

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

P00

Curs-4

Gavrilut Dragos

- ▶ Mostenirea
- ▶ Ce este o funcție virtuală ?
- ▶ Utilitatea funcțiilor virtuale
- ▶ Cum modelează compilatorul comportamentul funcțiilor virtuale
- ▶ Covarianti
- ▶ Clase abstracte
- ▶ Alinieri în memorie pentru clasele derivate
- ▶ Cast-uri între clasele derivate

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 insa 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

Ce este o funcție virtuală ?

Ce este o funcție virtuala ?

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 alăturat afișează “**B**” pe ecran. Din punct de vedere al moștenirii, atât A cat si B au o funcție cu numele Set cu aceeași semnătură.
- ❖ Funcția Set din clasa B ascunde funcția Set din clasa A.

Ce este o funcție virtuala ?

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 afișa însă “**A**” pentru ca se face cast-ul la o clasa de tipul A, si se va folosi funcția Set din clasa A.
- ❖ In realitate, pointerul “a” punctează la un obiect de tipul B.
- ❖ Ce putem face însă daca vrem sa păstrăm funcționalitatea unei funcții între cast-uri ?

Ce este o funcție virtuală ?

App.cpp

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

- ❖ Soluția este să utilizăm cuvântul cheie **“virtual”** în definiția unei funcții
- ❖ În urma execuției se va afișa **“B”**
- ❖ Prin utilizarea cuvântului cheie **virtual**, o funcție membru devine o proprietate a instanței și nu a clasei.

Utilitatea funcțiilor virtuale

Utilitatea funcțiilor virtuale

Funcțiile virtuale au mai multe roluri:

- ▶ Polimorfism
- ▶ In de-alocarea memorie (destructorul virtual)
- ▶ Tehnici anti-debugging / anti-analiza

Utilitatea funcțiilor virtuale

Polimorfismul este modalitatea prin care mai multe obiecte care au o interfața comună (în cazul C++ derivate sunt din aceeași clasă) pot fi convertite (cast-ate) la clasa de bază păstrând-și funcționalitatea specifică funcțiilor lor.

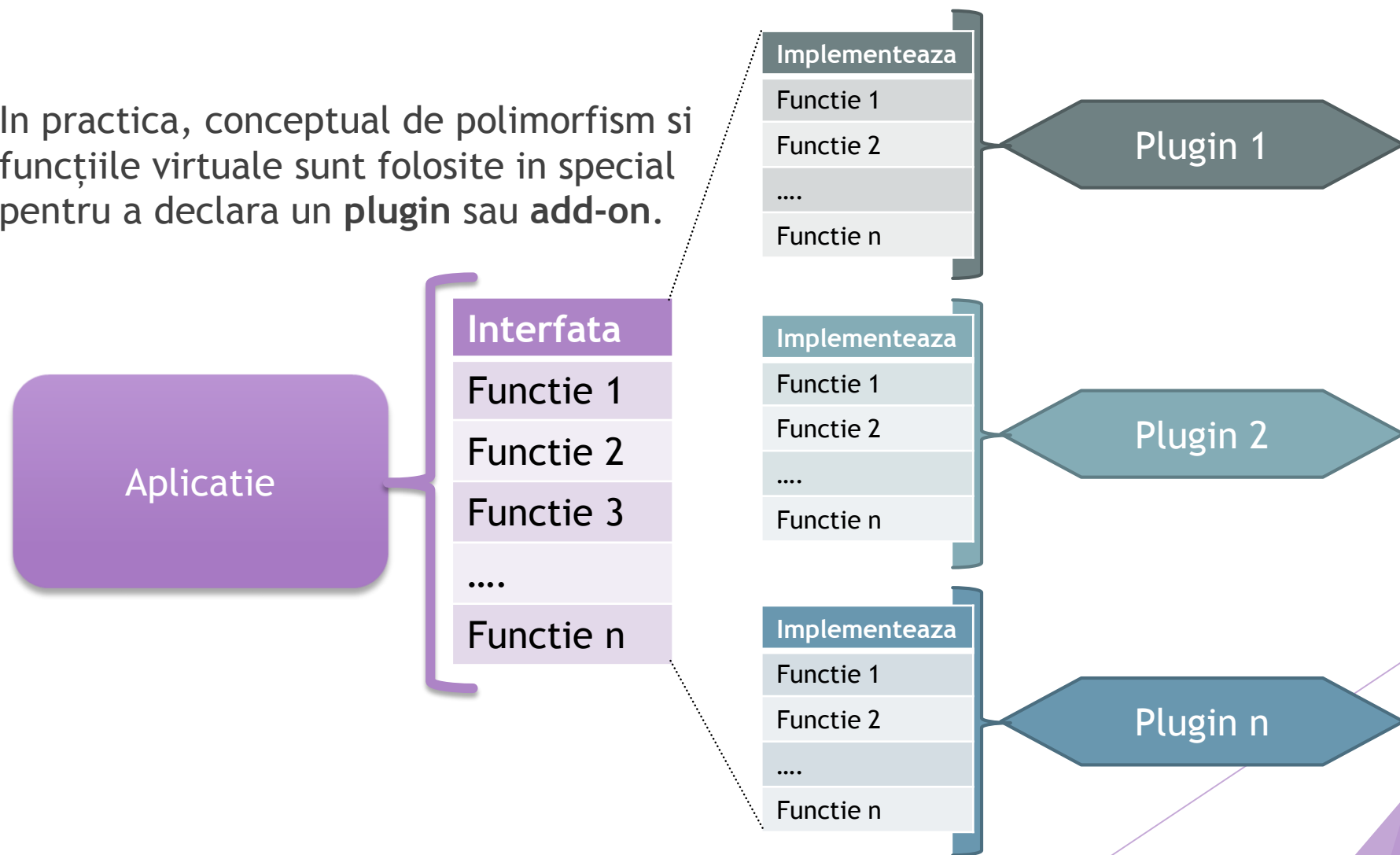
App-1.cpp

```
class Figura {  
    public: virtual void Draw() { printf("Figura"); }  
};  
class Cerc: public Figura {  
    public: void Draw() { printf("Cerc"); }  
};  
class Patrat: public Figura {  
    public: void Draw() { printf("Patrat"); }  
};  
void main()  
{  
    Figura *f[2];  
    f[0] = new Cerc();  
    f[1] = new Patrat();  
    for (int index = 0; index < 2; index++)  
        f[index]->Draw();  
}
```

- ❖ După execuția codului pe ecran se va afișa **"Cerc"** și **"Patrat"**.

Utilitatea funcțiilor virtuale

În practică, conceptual de polimorfism și funcțiile virtuale sunt folosite în special pentru a declara un **plugin** sau **add-on**.



Utilitatea funcțiilor virtuale

Particular pentru unele limbaje gen C++, utilizarea polimorfismului poate conduce la situații în care se dealoca un pointer de tipul clasei de baza ca în exemplul următor.

App-1.cpp

```
class Figura {
    public: virtual void Draw() { printf("Figura"); }
    public: ~Figura() { printf("Delete Figura\n"); }
};
class Cerc: public Figura {
    public: void Draw() { printf("Cerc"); }
    public: ~Cerc() { printf("Delete Cerc"); }
};
class Patrat: public Figura {
    public: void Draw() { printf("Patrat"); }
    public: ~Patrat() { printf("Delete Patrat"); }
};
void main() {
    Figura *f[2];
    f[0] = new Cerc();
    f[1] = new Patrat();
    for (int index = 0; index < 2; index++)
        delete (f[index]);
}
```

- ❖ După execuția codului pe ecran se va afișa **“Delete figura”** și **“Delete figura”**.
- ❖ Ce se întâmplă dacă într-una din clasele derivate mai alocăm memorie? Acea memorie nu va fi dealocată în acest caz.

Utilitatea funcțiilor virtuale

Particular pentru unele limbaje gen C++, utilizarea polimorfismului poate conduce la situații în care se dealoca un pointer de tipul clase de baza ca în exemplul următor.

App-1.cpp

```
class Figura {
    public: virtual void Draw() { printf("Figura"); }
    public: virtual ~Figura() { printf("Delete Figura\n"); }
};
class Cerc: public Figura {
    public: void Draw() { printf("Cerc"); }
    public: ~Cerc() { printf("Delete Cerc"); }
};
class Patrat: public Figura {
    public: void Draw() { printf("Patrat"); }
    public: ~Patrat() { printf("Delete Patrat"); }
};
void main() {
    Figura *f[2];
    f[0] = new Cerc();
    f[1] = new Patrat();
    for (int index = 0; index < 2; index++)
        delete (f[index]);
}
```

- ❖ Soluția este să declaram destructorul din clasa Figura ca fiind unul virtual. În acest fel, la apelul lui **delete**, se vor apela și destructorii claselor **Cerc** și **Patrat**.
- ❖ Codul va afișa:
Delete Cerc
Delete Figura
Delete Patrat
Delete Figura

Cum modelează compilatorul
comportamentul funcțiilor virtuale

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Fie următoarele programe care diferă doar prin utilizarea cuvântului “**virtual**” în cazul celui de-al doilea program.

App-1.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};
void main()
{
    printf("%d",sizeof(A));
}
```

App-2.cpp

```
class A
{
public:
    int a1, a2, a3;
    virtual void Set() { printf("A"); }
};
void main()
{
    printf("%d",sizeof(A));
}
```

- ❖ În urma execuției App-1 va afișa “**12**” iar App-2 va afișa “**16**” (pe o arhitectura pe 32 de biți). Pe o arhitectura pe 64 de biți App-2 va afișa “**24**”
- ❖ DE CE ?

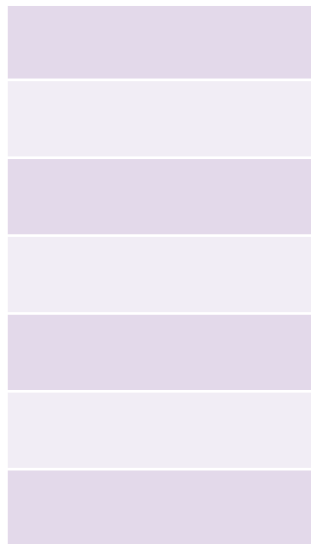
Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Utilizarea cuvântului “**virtual**” la măcar una dintre metodele unei clase adaugă la compilare încă un membru (la începutul clasei) care este un pointer (vfptr) către o lista cu toate funcțiile virtuale ale acelei clase.

App-1.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};
void main()
{
    A a;
}
```

Memorie



Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Utilizarea cuvântului “**virtual**” la măcar una dintre metodele unei clase adaugă la compilare încă un membru (la începutul clasei) care este un pointer (vfptr) către o lista cu toate funcțiile virtuale ale acelei clase.

App-1.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};
void main()
{
    A a;
}
```

Memorie

...

A::Set()

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Utilizarea cuvântului “**virtual**” la măcar una dintre metodele unei clase adaugă la compilare încă un membru (la începutul clasei) care este un pointer (vfptr) către o lista cu toate funcțiile virtuale ale acelei clase.

App-1.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};
void main()
{
    A a;
}
```

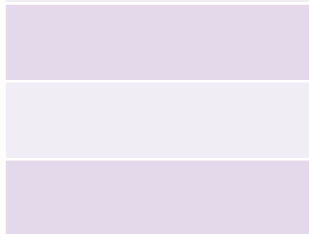
Memorie

...

A::Set()

...

main()



Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Utilizarea cuvântului “**virtual**” la măcar una dintre metodele unei clase adaugă la compilare încă un membru (la începutul clasei) care este un pointer (vfptr) către o lista cu toate funcțiile virtuale ale acelei clase.

App-1.cpp

```
class A
{
public:
    int a1, a2, a3;
    void Set() { printf("A"); }
};
void main()
{
    A a;
}
```

Memorie

...

A::Set()

...

main()

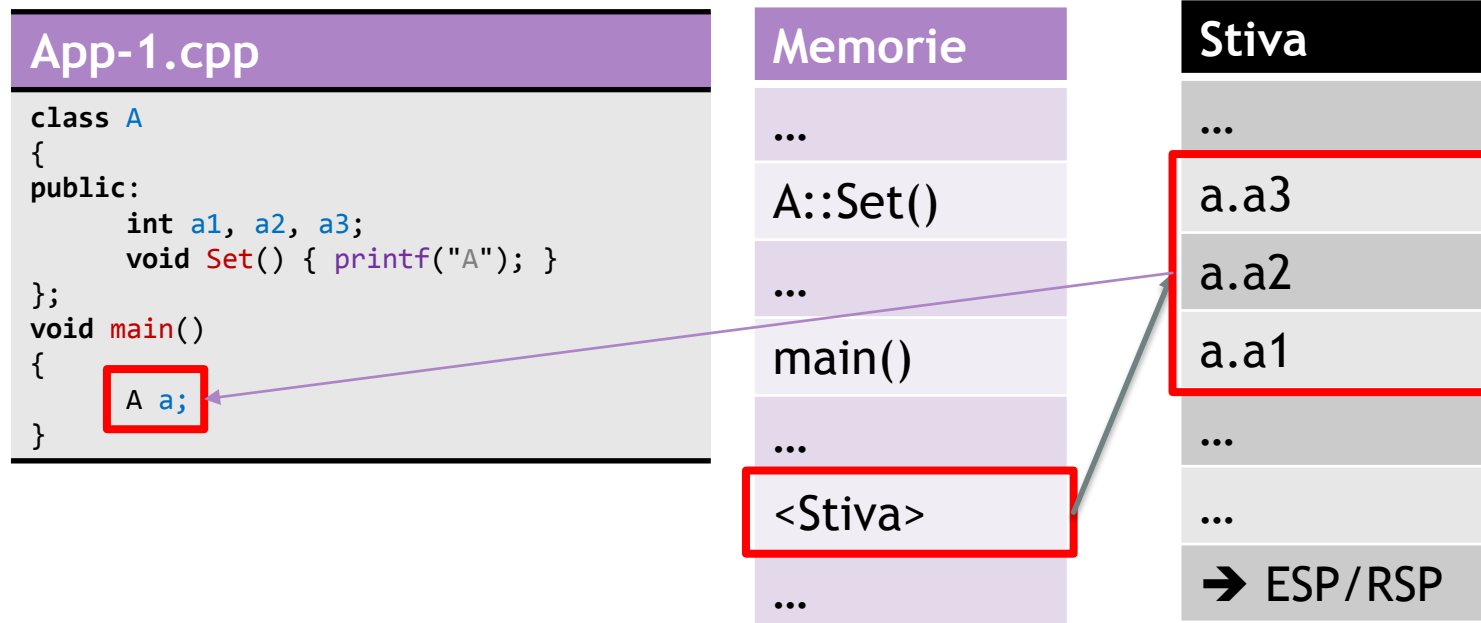
...

<Stiva>

...

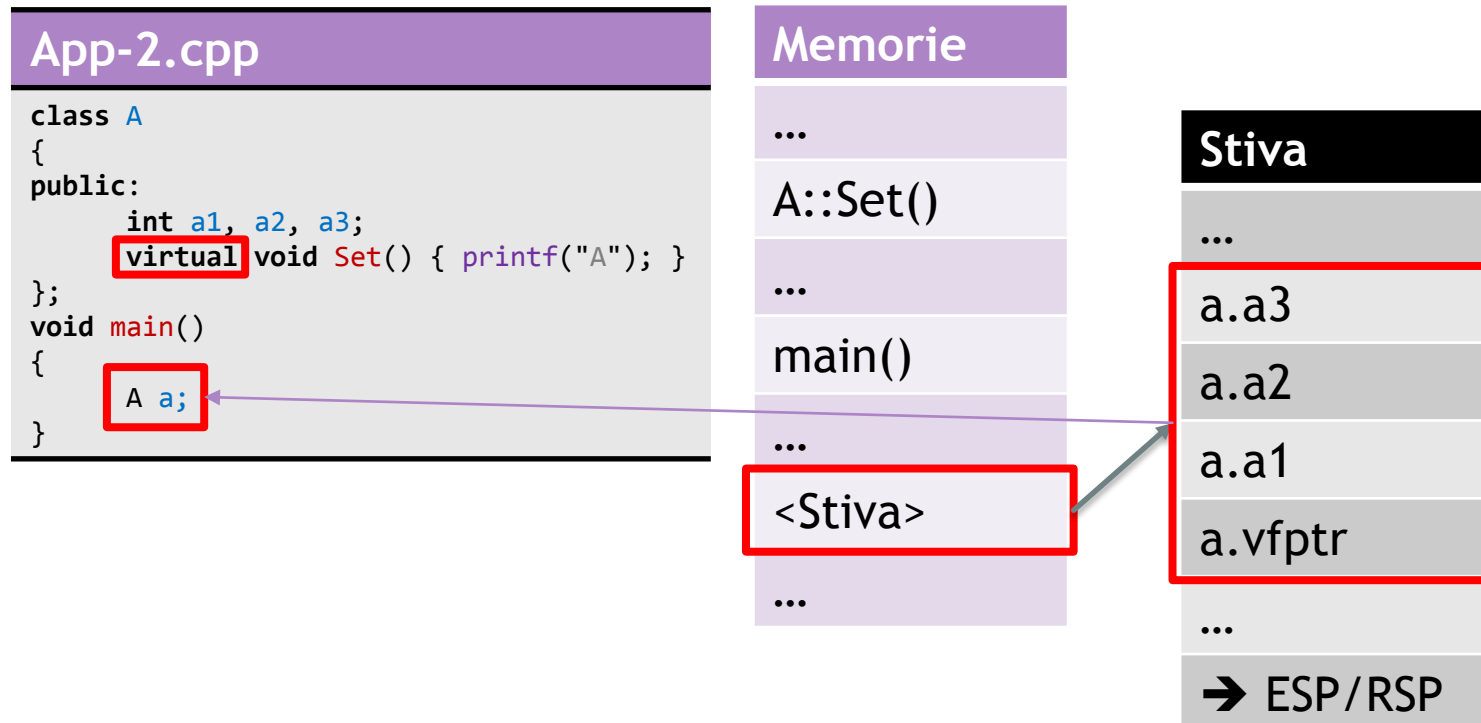
Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Utilizarea cuvântului “**virtual**” la măcar una dintre metodele unei clase adaugă la compilare încă un membru (la începutul clasei) care este un pointer (vfptr) către o lista cu toate funcțiile virtuale ale acelei clase.



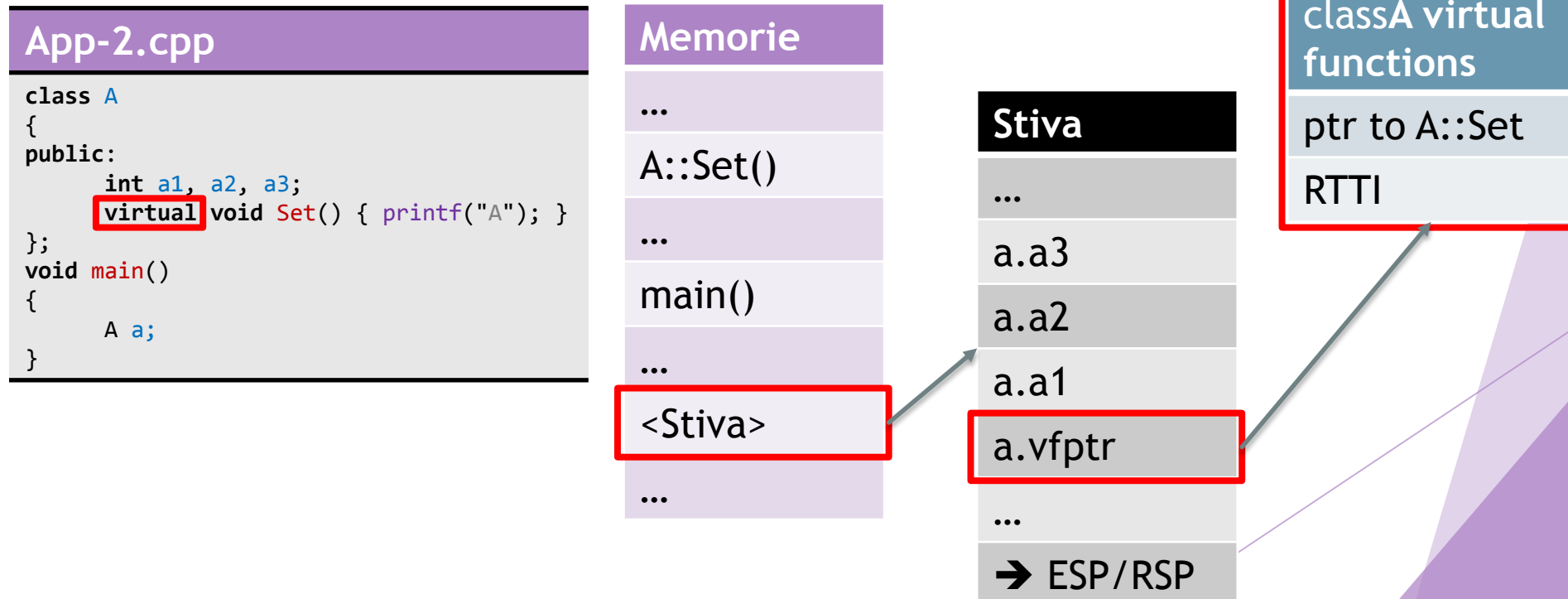
Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Utilizarea cuvântului “**virtual**” la măcar una dintre metodele unei clase adaugă la compilare încă un membru (la începutul clasei) care este un pointer (vfptr) către o lista cu toate funcțiile virtuale ale acelei clase.



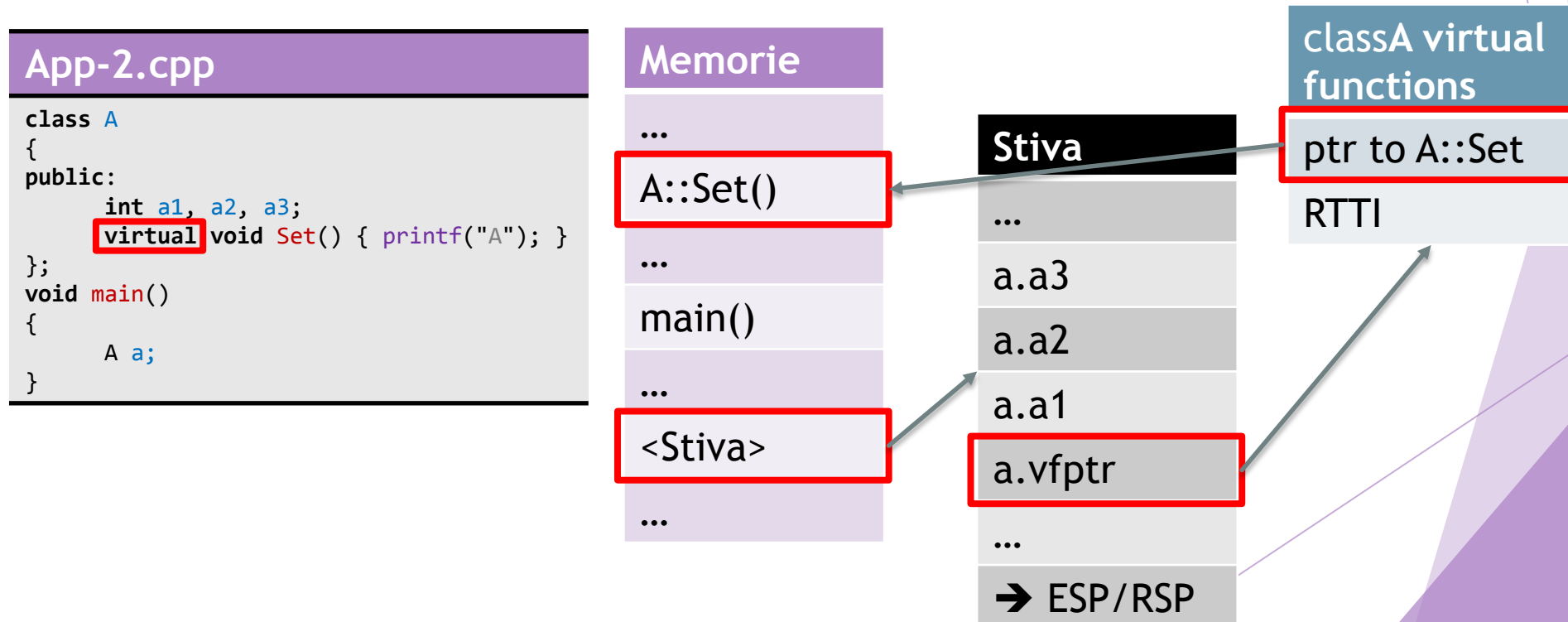
Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Utilizarea cuvântului “**virtual**” la măcar una dintre metodele unei clase adaugă la compilare încă un membru (la începutul clasei) care este un pointer (vfptr) către o lista cu toate funcțiile virtuale ale acelei clase.



Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Utilizarea cuvântului “**virtual**” la măcar una dintre metodele unei clase adaugă la compilare încă un membru (la începutul clasei) care este un pointer (vfptr) către o lista cu toate funcțiile virtuale ale acelei clase.



Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ De asemenea compilatorul fie creează un constructor implicit (daca nu este ceva definit) fie modifica constructorul/constructorii existenți ca sa seteze pointerul **vfptr** la lista de funcții virtuale.

App.cpp

```
class A
{
public:
    int x, y;
    int Calcul() { return x+y; }
};
void main()
{
    A a;
    a.x = 1;
    a.y = 2;
}
```

Disasm

```
A a;
a.x = 1;
mov     dword ptr [ebp-12],1
a.y = 2;
mov     dword ptr [ebp-8],2
```

- ❖ In cazul de fata, nu este creat un constructor automat si cum nici nu este specificat unul in clasa, nu se apelează nimic pentru inițializarea obiectului.

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ De asemenea compilatorul fie creează un constructor implicit (daca nu este ceva definit) fie modifica constructorul/constructorii existenți ca sa seteze pointerul **vfptr** la lista de funcții virtuale.

App.cpp

```
class A
{
public:
    int x, y;
    int Calcul() { return x+y; }
    A() { x = y = 0; }
};
void main()
{
    A a;
    a.x = 1;
    a.y = 2;
}
```

Disasm

```
A a;
lea     ecx,[ebp-16]
call    A::A
a.x = 1;
mov     dword ptr [ebp-16],1
a.y = 2;
mov     dword ptr [ebp-12],2
```

- ❖ In cazul de fata, este apelat constructorul default creat in clasa A.

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ De asemenea compilatorul fie creează un constructor implicit (daca nu este ceva definit) fie modifica constructorul/constructorii existenți ca sa seteze pointerul **vfp** la lista de funcții virtuale.

App.cpp	Disasm (A::A)
<pre>class A { public: int x, y; int Calcul() { return x+y; } A() { x = y = 0; } }; void main() { A a; a.x = 1; a.y = 2; }</pre>	<pre>push ebp mov ebp,esp mov dword ptr [ebp-8],ecx mov eax,dword ptr [ebp-8] mov dword ptr [eax+4],0 // this->y = 0 mov ecx,dword ptr [ebp-8] mov dword ptr [ecx],0 // this->x = 0 mov eax,dword ptr [ebp-8] pop ebp ret</pre>

- ❖ In cazul de fata, este apelat constructorul default creat in clasa A. Codul din constructorul default nu este modificat.

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ De asemenea compilatorul fie creează un constructor implicit (daca nu este ceva definit) fie modifica constructorul/constructorii existenți ca sa seteze pointerul **vfptr** la lista de funcții virtuale.

App.cpp

```
class A
{
public:
    int x, y;
    virtual int Calcul() {return x+y;}
};
void main()
{
    A a;
    a.x = 1;
    a.y = 2;
}
```

Disasm

```
A a;
lea     ecx,[ebp-20]
call    A::A
a.x = 1;
mov     dword ptr [ebp-16],1
a.y = 2;
mov     dword ptr [ebp-12],2
```

- ❖ In cazul de fata, deși nici un constructor default nu a fost definit, compilatorul creează unul implicit si îl apelează la inițializare.

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ De asemenea compilatorul fie creează un constructor implicit (daca nu este ceva definit) fie modifica constructorul/constructorii existenți ca sa seteze pointerul **vfptr** la lista de funcții virtuale.

App.cpp	Disasm
<pre>class A { public: int x, y; virtual int Calcul() {return x+y;} }; void main() { A a; a.x = 1; a.y = 2; }</pre>	<pre>A a; lea ecx,[ebp-20] call A::A a.x = 1; mov ecx,[ebp-16],1 mov ecx,[ebp-12],2</pre>
	<pre>push ebp mov ebp,esp mov dword ptr [ebp-8],ecx mov eax,dword ptr [ebp-8] mov dword ptr [eax], A-virtual-fnc-list mov eax,dword ptr [ebp-8] mov esp,ebp pop ebp ret</pre>

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ De asemenea compilatorul fie creează un constructor implicit (daca nu este ceva definit) fie modifica constructorul/constructorii existenți ca sa seteze pointerul **vfptr** la lista de funcții virtuale.

```
App.cpp
class A
{
public:
    int x, y;
    virtual int Calcul() {return x+y;}
    A() { x = y = 0; }
};
void main()
{
    A a;
    a.x = 1;
    a.y = 2;
}
```

```
Disasm
A a;
lea     ecx,[ebp-20]
call    A::A
a.x = 1;
mov     dword ptr [ebp-16],1
a.y = 2;
mov     dword ptr [ebp-12],2
```

- ❖ Daca definim un constructor, acesta va fi modificat pentru a include si setarea **vfptr**

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ De asemenea compilatorul fie creează un constructor implicit (daca nu este ceva definit) fie modifica constructorul/constructorii existenți ca sa seteze pointerul **vfptr** la lista de funcții virtuale.

```
App.cpp
class A
{
public:
    int x, y;
    virtual int Calcul() {return x+y;}
    A() { x = y = 0; }
};
void main()
{
    A a;
    a.x = 1;
    a.y = 2;
}
```

Disasm A::A

```
push    ebp
mov     ebp,esp
mov     dword ptr [ebp-8],ecx
mov     eax,dword ptr [ebp-8]
mov     dword ptr [eax],addr virt fnc
mov     eax,dword ptr [ebp-8]
mov     dword ptr [eax+8],0
mov     ecx,dword ptr [ebp-8]
mov     dword ptr [ecx+4],0
mov     eax,dword ptr [ebp-8]
mov     esp,ebp
pop     ebp
ret
```

- ❖ Codul colorat in **albastru** este adăugat automat de către compilator pentru a inițializa **vfptr**.

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Codul adițional adăugat de compilator se aplica pentru orice constructor (inclusive constructorul de copiere)

App.cpp

```
class A
{
public:
    int x, y;
    virtual int Calcul() {return x+y;}
    A() { x = y = 0; }
    A(const A& a) { x = a.x; y = a.y; }
};

void main()
{
    A a;
    A a2 = a;
}
```

Declarația apelează constructorul de copiere din A, constructor care este modificat ca să seteze și valoarea **vfptr**

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Pentru cazul operatorilor însă **nu se adaugă** modificări. De exemplu în cazul operatorului de asignare, nu vom avea cod în plus adăugat.

App.cpp

```
class A
{
public:
    int x, y;
    virtual int Calcul() {return x+y;}
    A() { x = y = 0; }
    A& operator = (A &a) { x = a.x; y = a.y; }
};

void main()
{
    A a;
    A a2;
    a2 = a;
}
```

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Apelul funcției virtuale prin referință din **vfptr** este făcut doar dacă instanța obiectului care face apelul e un pointer sau o referință.

App.cpp	Disasm
<pre>class A { public: int x, y; virtual int Calcul() {return x+y;} A() { x = y = 0; } }; void main() { A a; a.x = 1; a.y = 2; a.Calcul(); }</pre>	<pre>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</pre>

- ❖ Chiar dacă funcția **Calcul** este virtuală, nu are sens pentru compilator să facă apelul folosind **vfptr** în acest caz, pentru că nefiind un pointer `a.Calcul` nu poate puncta decât la o singură funcție.

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Apelul funcției virtuale prin referința din **vfptr** este făcut doar dacă instanța obiectului care face apelul e un pointer sau o referință.

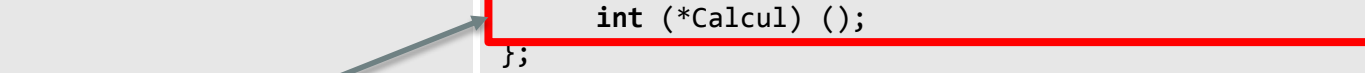
App.cpp	Disasm
<pre>class A { public: int x, y; virtual int Calcul() {return x+y;} A() { x = y = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>A a; lea ecx,[a] call A::A a.x = 1; mov dword ptr [ebp-10h],1 a.y = 2; mov dword ptr [ebp-0Ch],2 A* a2 = &a; lea eax,[a] 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</pre>

- ❖ In cazul de fata apelul se face folosind **vfptr**

Cum modelează compilatorul comportamentul funcțiilor virtuale

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { };</pre>

Cum modelează compilatorul comportamentul funcțiilor virtuale

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); };</pre> 

Cum modelează compilatorul comportamentul funcțiilor virtuale

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: int x;</pre>

Cum modelează compilatorul comportamentul funcțiilor virtuale

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x;</pre>

Cum modelează compilatorul comportamentul funcțiilor virtuale

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x: int A_Calcul() { return 0; }</pre>

Cum modelează compilatorul comportamentul funcțiilor virtuale

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x; int A_Calcul() { return 0; } }; A_VirtualFunctions Global_A_vfPtr; Global_A_vfPtr.Calcul = &A::A_Calcul;</pre>

Cum modelează compilatorul comportamentul funcțiilor virtuale

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x; int A Calcul() { return 0; } A() { x = 0; } }; A_VirtualFunctions Global_A_vfPtr; Global_A_vfPtr.Calcul = &A::A_Calcul;</pre>

Cum modelează compilatorul comportamentul funcțiilor virtuale

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x; int A_Calcul() { return 0; } A() { vfPtr = &Global_A_vfPtr; x = 0; } }; A_VirtualFunctions Global_A_vfPtr; Global_A_vfPtr.Calcul = &A::A_Calcul;</pre>

Cum modelează compilatorul comportamentul funcțiilor virtuale

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x; int A_Calcul() { return 0; } A() { vfPtr = &Global_A_vfPtr; x = 0; } }; A_VirtualFunctions Global_A_vfPtr; Global_A_vfPtr.Calcul = &A::A_Calcul; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; }</pre>

Cum modelează compilatorul comportamentul funcțiilor virtuale

App.cpp	Pseudo C/C++ Code
<pre>class A { public: int x; virtual int Calcul() {return 0;} A() { x = 0; } }; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->Calcul(); }</pre>	<pre>struct A_VirtualFunctions { int (*Calcul) (); }; class A { public: A_VirtualFunctions *vfPtr; int x; int A_Calcul() { return 0; } A() { vfPtr = &Global_A_vfPtr; x = 0; } }; A_VirtualFunctions Global_A_vfPtr; Global_A_vfPtr.Calcul = &A::A_Calcul; void main() { A a; a.x = 1; a.y = 2; A* a2 = &a; a2->vfPtr->Calcul(); }</pre>

Cum modelează compilatorul comportamentul funcțiilor virtuale

❖ **vfptr**-ul e doar un pointer. Prin urmare, poate fi schimbat în timpul execuției.

App.cpp

```
class A
{
public:
    int x;
    virtual void Print() { printf("A"); }
};
class B
{
public:
    int x;
    virtual void Print() { printf("B"); }
};
void main()
{
    A a;
    B b;
    A* a2 = &a;
    a.Print();
    a2->Print();
}
```

❖ Codul alăturat afișează “**AA**”.
Prima dată prin apelul funcției **Print** direct a doua oară prin apelul funcției **Print** din **vfptr**.

Cum modelează compilatorul comportamentul funcțiilor virtuale

❖ **vfptr**-ul e doar un pointer. Prin urmare, poate fi schimbat în timpul execuției.

App.cpp

```
class A
{
public:
    int x;
    virtual void Print() { printf("A"); }
};
class B
{
public:
    int x;
    virtual void Print() { printf("B"); }
};
void main()
{
    A a;
    B b;
    memcpy(&a, &b, sizeof(void*));
    A* a2 = &a;
    a.Print();
    a2->Print();
}
```

❖ Codul alăturat afișează “**AB**”. Prin utilizarea lui **memcpy** am suprascris **vfptr**-ul pentru obiectului “a” cu cel a lui “b”. Cum funcția **Print** are aceeași semnătură în ambele clase, codul va funcționa corect în timpul execuției.

Cum modelează compilatorul comportamentul funcțiilor virtuale

❖ **vfptr**-ul e doar un pointer. Prin urmare, poate fi schimbat în timpul execuției.

App.cpp

```
class A
{
public:
    int x;
    virtual void Print() { printf("A"); }
};
class B
{
public:
    int x;
    virtual void Print() { printf("B"); }
};
void main()
{
    A a;
    B b;
    memcpy(&a, &b, sizeof(void*));
    A* a2 = &a;
    A a3 = (*a2);
    A *a4 = &a3;
    a4->Print();
}
```

- ❖ Constructorul de copiere nu copie și **vfptr**-ul ci doar îl setează la valoarea implicit pentru clasa în cauză. Chiar dacă "a2" are un alt **vfptr**, a3 va avea **vfptr**-ul specific clasei A
- ❖ Codul alăturat va afișa "A"

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Dacă o funcție virtuală nu e suprascrisă se păstrează și la fel în clasa derivată.

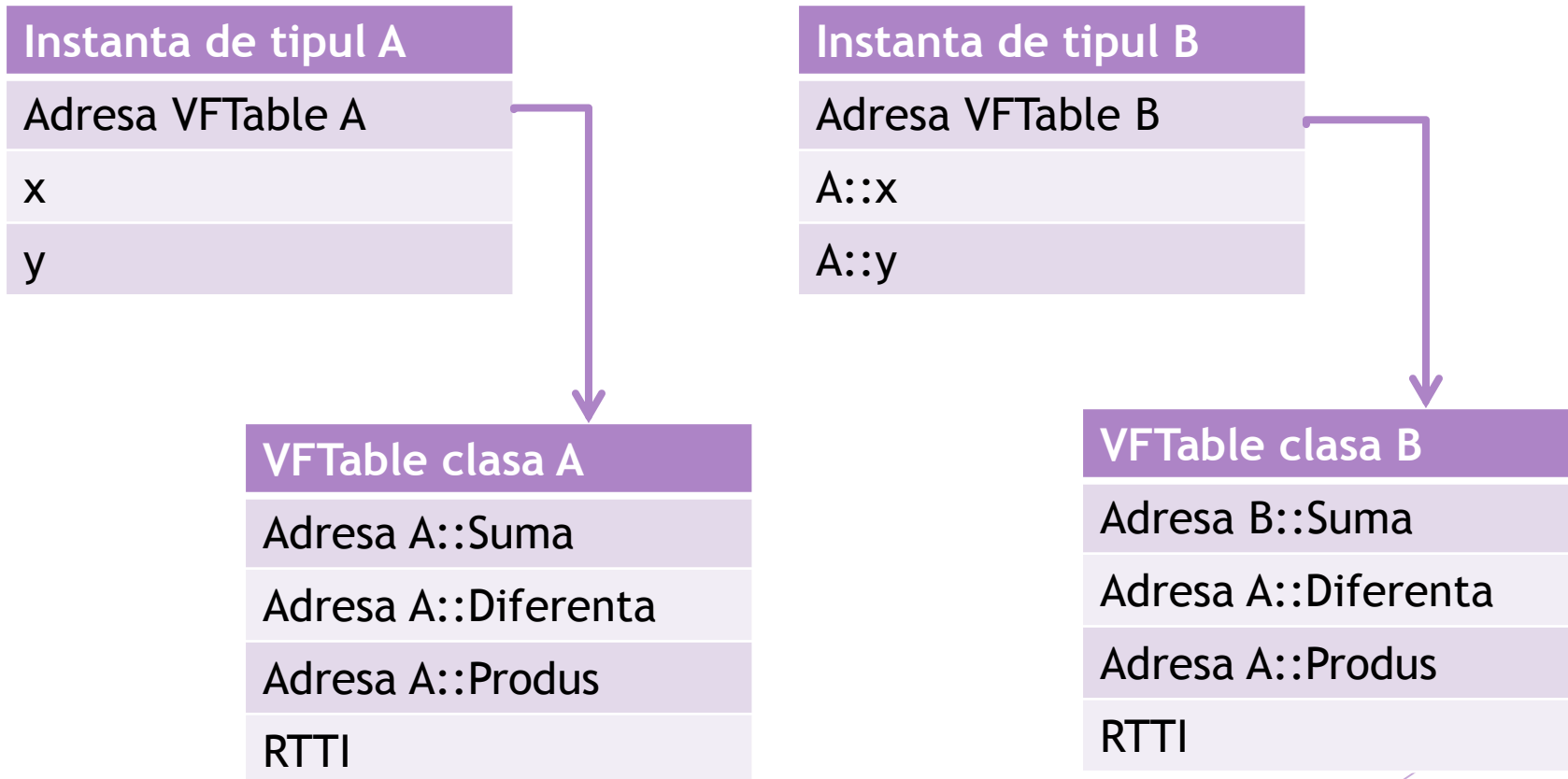
App.cpp

```
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();
}
```

- ❖ După execuția codului, valoarea lui “**x**” va fi 1.
- ❖ Chiar dacă se apelează funcția Suma, pe un pointer de tipul A*, obiectul în cauză este în realitate un B, și cum funcția Suma este virtuală, se apelează funcția Suma din B și nu din A.

Cum modelează compilatorul comportamentul funcțiilor virtuale



Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Clasa derivate poate sa adauge si ea la rândul ei alte funcții virtuale.

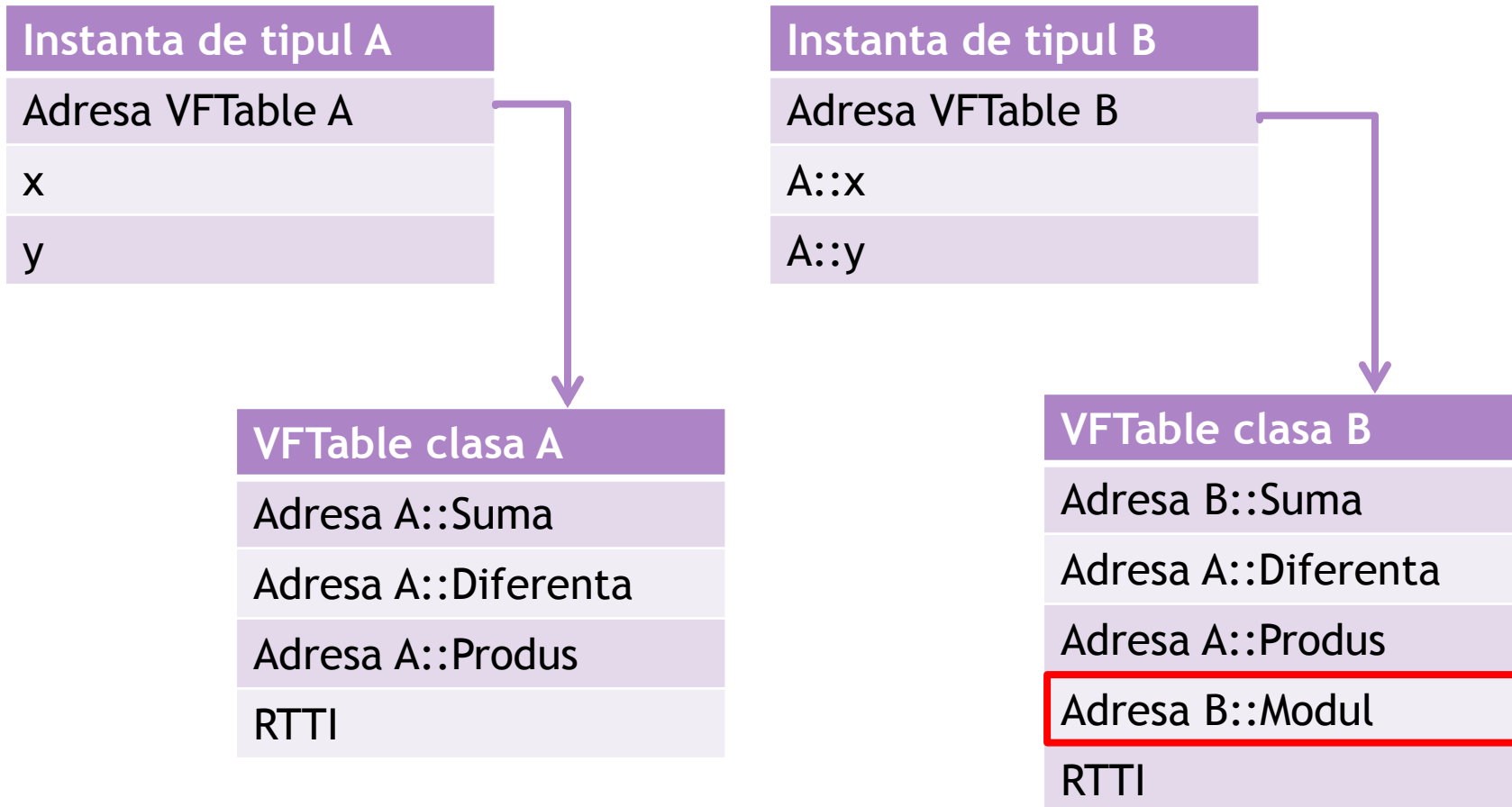
App.cpp

```
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; }
    virtual int Modul() { return 0; }
};

void main()
{
}
```

- ❖ In cazul de alăturat, clasa B mai adăuga si funcția Modul (de asemenea virtuala) care nu exista in clasa A din care este derivate B.
- ❖ Clasele care sunt derivate din B vor putea utiliza si funcția Modul.

Cum modelează compilatorul comportamentul funcțiilor virtuale



Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Atunci când o clasă deriva din două clase virtuale, se creează două tabele `vfptr` (una după alta) care țin adresele funcțiilor virtuale din clasele de bază.

App.cpp

```
class A {
public:
    int a1;
    virtual int Suma() { return 1; }
    virtual int Diferenta() { return 2; }
};
class B {
public:
    int b1,b2;
    virtual int Inmultire() { return 3; }
    virtual int Impartire() { return 4; }
};
class C : public A, public B {
public:
    int x, y;
};
void main() {
    C c;
    C *cptr = &c;
    cptr->Impartire();
    cptr->Diferenta();
}
```

Disasm

```
    cptr->Impartire();
    mov     ecx,dword ptr [cptr]
    add     ecx,8 //this pentru tipul B
    mov     eax,dword ptr [cptr]
    mov     edx,dword ptr [eax+8]
    mov     eax,dword ptr [edx+4]
    call    eax
    cptr->Diferenta();
    mov     eax,dword ptr [cptr]
    mov     edx,dword ptr [eax]
    mov     ecx,dword ptr [cptr]
    mov     eax,dword ptr [edx+4]
    call    eax
```

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Atunci când o clasă deriva din două clase virtuale, se creează două tabele `vfptr` (una după alta) care țin adresele funcțiilor virtuale din clasele de baza.

App.cpp

```
class A {
public:
    int a1;
    virtual int Suma() { return 1; }
    virtual int Diferenta() { return 2; }
};
class B {
public:
    int b1,b2;
    virtual int Inmultire() { return 3; }
    virtual int Impartire() { return 4; }
};
class C : public A, public B {
public:
    int x, y;
};
void main() {
    C c;
    C *cptr = &c;
    cptr->Impartire();
    cptr->Diferenta();
}
```

Offset	Camp
+ 0	A::vfptr
+ 4	A::a1
+ 8	B::vfptr
+ 12	B::b1
+ 16	B::b2
+ 20	C::x
+ 24	C::y

VTable clasa A

Adresa A::Suma

Adresa A::Diferenta

RTTI

VTable clasa B

Adresa B::Inmultire

Adresa B::Impartire

RTTI

Cum modelează compilatorul comportamentul funcțiilor virtuale

- ❖ Structura se păstrează și în cazul derivărilor ulterioare (de exemplu dacă facem o clasă D care o moștenește pe C).

App.cpp

```
class A {
public:
    int a1;
    virtual int Suma() { return 1; }
    virtual int Diferenta() { return 2; }
};
class B {
public:
    int b1,b2;
    virtual int Inmultire() { return 3; }
    virtual int Impartire() { return 4; }
};
class C : public A, public B {
public:
    int x, y;
};
class D : public C {
public:
    int d1;
};
```

Offset	Camp
+ 0	A::vfptr
+ 4	A::a1
+ 8	B::vfptr
+ 12	B::b1
+ 16	B::b2
+ 20	C::x
+ 24	C::y
+ 28	D::d1

VTable clasa A

Adresa A::Suma

Adresa A::Diferenta

RTTI

VTable clasa B

Adresa B::Inmultire

Adresa B::Impartire

RTTI

Covarianti

Covarianti

❖ Fie următorul cod.

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual A* clone() { return new B(); }
};
void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = b->clone();
}
```

❖ Codul nu compilează. Chiar dacă în realitate, funcția clone din clasa B returnează un obiect de tipul B, pentru că este definită ca o funcție care returnează un obiect de tipul A, compilatorul va spune că nu poate casta de la A* la un B*.

Covarianti

❖ Fie următorul cod.

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual A* clone() { return new B(); }
};
void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = b->clone();
}
```

❖ C++ oferă două soluții la aceasta problema:

Covarianti

❖ Fie următorul cod.

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual A* clone() { return new B(); }
};
void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = (B*) b->clone();
}
```

❖ C++ oferă două soluții la aceasta problema:

1. Cast explicit la tipul în care convertim.

Covarianti

❖ Fie următorul cod.

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual B* clone() { return new B(); }
};
void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = b->clone();
}
```

❖ C++ oferă două soluții la aceasta problema:

1. Cast explicit la tipul în care convertim.
2. Utilizarea covariantilor. Mai exact putem modifica tipul de **return** din funcția `clone` la `B*` din `A*`. În acest caz nu mai avem nevoie de cast-ul explicit.

Covarianti

❖ Fie următorul cod.

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual B* clone() { return new B(); }
};
void main()
{
    B *b = new B();
    B *ptrB;
    ptrB = b->clone();
    A *a = (A*)b;
    ptrB = (B*)a->clone();
}
```

❖ C++ oferă două soluții la aceasta problema:

1. Cast explicit la tipul în care convertim.
2. Utilizarea covariantilor. Mai exact putem modifica tipul de **return** din funcția **clone** la **B*** din **A***. În acest caz nu mai avem nevoie de cast-ul explicit.

Covarianții funcționează specific pe tipul pointerului. Apelul lui **clone** pe un pointer de tipul **A** chiar dacă va apela **B::clone** va necesita un cast la **B*** conform definiției lui **A::clone**.

Covarianti

❖ Fie următorul cod.

App.cpp

```
class A
{
public:
    int a1, a2;
    virtual A* clone() { return new A(); }
};
class B : public A
{
public:
    int b1, b2;
    virtual int* clone() { return new int(); }
};
void main()
{
    ...
}
```

error C2555: 'B::clone': overriding virtual function return type differs and is not covariant from 'A::clone'

❖ Codul alăturat nu compilează. Schimbarea tipului unei funcții virtuale este permisă doar dacă tipul este schimbat către un obiect derivate din tipul de return al funcției virtuale din clasa de baza.

Clase abstracte

Clase abstracte

- ▶ In C++ exista posibilitatea definirii unei așa numite funcții **virtuale pure** (adăugând **=0**) la sfârșitul definiției acelei funcții.
- ▶ Existenta unei funcții virtuale pure într-o clasa transforma acea clasa într-o clasa abstracta (clasa care nu poate fi instanțiala).
- ▶ O funcție virtuala pura obliga pe cel care deriva din clasa din care face parte sa asigure si o implementare pentru acea funcție daca dorește sa instantieze un obiect de acel tip.

App.cpp

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

- ❖ Codul alăturat nu va compila pentru ca clasa A are o funcție virtuala pura si nu se pot instanția obiecte pentru clase abstracte.

Clase abstracte

- In C++ exista posibilitatea definirii unei așa numite funcții **virtuale pure** (adăugând =0) la sfârșitul definiției acelei funcții.

App.cpp

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

- ❖ Codul alăturat va compila pentru ca clasa B implementeze funcția Set din clasa de baza.
- ❖ Pentru a se putea crea o instanță, e nevoie ca toate funcțiile virtuale pure să aibă o implementare !!!

Clase abstracte

- In C++ exista posibilitatea definirii unei așa numite funcții **virtuale pure** (adăugând =0) la sfârșitul definiției acelei funcții.

App.cpp

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

- ❖ Codul alăturat compilează. Clasele abstracte pot fi referite prin pointeri.

Clase abstracte

- ▶ Alte limbaje (gen Java sau C#) au un concept numit **interfața** (folosit pentru a asigura o echivalență moștenirii multiple precum și pentru polimorfism)
- ▶ Clasele abstracte din C++ sunt o generalizare a conceptului de **interfața** (la modul în care este definit în Java și C#). În aceste două limbaje o **interfața** este reprezentată doar de **funcții** care trebuie implementate în clasele derivate.
- ▶ În C++ o clasă abstractă are cel puțin o funcție virtuală pură, dar nu este limitată la această noțiune. O clasă abstractă poate avea și funcții cu **implementări**, poate avea și date membre.

Alinieri in memorie pentru clasele derivate

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 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 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**

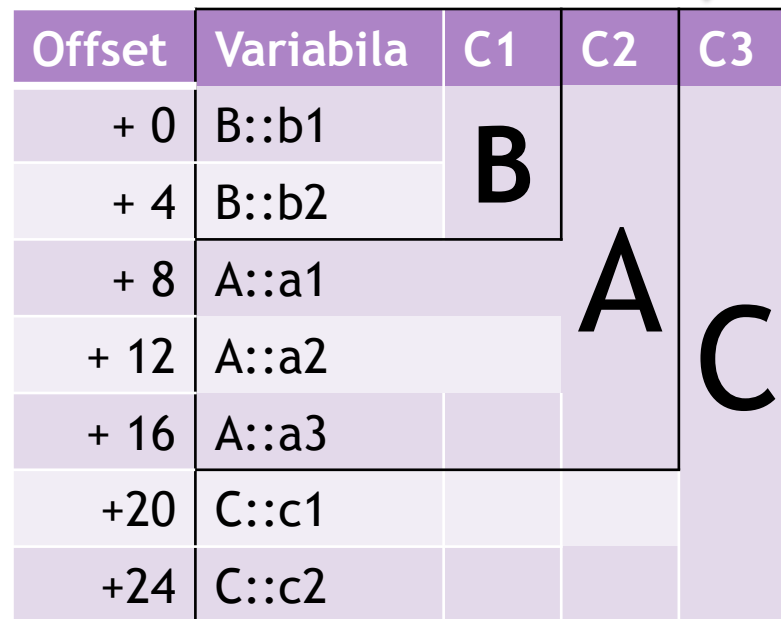
```
class B:
{
public:
    int b1,b2;
};
```

`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

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

- ❖ Moștenirea multipla poate produce situații ambigui. De exemplu in acest caz, clasa A are doua instanțe in cadrul clase C.

App.cpp

```
class A
{
public:
    int a1, a2, a3;
};
class B: public A
{
public:
    int b1, b2;
};
class C : public A, public B
{
public:
    int c1, c2;
};
void main()
{
}
```

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			

Alinieri in memorie pentru clasele derivate

- ❖ Moștenirea multipla poate produce situații ambigui. De exemplu in acest caz, clasa A are doua instanțe in cadrul clase C.

App.cpp

```
class A {  
public:  
    int a1, a2, a3;  
};  
class B: public A {  
public:  
    int b1, b2;  
};  
class C : public A, public B {  
public:  
    int c1, c2;  
};  
void main()  
{  
    C c;  
    c.a1 = 10;  
}
```

Compilatorul nu știe cum sa interpreteze apelul c.a1 (nu știe daca trebuie sa se refere la membrul a1 din cadrul derivării din clasa A sau la membrul a1 obținut din derivarea din clasa B (care e la rândul lui derivate din clasa A).

Codul nu compilează !!!

Alinieri in memorie pentru clasele derivate

- ❖ Moștenirea multipla poate produce situații ambigui. De exemplu in acest caz, clasa A are doua instanțe in cadrul clase C.

App.cpp

```
class A {  
public:  
    int a1, a2, a3;  
};  
class B: public A {  
public:  
    int b1, b2;  
};  
class C : public A, public B {  
public:  
    int c1, c2;  
};  
void main()  
{  
    C c;  
    c.A::a1 = 10;  
    c.B::A::a1 = 20;  
}
```

- ❖ Soluția este să descriem un membru prin calea lui întreaga (lista de clase din care este derivate). In exemplul alăturat, **c.A::a1** se refera la membrul **a1** datorat derivării directe din A, iar **c.B::A::a1** se refera la membrul **a1** datorat derivării directe a lui C din B, care la rândul lui este derivate din A.
- ❖ Dar daca dorim sa ca un obiect de tipul A sa fie instantiate o singura data ?
- ❖ Aceasta problema mai poarta numele si de “**problema carourilor**”

Alinieri in memorie pentru clasele derivate

- ❖ Moștenirea multipla poate produce situații ambigui. De exemplu in acest caz, clasa A are doua instanțe in cadrul clase C.

App.cpp

```
class A {
public:
    int a1, a2, a3;
};
class B: public virtual A {
public:
    int b1, b2;
};
class C : public virtual A, public B {
public:
    int c1, c2;
};
void main()
{
    C c;
    c.a1 = 10;
    c.a2 = 20;
}
```

- ❖ O altă soluție este utilizarea cuvântului cheie “**virtual**” atunci când se face derivarea dintr-o clasa. In cazul alăturat trebuie daca orice derivare din A se face folosind virtual, A va avea o singura instanța intr-un obiect C.
- ❖ Pentru ca acest cod sa funcționeze, trebuie ca atât C cat si B sa deriveze folosind **virtual** din A.

Alinieri in memorie pentru clasele derivate

- ❖ Exact ca si in cazul funcțiilor virtuale si in acest caz se creează un constructor virtual sau se modifica cele existente.

App.cpp	Disasm
<pre>class A { public: int a1, a2, a3; }; class B: public virtual A { public: int b1, b2; }; class C : public virtual A, public B { public: int c1, c2; }; void main() { C c; c.a1 = 10; c.b1 = 20; }</pre>	<pre>C c; push 1 lea ecx,[c] call C::C c.a1 = 10; mov eax,dword ptr [c] mov ecx,dword ptr [eax+4] mov dword ptr [c+ecx],10 c.b1 = 20; mov dword ptr [c+20],20</pre>

Alinieri in memorie pentru clasele derivate

- ❖ Codul generat in constructor primește un parametru (**True(1)** sau **False(0)**). Acest parametru ii spune constructorului daca trebuie sa seteze sau nu tabela cu indexi a clasei.

App.cpp

```
class A {  
public:  
    int a1, a2, a3;  
};  
class B: public virtual A {  
public:  
    int b1, b2;  
};  
class C : public virtual A, public B {  
public:  
    int c1, c2;  
};  
void main()  
{  
    C c;  
    c.a1 = 10;  
    c.b1 = 20;  
}
```

Disasm C::C

```
push    ebp  
mov     ebp,esp  
mov     dword ptr [this],ecx  
cmp     dword ptr [ebp+8],0  
je      DONT_SET_VAR_PTR  
mov     eax,dword ptr [this]  
mov     dword ptr [eax],addr_index  
DONT_SET_VAR_PTR:  
push    0  
mov     ecx,dword ptr [this]  
call    B::B  
mov     eax,dword ptr [this]  
mov     esp,ebp  
pop     ebp  
ret     4
```

Alinieri in memorie pentru clasele derivate

- ❖ In memorie după apelul constructorului, un obiect de tipul C va arata in felul următor:

Offset	Variabila	C1	C2
+ 0	Ptr Class C Variable Offssets Table		
+ 4	B::b1	B	C
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1	A	
+ 24	A::a2		
+ 28	A::a3		

Offset	Offset relative la C
+ 0	0
+ 4	Virtual A 20

Alinieri in memorie pentru clasele derivate

- ❖ Asignarea unei valori care aparține unei clase din care s-a declarat folosind cuvântul cheie **virtual** se face în 3 pași.

App.cpp

```
class A { ... }  
class B: public virtual A { ... }  
class C : public virtual A, public B { ... }  
void main()  
{  
    C c:  
    c.a1 = 10;  
}
```

Disasm

c.a1 = 10;

```
mov     eax,dword ptr [c]  
mov     ecx,dword ptr [eax+4]  
mov     dword ptr [c+ecx],10
```

Alinieri in memorie pentru clasele derivate

- ❖ In primul pas, EAX primește primul dword care se găsește la începutul lui c (adică pointerul către tabela cu indexi pentru variabilele virtuale din C).

App.cpp

```
class A { ... }  
class B: public virtual A { ... }  
class C : public virtual A, public B { ... }
```

Offset	Variabila	C1	C2
+ 0	Ptr Class C Variable Offsets Table		
+ 4	B::b1	B	C
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1	A	
+ 24	A::a2		
+ 28	A::a3		

Disasm

```
c.a1 = 10;
```

```
mov    eax,dword ptr [c]  
mov    ecx,dword ptr [eax+4]  
mov    dword ptr [c+ecx],10
```

Alinieri in memorie pentru clasele derivate

- ❖ In al doilea pas, ECX primește valoarea de la a doua poziție (+4) din tabela cu indexi la care punctează EAX (adică valoarea 20 → offset-ul lui A::a1 din C)

App.cpp

```
class A { ... }  
class B: public virtual A { ... }  
class C : public virtual A, public B { ... }
```

Offset	Variabila	C1	C2
+ 0	Ptr Class C Variable Offssets Table		
+ 4	B::b1	B	C
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1	A	
+ 24	A::a2		
+ 28	A::a3		

Disasm

```
c.a1 = 10;
```

```
mov     eax,dword ptr [c]  
mov     ecx,dword ptr [eax+4]  
mov     dword ptr [c+ecx],10
```

Offset	Offset relativ la C
+ 0	0
+ 4	Virtual A 20

Alinieri in memorie pentru clasele derivate

- ❖ In ultimul pas, la offset-ul date de ECX (adică 20) fata de adresa lui “c” se stochează valoarea 10

App.cpp

```
class A { ... }  
class B: public virtual A { ... }  
class C : public virtual A, public B { ... }
```

Offset	Variabila	C1	C2
+ 0	Ptr Class C Variable Offssts Table		
+ 4	B::b1	B	C
+ 8	B::b2		
+ 12	C::c1		
+ 16	C::c2		
+ 20	A::a1	A	
+ 24	A::a2		
+ 28	A::a3		

Disasm

```
c.a1 = 10;
```

```
mov     eax,dword ptr [c]  
mov     ecx,dword ptr [eax+4]  
mov     dword ptr [c+ecx],10
```

Offset	Offset relative la C
+ 0	0
+ 4	Virtual A 20

Alinieri in memorie pentru clasele derivate

- ❖ Membri din derivările virtuale sunt puși ultimii in structura. In cazul de mai jos, cum doar A este derivate virtual, el va fi adăugat ultimul intr-un obiect C

App.cpp

```
class A
{ ... }
class B: public virtual A
{ ... }
class C : public virtual A,
          public B
{ ... }
```

Offset Camp

+ 0	ptr class C virtual members offsets
+ 4	C::B::b1
+ 8	C::B::b2
+ 12	C::c1
+ 16	C::c2
+ 20	A::a1 (virtual A from C)
+ 24	A::a2 (virtual A from C)
+ 28	A::a3 (virtual A from C)

Offset	Offset relative la C
+ 0	0
+ 4	Virtual A 20

Alinieri in memorie pentru clasele derivate

- ❖ Daca in C derivam si pe B tot cu virtual, B va fi adăugat după A (A este primul derivat virtual)

App.cpp

```
class A
{ ... }
class B: public virtual A
{ ... }
class C : public virtual A,
          public virtual B
{ ... }
```

Offset	Offset relative la C	
+ 0		0
+ 4	Virtual A	12
+ 8	Virtual B	24

Offset Camp

+ 0	ptr class C virtual members offsets
+ 4	C::c1
+ 8	C::c2
+ 12	A::a1 (virtual A from C)
+ 16	A::a2 (virtual A from C)
+ 20	A::a3 (virtual A from C)
+ 24	ptr class B virtual members offsets
+ 28	B::b1 (virtual B from C)
+ 32	B::b2 (virtual B from C)

Alinieri in memorie pentru clasele derivate

❖ In cazul tabelii de indexi din B, offset-ul “-12” se refera la poziția lui A (membru virtual in B) relative la poziția lui B (deci $24 - 12 = 12$ - poziție in C)

App.cpp

```
class A
{ ... }
class B: public virtual A
{ ... }
class C : public virtual A,
          public virtual B
{ ... }
```

Offset	Camp
+ 0	ptr class C virtual members offsets
+ 4	C::c1
+ 8	C::c2
+ 12	A::a1 (virtual A from C)
+ 16	A::a2 (virtual A from C)
+ 20	A::a3 (virtual A from C)
+ 24	ptr class B virtual members offsets
+ 28	B::b1 (virtual B from C)
+ 32	B::b2 (virtual B from C)

Offset	Offset relative la B	
+ 0	0	
+ 4	Virtual A	-12

Alinieri in memorie pentru clasele derivate

- ❖ Daca doar din B se face derivare virtuala, atunci câmpurile se organizează in felul următor:

```
App.cpp
class A
{ ... }
class B: public A
{ ... }
class C : public A,
          public virtual B
{ ... }
```

Offset	Camp
+ 0	A::a1
+ 4	A::a1
+ 8	A::a3
+ 12	ptr class C virtual members offsets
+ 16	C::c1
+ 20	C::c2
+ 24	B::A::a1
+ 28	B::A::a2
+ 32	B::A::a3
+ 36	B::b1
+ 40	B::b2

Offset	Offset relative la C	
+ 0	-12	
+ 4	Virtual B	12

Alinieri in memorie pentru clasele derivate

- ❖ Daca doar din B se face derivare virtuala, atunci câmpurile se organizează in felul următor:

App.cpp

```
class A
{ ... }
class B: public A
{ ... }
class C : public A,
          public virtual B
{ ... }
```

Offset Camp

+ 0 A::a1

+ 4 A::a1

+ 8 A::a3

+ 12 ptr class C virtual members offsets

+ 16 C::c1

Primul index (+0) reprezintă la ce offset se afla începutul unui obiect de tipul C fata de offsetul in care se găsește pointerul către tabela cu offseturi virtuale.

+ 32 B::A::a3

+ 36 B::b1

+ 40 B::b2

Offset	Offset relative la C	
+ 0		-12
+ 4	Virtual B	12

Alinieri in memorie pentru clasele derivate

- ❖ Daca doar din B se face derivare virtuala, atunci câmpurile se organizează in felul următor:

```
App.cpp
class A
{ ... }
class B: public A
{ ... }
class C : public A,
          public virtual B
{ ... }
```

Offset	Camp
+ 0	A::a1
+ 4	A::a1
+ 8	A::a3
+ 12	ptr class C virtual members offsets
+ 16	C::c1
+ 20	C::c2

Offset	Offset relative la C	
+ 0		-12
+ 4	Virtual B	12

Al doilea index (+4) reprezintă la ce offset se afla începutul unui obiectului virtual B fata de offsetul in care se găsește pointerul către tabela cu offseturi virtuale.

+ 36	B::b1
+ 40	B::b2

Cast-uri intre clasele derivate

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
```