


The background features abstract, overlapping green geometric shapes, primarily triangles and polygons, in various shades of green, creating a modern, layered effect.

# P00

Curs-4

Gavrilut Dragos

- ▶ Mostenire
- ▶ Alinieri in memorie pentru clasele derivate
- ▶ Cast-uri intre clasele derivate
- ▶ Functii virtuale
- ▶ Clase abstracte (interfete)

- 
- ▶ **Mostenire**
  - ▶ Alinieri in memorie pentru clasele derivate
  - ▶ Cast-uri intre clasele derivate
  - ▶ Functii virtuale
  - ▶ Clase abstracte (interfete)

# Mostenirea

- ▶ Mostenirea este procesul prin care o serie de proprietati (metode si membri) se adauga la una sau mai multe clase rezultand o clasa nou (care are toate proprietatile claselor din care a fost derivata + o serie de proprietati noi).
- ▶ Un exemplu ar putea fi clasa automobil cu urmatoarele proprietati:
  - ▶ Numar de usi
  - ▶ Numar de roti
  - ▶ Dimensiune
- ▶ Din aceasta clasa derivam clasa DACIA, care e in cazul de fata o particularizare a clasei automobile (pastreaza proprietatile clasei automobile) dar mai adauga cateva in plus (de exemplu o caracteristica a motorului care este prezenta DOAR la autoturismele DACIA)

# Mostenirea

- ▶ Mostenirea poate fi simpla sau multipla

- ▶ Mostenire simpla

## Simpla

```
class <nume_clasa>: <modificator de access> <clasa de baza> { ... }
```

- ▶ Mostenire multipla

## Multipla

```
class <nume_clasa>: <modificator de access> <clasa de baza 1> ,  
                  <modificator de access> <clasa de baza 2> ,  
                  <modificator de access> <clasa de baza 3> ,  
                  ...  
                  <modificator de access> <clasa de baza n> ,  
{ ... }
```

- ▶ Modificatorul de access este optional (si are una din valorile public / private sau protected). Daca nu se specifica se considera ca mostenirea e de tipul private.

# Mostenirea

- Exemplu. Clasa Derived mosteneste membri si metodele clasei Base. Din acest motiv poate apela metoda SetX sau accesa membrul x.

## App.cpp

```
class Base
{
public:
    int x;
    void SetX(int value);
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
};

void main()
{
    Derived d;
    d.SetX(100);
    d.x = 10;
    d.SetY(200);
}
```

# Mostenirea

- Codul de mai jos nu compileaza. Clasa Derived mosteneste clasa Base, dar variabila x din Base este declarata ca si private. Asta inseamna ca poate fi accesata doar din clasa Base si NU si din clase derivate din Base

## App.cpp

```
class Base
{
private:
    int x;
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void SetX(int value);
};
void Derived::SetX(int value)
{
    x = value;
}
void main()
{
    Derived d;
    d.SetX(100);
    d.SetY(200);
}
```

# Mostenirea

- Solutia in acest caz este utilizarea unui al treilea tip de modificador de access (protected). Protected specifica ca variabila nu poate fi accesata din afara, dar poate fi accesata dintr-o clasa derivata.

## App.cpp

```
class Base
{
    protected:
        int x;
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void SetX(int value);
};
void Derived::SetX(int value)
{
    x = value;
}
void main()
{
    Derived d;
    d.SetX(100);
    d.SetY(200);
}
```



# Mostenirea

- Codul de mai jos nu compileaza. “x” este protected in clasa Base - poate fi accesat de o clasa derivata, dar nu poate fi accesat din afara clasei.

## App.cpp

```
class Base
{
protected:
    int x;
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void SetX(int value);
};
void Derived::SetX(int value)
{
    x = value;
}
void main()
{
    Derived d;
    d.SetX(100);
    d.x = 100;
}
```

# Mostenirea

- Modificatorii de access (tinand cont si de mostenire) permit accesul la date astfel:

Modificator de acces	In aceeaasi clasa	In clasa derivata	In afara clasei	Functie friend In clasa de baza	Functie friend in clasa derivata
public	DA	DA	DA	DA	DA
protected	DA	DA	NU	DA	DA
private	DA	NU	NU	DA	NU

# Mostenirea

- Codul de mai jos NU compileaza. “x” din Base este private, iar functia friend definite in Derived NU il poate accesa

## App.cpp

```
class Base
{
    private:
        int x;
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void friend SetX(Derived &d);
};
void SetX(Derived &d)
{
    d.x = 100;
}
void main()
{
    Derived d;
    SetX(d);
}
```

# Mostenirea

- Solutia este sa transformam x din private in protected sau public sau sa punem o functie friend in clasa Base

## App.cpp

```
class Base
{
    protected:
        int x;
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void friend SetX(Derived &d);
};
void SetX(Derived &d)
{
    d.x = 100;
}
void main()
{
    Derived d;
    SetX(d);
}
```

# Mostenirea

- Atentie unde definiti functia friend. Codul de mai jos NU compileaza pentru ca functia friend e definita in clasa Derived - chiar daca are un parametru de tip Base, ea poate accesa doar membri private din clasa Derive.

## App.cpp

```
class Base
{
private:
    int x;

};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
    void friend SetX(Base &d);
};
void SetX(Base &d)
{
    d.x = 100;
}
void main()
{
    Derived d;
}
```

# Mostenirea

- Codul current insa functioneaza corect - pentru ca functia friend SetX este definite in clasa Base (deci poate accesa membri privati ai acestei clase).

## App.cpp

```
class Base
{
private:
    int x;
public:
    void friend SetX(Base &d);
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value);
};
void SetX(Base &d)
{
    d.x = 100;
}
void main()
{
    Derived d;
}
```

# Mostenirea

- ▶ Modificatorii de access nu se aplica doar la membri si metode, ci si la relatia de mostenire.
- ▶ Rezultatul e ca metodele si membri din clasa de baza isi schimba modificatorul de access in functie de cum se realizeaza relatia de mostenire.

## App.cpp

```
class Base
{
public:
    int x;
};
class Derived : public Base
{
    int y;
public:
    void SetY(int value) { ... }
};
void main()
{
    Derived d;
    d.x = 100;
}
```

- ❖ In cazul de fata, pentru ca derivarea se face folosind tot public, “x” din clasa de baza ramane tot public → deci va putea fi accesat din afara clasei

# Mostenirea

- ▶ Modificatorii de access nu se aplica doar la membri si metode, ci si la relatia de mostenire.
- ▶ Rezultatul e ca metodele si membri din clasa de baza isi schimba modificatorul de access in functie de cum se realizeaza relatia de mostenire.

## App.cpp

```
class Base
{
public:
    int x;
};
class Derived : private Base
{
    int y;
public:
    void SetY(int value) { ... }
};
void main()
{
    Derived d;
    d.x = 100;
}
```

- ❖ Codul alaturat inasa nu compileaza, pentru ca mostenirea facandu-se cu specificatorul private transforma pe “x” din clasa Base din public in private, deci prin urmare nu mai poate fi accesat din afara clasei lui (pentru o variabila de tipul Derived)
- ❖ Poate fi accesat fara probleme din afara clasei pentru o variabila de tipul Base (unde este public in continuare)



# Mostenirea

- Regulile de schimbare a specificatorului de access pentru membrii si metodele unei clase de baza in cazul unei mosnteniri sunt in urmatoarele:

Specificator de access folosit la mostenire →	public	private	protected
Specificator de access folosit in clasa de baza la metode si membri			
public	public	private	protected
private	private	private	private
protected	protected	private	protected

- ▶ Mostenire
- ▶ Alinieri in memorie pentru clasele derivate
- ▶ Cast-uri intre clasele derivate
- ▶ Functii virtuale
- ▶ Clase abstracte (interfete)

# Alinieri in memorie pentru clasele derivate

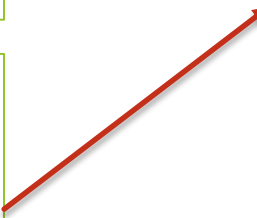
```
class A
{
public:
    int a1,a2,a3;
};
```

`sizeof(A)` = **12**

```
class B: public A
{
public:
    int b1,b2
};
```

`sizeof(B)` = **20**

Offset	Variabila	C1	C2
+ 0	A::a1	A	B
+ 4	A::a2		
+ 8	A::a3		
+ 12	B::b1		
+ 16	B::b2		



# Alinieri in memorie pentru clasele derivate

```
class A
{
public:
    int a1,a2,a3;
};
```

`sizeof(A)` = **12**

```
class B:
{
public:
    int b1,b2;
};
```

`sizeof(B)` = **8**

```
class C:public A,B
{
public:
    int c1,c2;
};
```

`sizeof(C)` = **28**

Offset	Variabila	C1	C2	C3
+ 0	A::a1	A	B	C
+ 4	A::a2			
+ 8	A::a3			
+ 12	B::b1			
+ 16	B::b2			
+20	C::c1			
+24	C::c2			

# Alinieri in memorie pentru clasele derivate

```
class A
{
public:
    int a1,a2,a3;
};
```

`sizeof(A)` = **12**

```
class B:
{
public:
    int b1,b2;
};
```

`sizeof(B)` = **8**

Alinierea in memorie in cazul claselor derivate se face in ordinea in care se face derivarea.

```
class C:public B,A
{
public:
    int c1,c2;
};
```

`sizeof(C)` = **28**

Offset	Variabila	C1	C2	C3
+ 0	B::b1	B	A	C
+ 4	B::b2			
+ 8	A::a1			
+ 12	A::a2			
+ 16	A::a3			
+20	C::c1			
+24	C::c2			

# Alinieri in memorie pentru clasele derivate

```
class A
{
public:
    int a1,a2,a3;
};
```

```
class B:public A
{
public:
    int b1,b2;
};
```

```
class C:public A,B
{
public:
    int c1,c2;
};
```

warning C4584: 'C' : base-class 'A' is already a base-class of 'B'.

Offset	Variabila	C1	C2	C3
+0	A::a1	A		C
+4	A::a2			
+8	A::a3			
+12	B::A::a1	B::A	B	
+16	B::A::a2			
+20	B::A::a3			
+24	B::b1			
+28	B::b2			
+32	C::c1			
+36	C::c2			

# Mostenire - multipla - ambiguitati


```
class A
{
public:
    int a1,a2,a3;
};
```

```
class B:public A
{
public:
    int b1,b2;
};
```

```
class C:public A,B
{
public:
    int c1,c2;
};
```

```
void main(void)
{
    C c;
    c.a1 = 10;
    c.A::a1 = 10;
    c.B::A::a1 = 20;

    c.b1 = 10;
}
```

- 
- ▶ Mostenire
  - ▶ Alinieri in memorie pentru clasele derivate
  - ▶ **Cast-uri intre clasele derivate**
  - ▶ Functii virtuale
  - ▶ Clase abstracte (interfete)



# Cast-uri intre clasele derivate

- ▶ Presupunand ca o clasa A este derivata dintr-o clasa B, atunci este posibil ca un obiect de tipul A sa fie convertit intr-un obiect de tipul B
- ▶ Acest lucru e firesc (A contine un obiect de tipul B).
- ▶ Regula de cast este urmatoarea:
  - ❖ Oricand se poate face un cast al unei clase catre una din clasele din care a fost derivate
  - ❖ Inversa trebuie specifica prin cast explicit
  - ❖ Daca se suprascrie operatorul de cast, atunci regulile de mai sus nu se mai aplica.

# Cast-uri intre clasele derivate

```
class A
{
public:
    int a1,a2,a3;
};
```

```
class B:public A
{
public:
    int b1,b2;
};
```

```
void main(void)
{
    B b;
    A* a = &b;
}
```

```
;B b;
lea    ecx,[b]
call   B::B
;A* a = &b;
lea    eax,[b]
mov    dword ptr [a],eax
```

# Cast-uri intre clasele derivate

```
class A
{
public:
    int a1,a2,a3;
};
```

```
class B
{
public:
    int b1,b2;
};
```

```
class C:public A,B
{
public:
    int c1,c2;
};
```

```
void main(void)
{
    C c;
    A* a;
    B* b;

    a = &c;
    b = &c;
}
```

## Disasm

```
a = &c;
    lea    eax,[c]
    mov    dword ptr [a],eax
b = &c;
    lea    eax,[c]
    test   eax,eax
    je     NULL_CAST
    lea    ecx,[c]
    add    ecx,0Ch
    mov    dword ptr [ebp-104h],ecx
    jmp    GOOD_CAST
NULL_CAST:
    mov    dword ptr [ebp-104h],0
GOOD_CAST:
    mov    edx,dword ptr [ebp-104h]
    mov    dword ptr [b],edx
_asm nop;
    nop
```

# Cast-uri intre clasele derivate

```
class A
{
public:
    int a1,a2,a3;
};
```

```
class B
{
public:
    int b1,b2;
};
```

```
class C:public A,B
{
public:
    int c1,c2;
};
```

```
void main(void)
{
    C c;
    A* a;
    B* b;
    C* c2;
    a = &c;
    b = &c;
    c2 = (C*)b;
}
```

## Disasm

```
c2 = (C*)b;
    cmp     dword ptr [b],0
    je      NULL_CAST
    mov     eax,dword ptr [b]
    sub     eax,0Ch
    mov     dword ptr [ebp-110h],eax
    jmp     GOOD_CAST
NULL_CAST:
    mov     dword ptr [ebp-110h],0
GOOD_CAST:
    mov     ecx,dword ptr [ebp-110h]
    mov     dword ptr [c2],ecx
```

- ▶ Mostenire
- ▶ Alinieri in memorie pentru clasele derivate
- ▶ Cast-uri intre clasele derivate
- ▶ **Functii virtuale**
- ▶ Clase abstracte (interfete)

# Functii virtuale

- ▶ Functiile virtuale sunt functii care se defines in clasa si sunt specifice instantei si nu clasei.
- ▶ Se definesc folosind cuvantul cheie **virtual**

## App.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};
class B: public A
{
public:
    int b1, b2;
    void Set() { printf("B"); }
};
void main()
{
    B b;
    b.Set();
}
```

- ❖ Codul alaturat afiseaza “B” pe ecran. Din punct de vedere al mostenirii, atat A cat si B au o functie cu numele Set, fara parametric si de tipul void.
- ❖ Din punct de vedere al compilatorului, functia Set din B ascunde functia Set din A.

# Functii virtuale

- ▶ Functiile virtuale sunt functii care se definesc in clasa si sunt specifice instantei si nu clasei.
- ▶ Se definesc folosind cuvantul cheie **virtual**

## App.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};
class B: public A
{
public:
    int b1, b2;
    void Set() { printf("B"); }
};
void main()
{
    B b;
    A* a = &b;
    a->Set();
}
```

- ❖ In cazul de fata, codul va afisa insa "A" pentru ca se face cast-ul la o clasa de tipul A, iar apelul functiei Set se va face din clasa A.
- ❖ Ce putem face insa daca vrem sa pastram functionalitatea unei functii intre cast-uri ?

# Functii virtuale

```
class A
{
public:
    int x,y;
    virtual int Calcul() { return 0; }
};
```

`sizeof(Test)` = **12**

?	?	?	?	x	x	x	x	y	y	y	y								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19



# Funcții Virtuale

- Fie urmatorul cod:

## App.cpp

```
class A
{
public:
    int x, y;
    virtual int Calcul() {
        return 0;
    }
};

void main()
{
    A a;
    a.x = 1;
    a.y = 2;
    a.Calcul();
}
```

## Disasm

```
A a;
    lea     ecx,[a]
    call    A::A
a.x = 1;
    mov     dword ptr [ebp-10h],1
a.y = 2;
    mov     dword ptr [ebp-0Ch],2
a.Calcul();
    lea     ecx,[a]
    call    A::Calcul
```

# Functii Virtuale

- Fie urmatorul cod:

## App.cpp

```
class A
{
public:
    int x, y;
    virtual int Calcul() {
        return 0;
    }
};

void main()
{
    A a;
    a.x = 1;
    a.y = 2;
    A* a2 = &a;
    a2->Calcul();
}
```

## Disasm

```
A a;
    lea     ecx,[a]
    call    A::A
a.x = 1;
    mov     dword ptr [ebp-10h],1
a.y = 2;
    mov     dword ptr [ebp-0Ch],2
A* a2 = &a;
    lea     eax,[a]
    mov     dword ptr [a2],eax
a2->Calcul();
    mov     eax,dword ptr [a2]
    mov     edx,dword ptr [eax]
    mov     ecx,dword ptr [a2]
    mov     eax,dword ptr [edx]
    call    eax
```

# Functii Virtuale

- Fie urmatorul cod:

## App.cpp

```
class A
{
public:
    int x, y;
    virtual int Calcul() {
        return 0;
    }
};

void main()
{
    A a;
    a.x = 1;
    a.y = 2;
    A* a2 = &a;
    a2->Calcul();
}
```

## Pseudo-C

```
struct A_VirtualFunctions
{
    int (*Calcul) ();
};

class A
{
public:
    A_VirtualFunctions *vfPtr;
    int x, y;
    int A_Calcul() { return 0; }
};

A_VirtualFunctions Global_A_vfPtr;
A_vfPtr.Calcul = &A::A_Calcul;

void main()
{
    A a;
    a.vfPtr = Global_A_vfPtr;
    a.x = 1;
    a.y = 2;
    A* a2 = &a;
    a2->vfptr->Calcul();
}
```

# Functii Virtuale

- Fie urmatorul cod:

## App.cpp

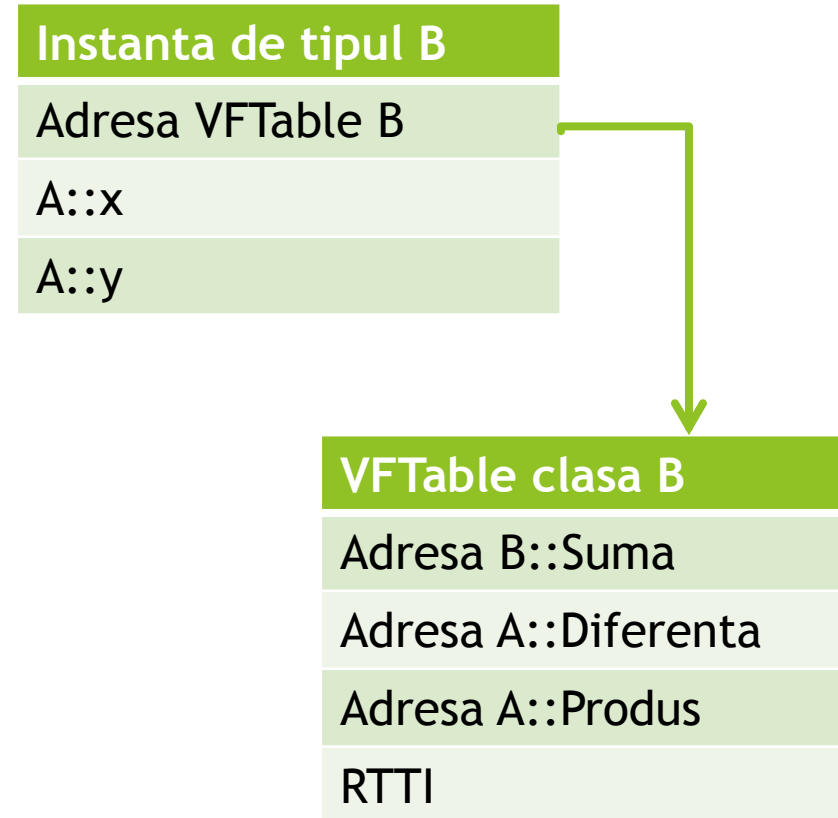
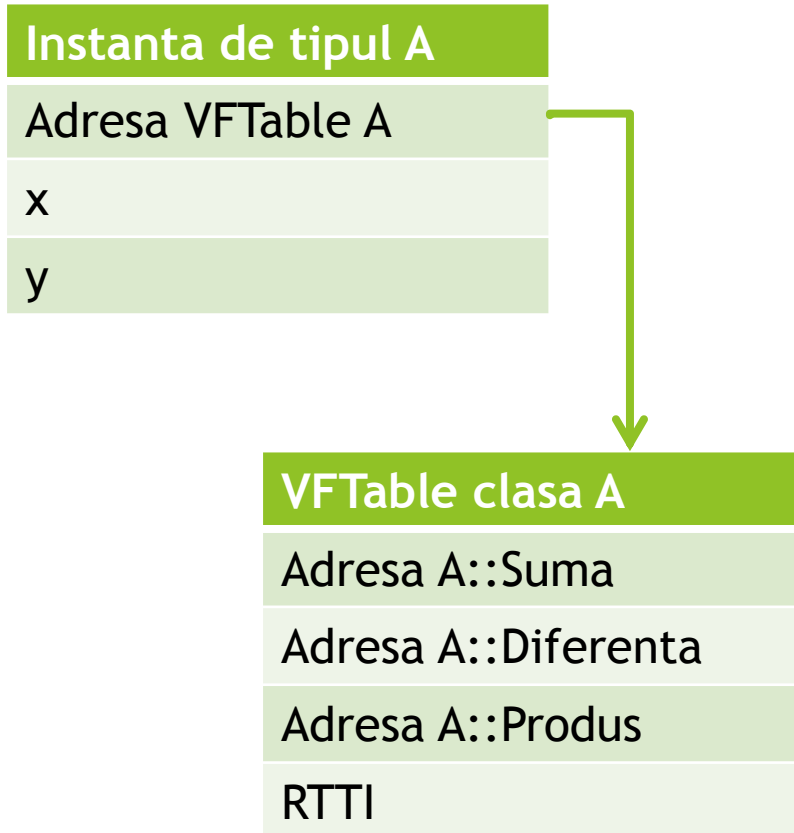
```
class A
{
public:
    int x, y;
    virtual int Suma() { return x + y; }
    virtual int Diferenta() { return x - y; }
    virtual int Produs() { return x*y; }
};
class B : public A
{
public:
    int Suma() { return 1; }
};


void main()
{
    B b;
    b.x = 1;
    b.y = 2;

    A* a;
    a = &b;
    int x = a->Suma();
}
```

- ❖ Dupa executia codului, valoarea lui “x” va fi 1.
- ❖ Chiar daca se apeleaza functia Suma, pe un pointer de tipul A\*, obiectul in cauza este in realitate un B, si cum functia Suma este virtuala, se apeleaza functia Suma din B si nu din A.

# Functii Virtuale



- 
- ▶ Mostenire
  - ▶ Alinieri in memorie pentru clasele derivate
  - ▶ Cast-uri intre clasele derivate
  - ▶ Functii virtuale
  - ▶ Clase abstracte (interfete)

# Clase abstracte

- ▶ In C++ exista posibilitatea definirii unei asa numite functii virtuale pure (adaugand =0) la sfarsitul definitiei acelei functii.
- ▶ Existenta unei functii virtuale pure intr-o clasa transforma acea clasa intr-o clasa abstracta (clasa care nu poate fi instantiata). Principiile POO viitoare au denumit această clasa apartea interfata.
- ▶ O functie virtuala pura obliga pe cel care deriva din clasa din care face parte sa asigure si o implementare pentru acea functie.

## App.cpp

```
class A
{
public:
    int a1, a2, a3;
    virtual void Set() = 0;
};
void main()
{
    A a;
}
```

- ❖ Codul alaturat nu va compila pentru ca clasa A are o functie virtuala pura si nu se pot instantia obiecte pentru clase abstracte.