = 0):data(d)      {     }
    Singleton & operator=(Singleton & ob) {
        if (this != &ob) data = ob.data; return *this; }
    Singleton(const Singleton & ob) { data = ob.data; }
};

Singleton* Singleton::uniqueInstance = NULL;

int main()
{
    Singleton* s1 = Singleton::instance();
    cout << s1->getValue() << endl;
    Singleton* s2 = Singleton::instance();
    s2->setValue(9);
    cout << s1->getValue() << endl;
    return 0;
}
```



## Sabloane de proiectare (Design patterns)

### *Singleton (clase cu o singura instanta) - exemplu*

```
#include <iostream>

using namespace std;

class Ceas_intern
{
    static Ceas_intern* instanta;
    int timestamp;

    Ceas_intern(int d = 0):timestamp(d) { }
    Ceas_intern & operator=(Ceas_intern & ob);
    Ceas_intern(const Ceas_intern & ob);

public:
    static Ceas_intern* get_instanta()
    {
        if (instanta == NULL) instanta = new Ceas_intern();
        return instanta;
    }
    void adauga_zile(int);
    void adauga_luni(int);
    int get_timestamp();
};

Ceas_intern* Ceas_intern::instanta = NULL;
```



## Sabloane de proiectare (Design patterns)

### Singleton (clase cu o singura instanta) - exemplu

```
Ceas_intern & Ceas_intern::operator=(Ceas_intern & ob)
{
    if (this != &ob)
        timestamp = ob.timestamp;
    return *this;
}

Ceas_intern::Ceas_intern(const Ceas_intern & ob)
{
    timestamp = ob.timestamp;
}

void Ceas_intern::adauga_zile(int d)
{
    timestamp+=d;
}

void Ceas_intern::adauga_luni(int m)
{
    timestamp+=22*m;
}

int Ceas_intern::get_timestamp()
{
    return timestamp;
}
```

```
int main()
{
    Ceas_intern* ob1 = Ceas_intern::get_instanta();
    cout<<ob1->get_timestamp()<<endl;
    ob1->adauga_zile(10);
    cout<<ob1->get_timestamp()<<endl;
    Ceas_intern* ob2 = Ceas_intern::get_instanta();
    cout<<ob2->get_timestamp()<<endl;
    ob2->adauga_zile(10);
    cout<<ob2->get_timestamp()<<endl;
    cout<<ob1->get_timestamp()<<endl;
}
```

0  
10  
10  
20  
20



## Sabloane de proiectare (Design patterns)

### 4. Observer

**Intentia:** Defineste o dependenta “unul la mai multi” intre obiecte, astfel incat atunci cand unul dintre obiecte isi schimba starea toate obiectele dependente sunt notificate si actualizate automat.

**Alte nume prin care este cunoscut:** Dependents, Publish-Subscribe.

**Motivatia** descrie cum sa se stabileasca relatiile intre clase.

Obiectele cheie in acest sablon sunt **subiect** si **observator**.

Un subiect poate avea orice numar de observatori dependenti.

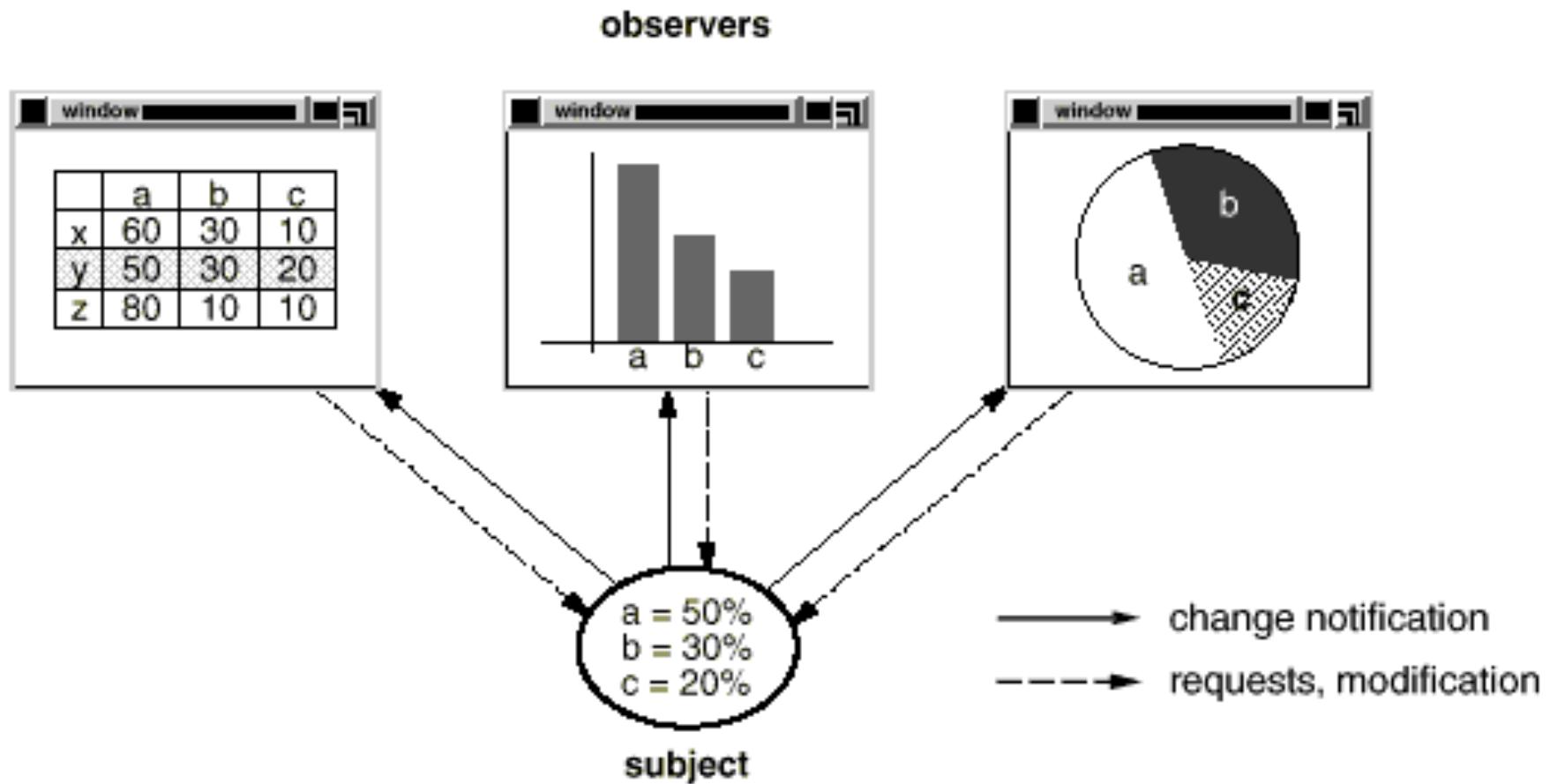
Totii observatorii sunt notificati ori de cate ori subiectul isi schimba starea.

Ca raspuns la notificare, fiecare observator va interoga subiectul pentru a-si sincroniza starea cu starea subiectului.



## Sabloane de proiectare (Design patterns)

### Observer

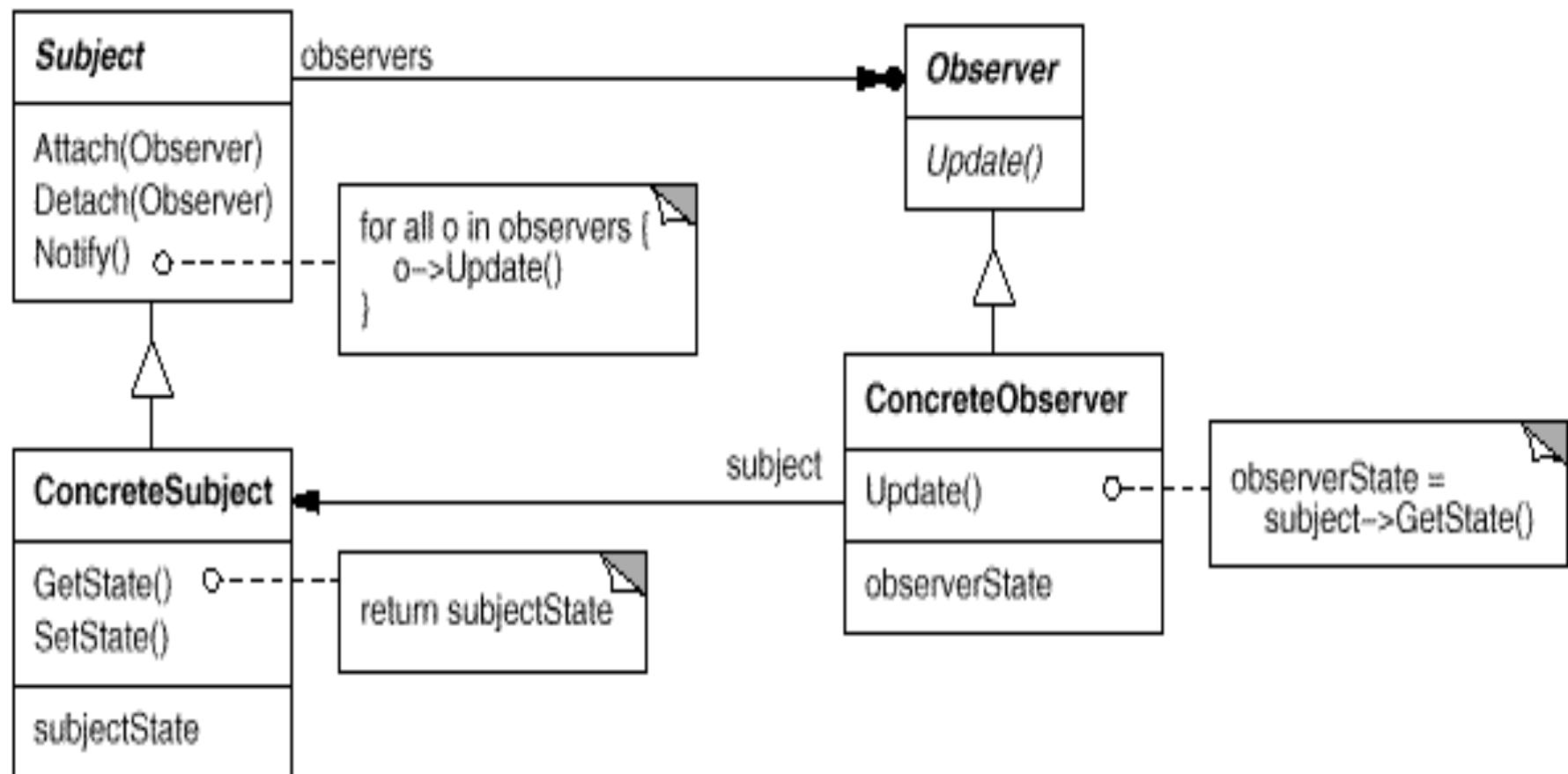




## Sabioane de proiectare (Design patterns)

### Observer

### Structura





## Sabloane de proiectare (Design patterns)

### *Observer*

### *Aplicabilitatea*

Sablonul poate fi utilizat în oricare dintre urmatoarele situații:

- cand o abstractie are două aspecte, unul dependent de celalalt (daca sunt encapsulate în obiecte separate, ele pot fi reutilizate independent).
  
- cand modificarea unui obiect necesită modificarea altor obiecte și nu se stie care obiecte trebuie să fie modificate.
  
- cand un obiect trebuie să notifice alte obiecte fără a face presupuneri despre cine sunt aceste obiecte.



## Sabloane de proiectare (Design patterns)

### *Observer*

### *Participantii*

#### **Subject**

Cunoaște observatorii sai.

Un obiect **Subject** poate fi observat de orice număr de obiecte **Observer**

Furnizează o interfață pentru atașarea și detasarea obiectelor **Observer**.

#### **Observer**

Defineste o interfață pentru actualizarea obiectelor care trebuie să fie notificate (anunțate) despre modificările din subiect.



## Sabloane de proiectare (Design patterns)

### *Observer*

### *Participantii*

#### **ConcreteSubject**

Memoreaza starea de interes pentru obiectele ConcreteObserver.  
Trimite o notificare observatorilor sai atunci cand i se schimba starea.

#### **ConcreteObserver**

Mentine o referinta la un obiect ConcreteSubject.  
Memoreaza starea care trebuie sa ramana consistenta cu a subiectului.  
Implementeaza interfata de actualizare a clasei Observer



## Sabloane de proiectare (Design patterns)

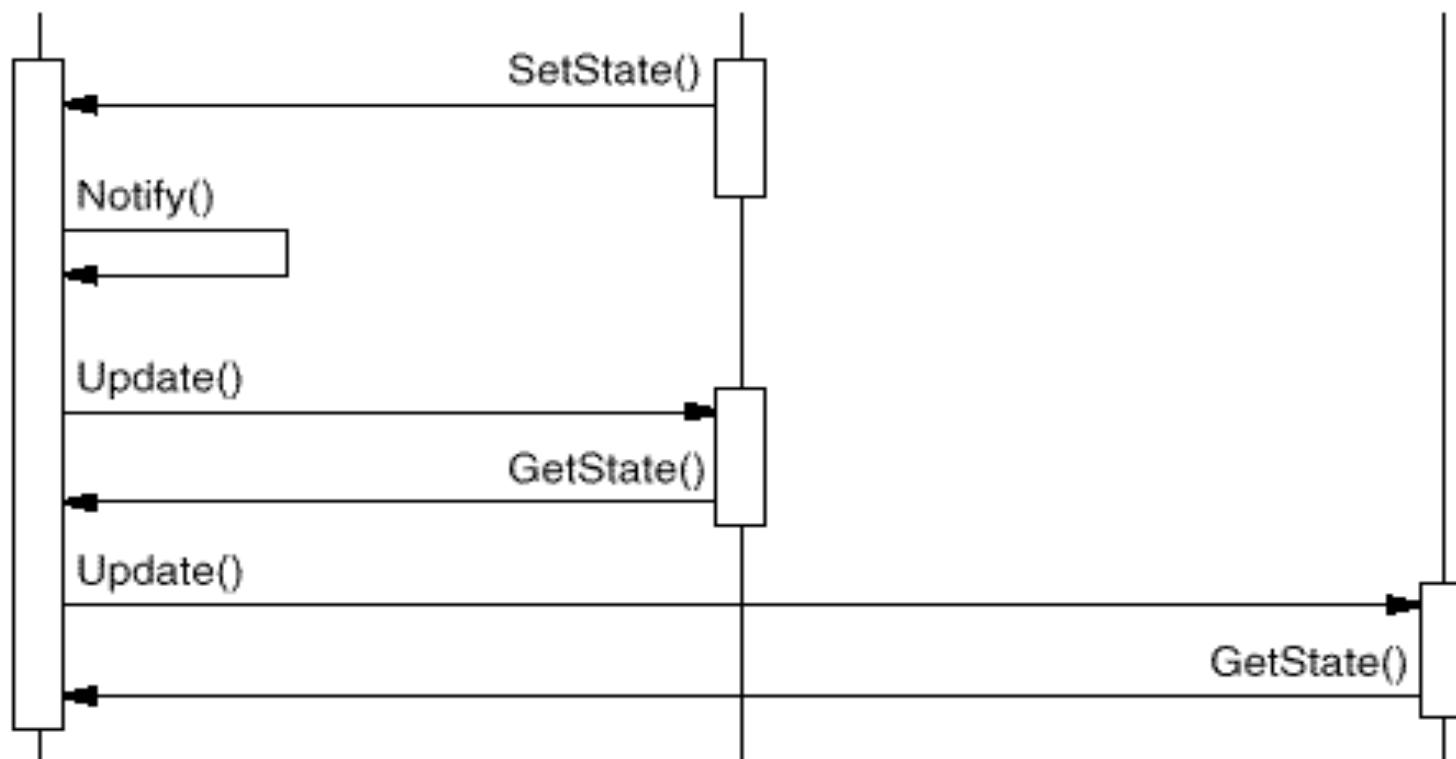
### Observer

#### Colaborari

aConcreteSubject

aConcreteObserver

anotherConcreteObserver





## Sabloane de proiectare (Design patterns)

### *Observer*

#### *Exemplu de cod*

Interfata **Observer** este definită printr-o clasa abstractă:

```
class Observer {  
public:  
    virtual ~Observer();  
    virtual void Update(Subject* theChangedSubject) = 0;  
protected:  
    Observer();  
};
```



## Sabloane de proiectare (Design patterns)

### Observer

#### Exemplu de cod

Interfata **Subject** este definită prin urmatoarea clasa:

```
class Subject {
public:
    virtual ~Subject();
    virtual void Attach(Observer* );
    virtual void Detach(Observer* );
    virtual void Notify();
protected:
    Subject();
private:
    List<Observer*> *_observers;
};

void Subject::Attach (Observer* o) { _observers->Append(o); }
void Subject::Detach (Observer* o) { _observers->Remove(o); }
void Subject::Notify () {
    ListIterator<Observer*> i(_observers);
    //construieste un iterator, i, pentru containerul _observers
    for (i.First(); !i.IsDone(); i.Next()) { i.CurrentItem()->Update(this); }
}
```



## Sabloane de proiectare (Design patterns)

### *Observer*

un subiect concret

```
class ClockTimer : public Subject {
public:
    ClockTimer();
    virtual int GetHour();
    virtual int GetMinute();
    virtual int GetSecond();
    void Tick();
};

void ClockTimer::Tick () {
    // update internal time-keeping state
    // ...
    Notify();
}
```



## Sabloane de proiectare (Design patterns)

### *Observer*

un observator concret care mosteneste in plus o interfata grafica

```
class DigitalClock: public Widget, public Observer
{
public:
    DigitalClock(ClockTimer* );
    virtual ~DigitalClock();
    virtual void Update(Subject* );
        // overrides Observer operation
    virtual void Draw();
        // overrides Widget operation;
        // defines how to draw the digital clock
private:
    ClockTimer* _subject;
};
```



## Sabloane de proiectare (Design patterns)

### Observer

un observator concret care mosteneste in plus o interfata grafica

```
DigitalClock::DigitalClock (ClockTimer* s) {
    _subject = s;
    _subject->Attach(this); }

DigitalClock::~DigitalClock () { _subject->Detach(this);}

void DigitalClock::Update (Subject* theChangedSubject) {
    if (theChangedSubject == _subject) { Draw(); } }

void DigitalClock::Draw () { // get the new values from the subject
    int hour = _subject->GetHour();
    int minute = _subject->GetMinute();
    // draw the digital clock
}
```



## Sabloane de proiectare (Design patterns)

### *Observer*

#### un alt observator

```
class AnalogClock : public Widget, public Observer {  
public:  
    AnalogClock(ClockTimer*);  
    virtual void Update(Subject*);  
    virtual void Draw();  
    // ...  
};  
  
/* crearea unui AnalogClock si unui DigitalClock care arata  
acelasi timp: */  
  
ClockTimer* timer = new ClockTimer;  
AnalogClock* analogClock = new AnalogClock(timer);  
DigitalClock* digitalClock = new DigitalClock(timer);
```



## Sabloane de proiectare (Design patterns)

### 5. Abstract Object Factory

#### intenție

- de a furniza o interfață pentru crearea unei familii de obiecte intercorelate sau dependente fără a specifica clasa lor concreta.

#### aplicabilitate

- un sistem ar trebui să fie independent de modul în care sunt create produsele, compuse sau reprezentate
- un sistem ar urma să fie configurat cu familii multiple de produse
- o familie de obiecte intercorelate este proiectată pentru astfel ca obiectele să fie utilizate împreună
- **se dorește furnizarea unei biblioteci de produse, dar se dorește accesibila numai interfața, nu și implementarea.**



## Sabloane de proiectare (Design patterns)

### *Abstract Object Factory*

#### colaborari

- normal se creeaza o singura instanta

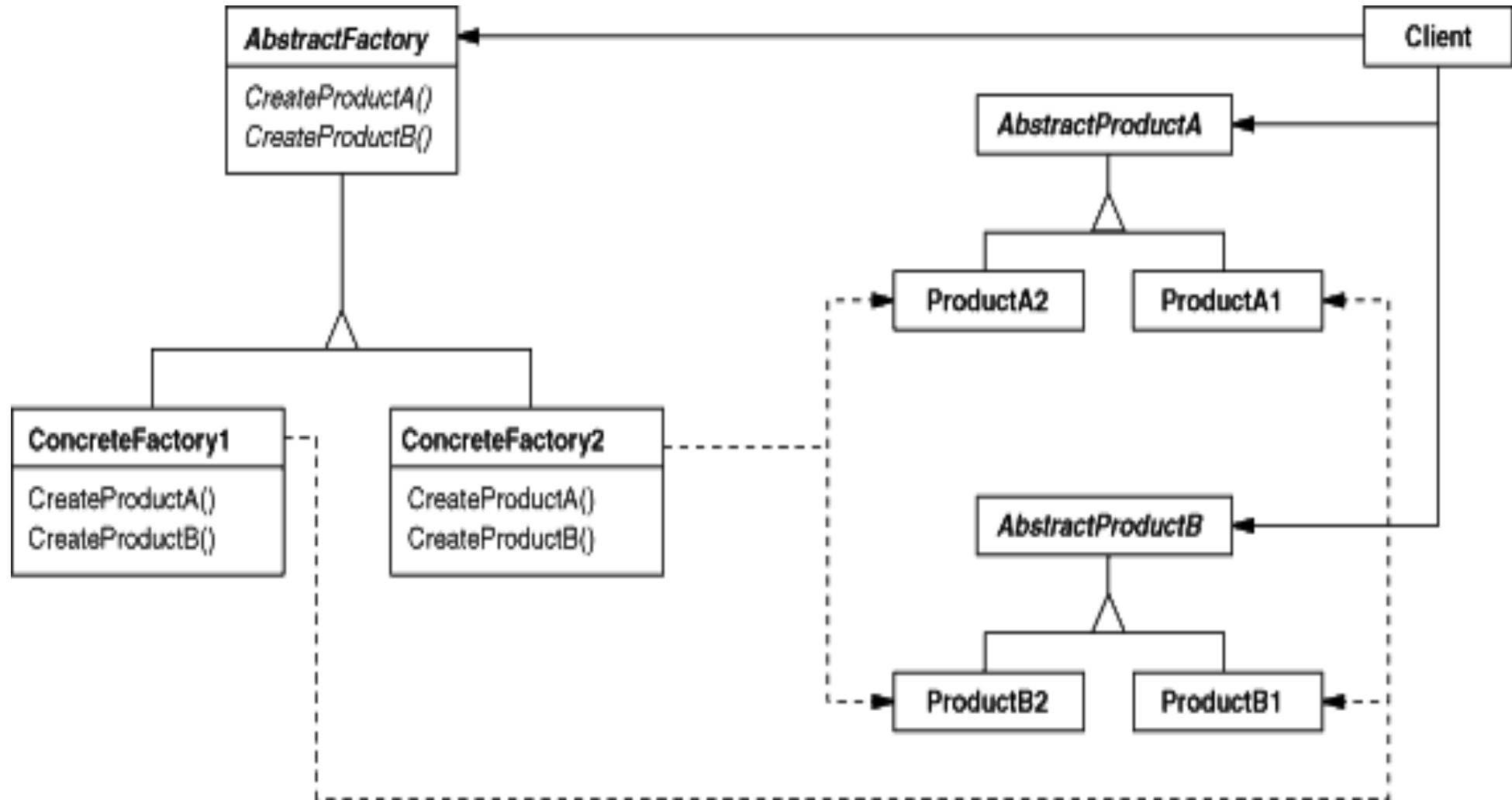
#### consecinte

- izoleaza clasele concrete
- simplifica schimbul familiei de produse
- promoveaza consistenta printre produse
- suporta noi timpul noi familii de produse usor
- respecta principiul deschis/inchis



## Sabloane de proiectare (Design patterns)

### *Abstract Object Factory* structura





## Sabloane de proiectare (Design patterns)

### *Abstract Object Factory*

#### implementare

o functie delegat (callback) este o functie care nu este invocata explicit de programator; responsabilitatea apelarii este delegata altei functii care primeste ca parametru adresa functiei delegat

Fabrica de obiecte utilizeaza functii delegat pentru crearea de obiecte: pentru fiecare tip este delegata functia care creeaza obiecte de acel tip.



## Sabloane de proiectare (Design patterns)

### *Abstract Object Factory*

#### solutia

- definim mai intai clasa de baza ca si clasa abstracta

```
class Figura {  
public:  
    virtual void afis() = 0;  
};
```

- Definim grupurile de produse / tipurile de produse



## Sabloane de proiectare (Design patterns)

### Abstract Object Factory

#### solutia

- Definim grupurile de produse / tipurile de produse

```
***** Produse tip A *****  
class Cerc : public Figura {  
public:  
    void afis() {  
        cout << "Cerc\n";  
    }  
};  
  
class Patrat : public Figura {  
public:  
    void afis() {  
        cout << "Patrat\n";  
    }  
};
```

```
***** Produse tip B *****  
class Elipsa : public Figura {  
public:  
    void afis() {  
        cout << "Elipsa\n";  
    }  
};  
  
class Dreptunghi : public Figura {  
public:  
    void afis() {  
        cout << "Dreptunghi\n";  
    }  
};
```



## Sabloane de proiectare (Design patterns)

### *Abstract Object Factory*

#### solutia

definim apoi o fabrica de figuri, adica o clasa care sa gestioneze tipurile de figuri

```
class Factory {
public:
    virtual Figura* creeaza_figuri_fara_colturi() = 0;
    virtual Figura* creeaza_figuri_cu_colturi() = 0;
};
```



## Sabloane de proiectare (Design patterns)

### *Abstract Object Factory*

#### solutia

Generam factory pentru grupuri de produse

```
class Figuri_simple : public Factory {
public:
    Figura* creeaza_figuri_fara_colturi() {
        return new Cerc;
    }
    Figura* creeaza_figuri_cu_colturi() {
        return new Patrat;
    }
};

class Figuri_robuste : public Factory {
public:
    Figura* creeaza_figuri_fara_colturi() {
        return new Elipsa;
    }
    Figura* creeaza_figuri_cu_colturi() {
        return new Dreptunghi;
    }
};
```



## Sabloane de proiectare (Design patterns)

### *Abstract Object Factory*

#### solutia

Apel fara a numi efectiv figurile

```
int main() {  
  
    Factory* factory = new Figuri_simple();  
    Figura* f[3];  
  
    f[0] = factory->creeaza_figuri_fara_colturi();  
    f[1] = factory->creeaza_figuri_cu_colturi();  
    f[2] = factory->creeaza_figuri_fara_colturi();  
  
    for (int i=0; i < 3; i++) f[i]->afis();  
}
```



## Sabloane de proiectare (Design patterns)

### 6. Strategy pattern

#### intenție

- Presupune encapsularea separată a fiecarui algoritm dintr-o familie, facând astfel ca algoritmii respectivi să fie interschimbabili.

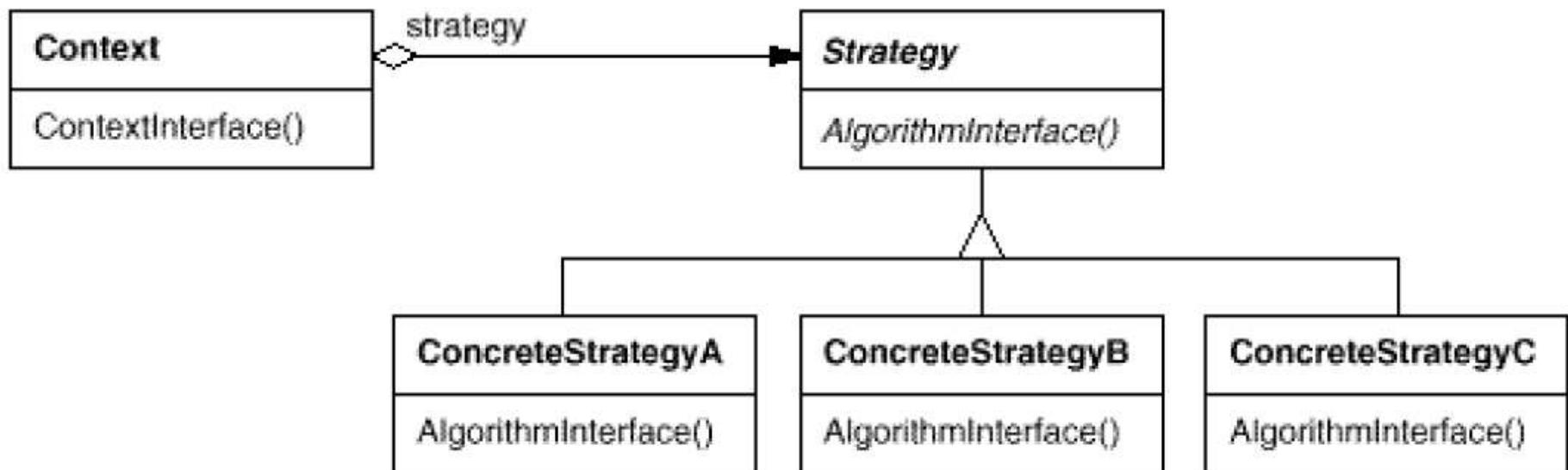
#### aplicabilitate

- mai multe clase înrudite diferă doar prin comportament;
- sunt necesare mai multe variante ale unui algoritm, care diferă între ele, de exemplu, prin compromisul spațiu-temp adoptat;
- un algoritm utilizează date pe care clientul algoritmului nu trebuie să le cunoască;
- într-o clasa sunt definite mai multe acțiuni care apar ca structuri conditionale multiple. În loc de aceasta, se recomandă plasarea ramaurilor conditionale înrudită în cale o clasa strategy separată.



## Sabloane de proiectare (Design patterns)

### Strategy pattern structura



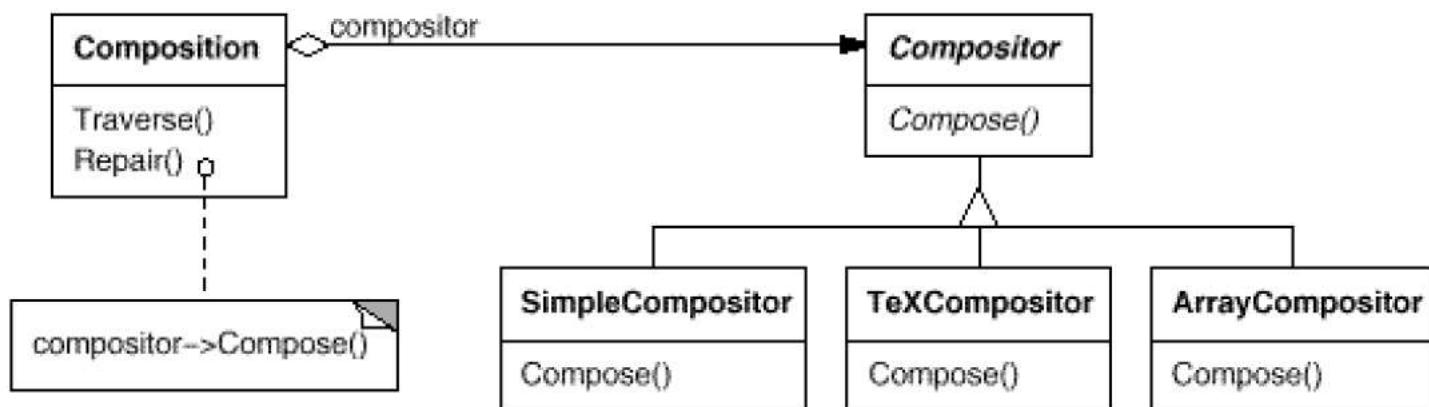


## Sabloane de proiectare (Design patterns)

### *Strategy pattern* exemplu

Algoritm care imparte un text pe linii

- SimpleComposer – algoritm simplu cu \n
- TeXComposer – algoritm care grupeaza, mai eficient, pe paragrafe
- ArrayComposer – algoritm care imparte textul astfel incat, pe fiecare linie, sa existe acelasi nr de caractere





## Sabloane de proiectare (Design patterns)

### *Strategy pattern*

#### Exemplu implementare

Problema!!

```
class Turist
{
    string nume;
public:
    void transport_cu_masina () {cout<<"masina\n";}
    void transport_cu_avion () {cout<<"avion\n";}
    void transport_cu_bicicleta () {cout<<"bicicleta\n";}
};
```



## Sabloane de proiectare (Design patterns)

### *Strategy pattern*

#### Exemplu implementare

#### Strategiile de transport

```
class I_Transport
{
public:
    virtual void transport() const = 0;
};

class Masina : public I_Transport
{
public:
    void transport() const {cout<<"masina\n";}
};

class Avion : public I_Transport
{
public:
    void transport() const {cout<<"avion\n";}
};

class Bicicleta : public I_Transport
{
public:
    void transport() const {cout<<"bicicleta\n";}
};
```



## Sabloane de proiectare (Design patterns)

### Strategy pattern

#### Exemplu implementare

Noua clasa turist (care contine un pointer catre modul de deplasare)

```
class Turist
{
    string nume;
    I_Transport* modTransport;
public:
    Turist (string n, I_Transport* modInitial){nume = n; modTransport = modInitial;}

    void setModTransport(I_Transport* modNou)
    {
        if (modTransport != NULL) delete modTransport;
        modTransport = modNou;
    }
    void deplasare()
    {
        if (modTransport == NULL) throw "Nu ai selectat un mijloc de transport";
        modTransport->transport(); // nu se mai leaga de implementare ca inainte
    }
    virtual ~Turist(){delete modTransport;}
};
```



## Sabloane de proiectare (Design patterns)

### *Strategy pattern*

#### Exemplu implementare

Apel

```
int main()
{
    Turist t("Popescu", new Avion());
    t.deplasare();

    t.setModTransport(new Masina());
    t.deplasare();

    t.setModTransport(new Bicicleta());
    t.deplasare();
}
```



## Curs 14

*Succes la colocviu și la examenul scris!*