



UNIVERSITATEA BABEŞ-BOLYAI
Facultatea de Matematică și Informatică



Programare orientată obiect

Curs 07

Laura Dioşan

Cuprins

- Moștenire
 - Elemente generice
 - folosind void*
 - folosind clase
 - Constructori și destructori
 - Operatorul de atribuire
 - Generalizare, specializare
 - Moștenire multiplă – *diamond problem*

Relația de derivare (moștenire)

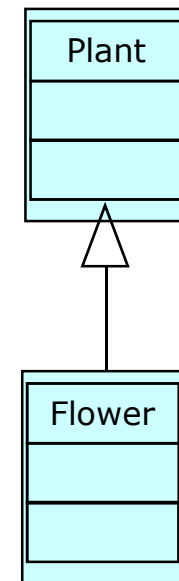
- În cazul unui limbaj ne-OO
 - Cu ajutorul *void**
- În cazul unui limbaj OO
 - Cu ajutorul claselor derivate

Relația de derivare (moștenire)

- Permite crearea unei noi clase (numită *clasă derivată*, *clasă moștenită*, *sub-clasă* sau *clasă fiu*) dintr-o clasă sau din mai multe clase (numită/e *clasă de bază*, *super clasă* sau *clasă părinte*)
- Între 2 clase A (clasa părinte) și B (clasa fiu) există o relație de derivare (moștenire) dacă:
 - B are toate datele și metodele lui A
 - B poate redefini/suprascrie metode din A
 - B poate adăuga date și metode noi celor moștenite din A

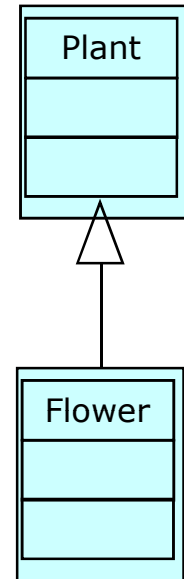
Relația de derivare (moștenire) - UML

- Clasa fiu moștenește caracteristicile clasei părinte
 - Clasa nouă este un ca un fel de clasă veche
 - *A este o B*
 - *A este ca B*
 - *A este un fel de B*
- Floarea este o Plantă



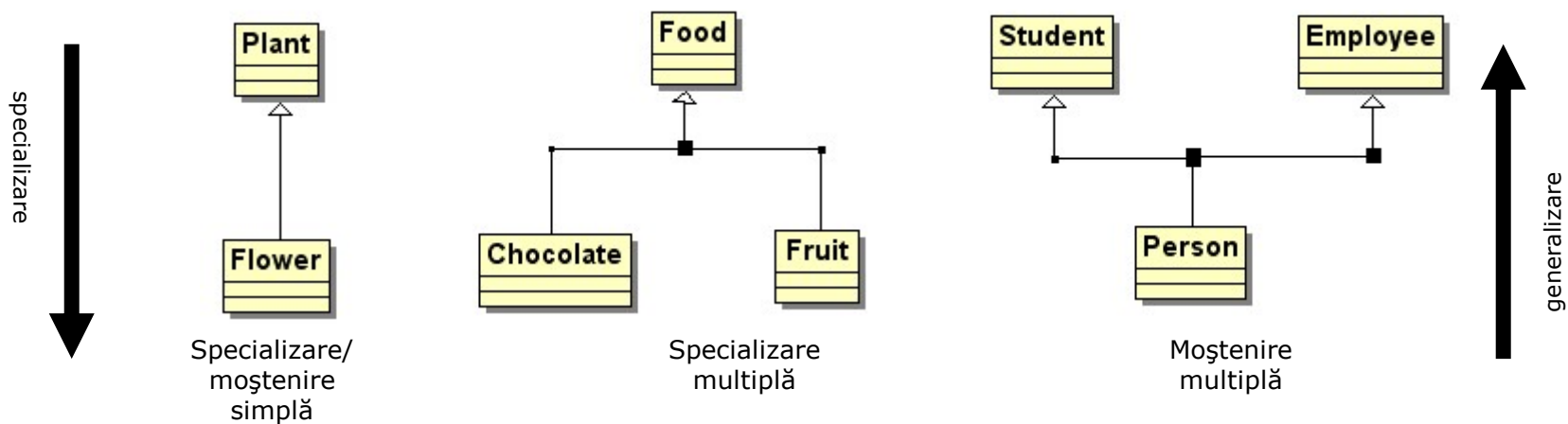
Relația de derivare – concepte noi

- Clasă de bază (CB), super clasă sau clasă părinte
 - *Plantă*
- Clasă derivată (CD), clasă moștenită, sub clasă sau clasă fiu
 - *Floare*
- Metode redefinite/suprascrise
 - Metode care sunt definite atât în clasa de bază, cât și în clasa fiu, iar clasa fiu
 - *schimbă* comportamentul
 - *nu schimbă* signatura (numele și lista parametrilor)
 - unei metode existente în clasa părinte
 - != funcții supraîncărcate
- Date membre și metode noi (în clasa derivată)



Tipologie

- Relația de derivare → ierarhii de clase
 - Nr. de părinți
 - Derivare (moștenire) simplă
 - O singură clasă de bază → ierarhia este un arbore
 - Derivare (moștenire) multiplă
 - Mai multe clase de bază → ierarhia este un graf
 - posibile confuzii
 - Semantica
 - Specializare (de la general la particular)
 - Generalizare (de la particular la general)



Declarare

```
class nume_clasă_derivată : listă_clasă_bază{  
    @date și metode – noi și redefinite/suprascrise  
}  
unde:  
Listă_clasă_bază =  
    [<protecție>] clasă_bază1, [<protecție>] clasă_bază2, ..., [<protecție>] clasă_bazăn
```

- protecție:
 - *private* (implicit)
 - *protected*
 - *public*

- a se consulta exemplul din directorul 06/inheritanceType
 - *Base.h, Derived.h, test.cpp*

Protecția derivării (moștenirii)

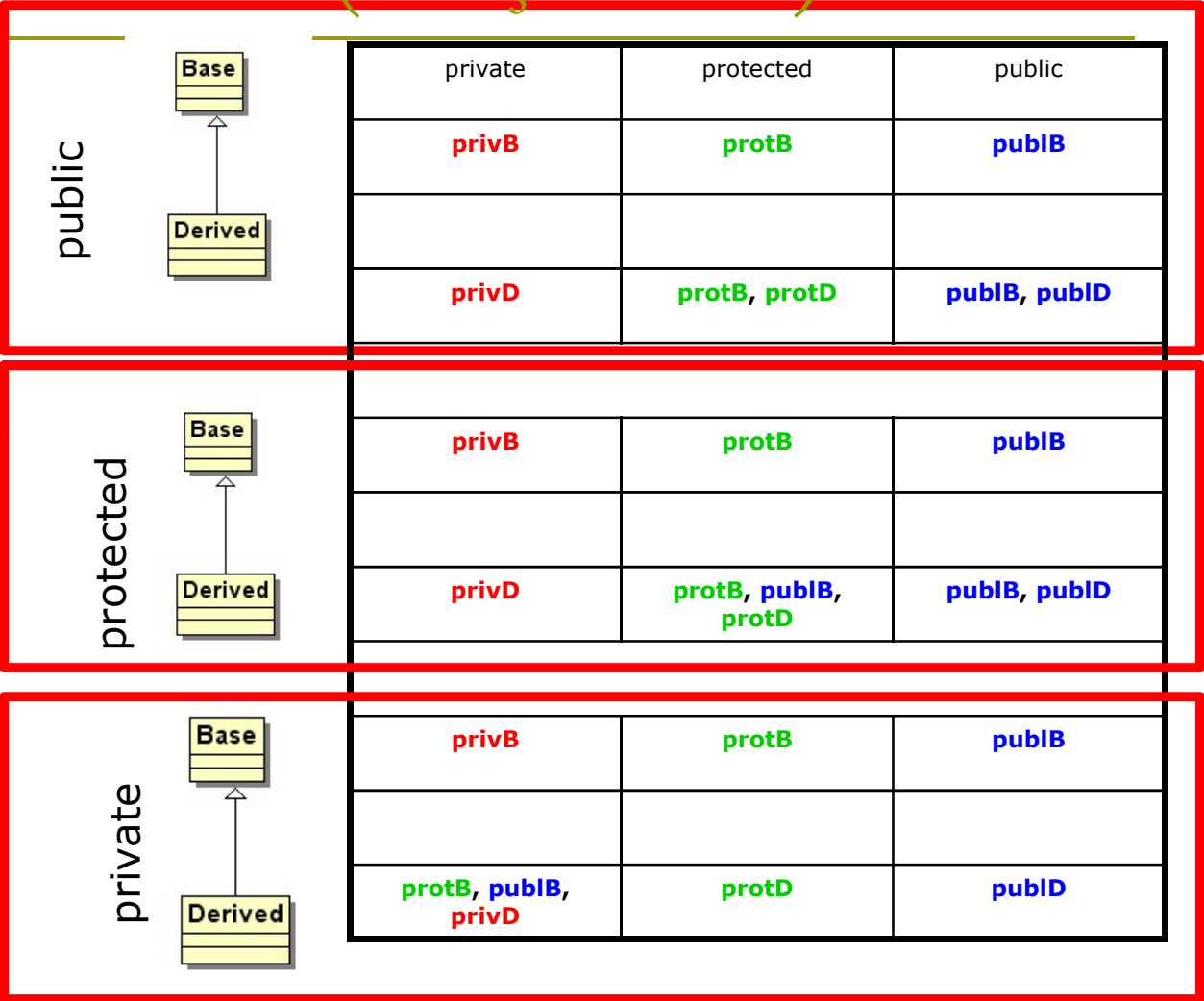
```

class Bază{
private:
    tip privB;
protected:
    tip protB;
public:
    tip publB;
}
    
```

```

class Derivată{
private:
    tip privD;
protected:
    tip protD;
public:
    tip publD;
}
    
```

Bază →	-	#	+
↓Derivată			
-	X	-	-
#	X	#	#
+	x	#	+



Un atribut protected din Baza devine privat in Derivata

Un atribut privat din Baza nu este vizibil in Derivata

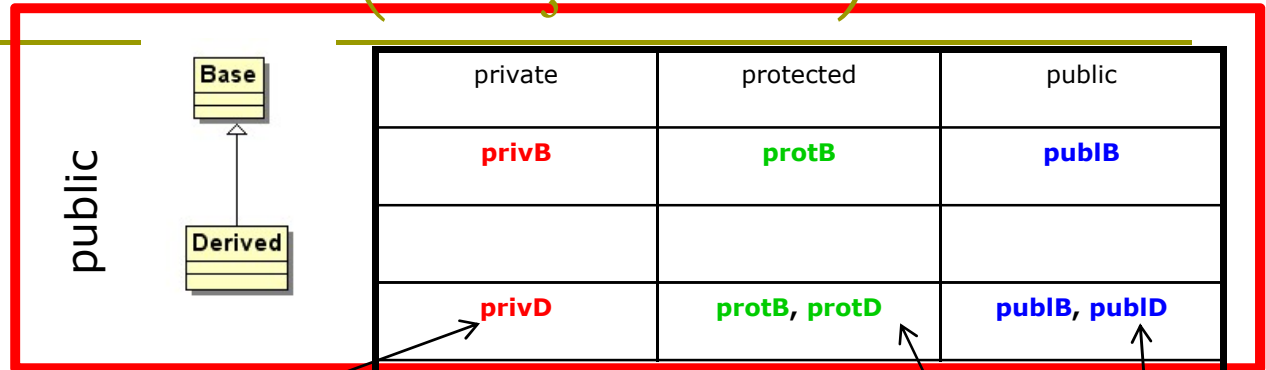
Protecția derivării (moștenirii)

```

class Bază{
private:
    tip privB;
protected:
    tip protB;
public:
    tip pubB;
}
    
```

```

class Derivată{
private:
    tip privD;
protected:
    tip protD;
public:
    tip pubD;
}
    
```



In Derivata exista acces doar la atributul privat propriu (privD); privB nu se poate accesa (fiind atribut privat in clasa Baza)

In Derivata se pot accesa ambele atribute protected: protD (conform definirii clasei Derivata) si protB (conform mostenirii publice din clasa Baza)

In Derivata se pot accesa ambele atribute public: pubD (conform definirii clasei Derivata) si pubB (conform mostenirii publice din clasa Baza)

Bază →	-	#	+
↓Derivată			
-	X	-	-
#	X	#	#
+	x	#	+

Un atribut protected din Baza devine privat in Derivata

Un atribut privat din Baza nu este vizibil in Derivata

Protecția derivării (moștenirii)

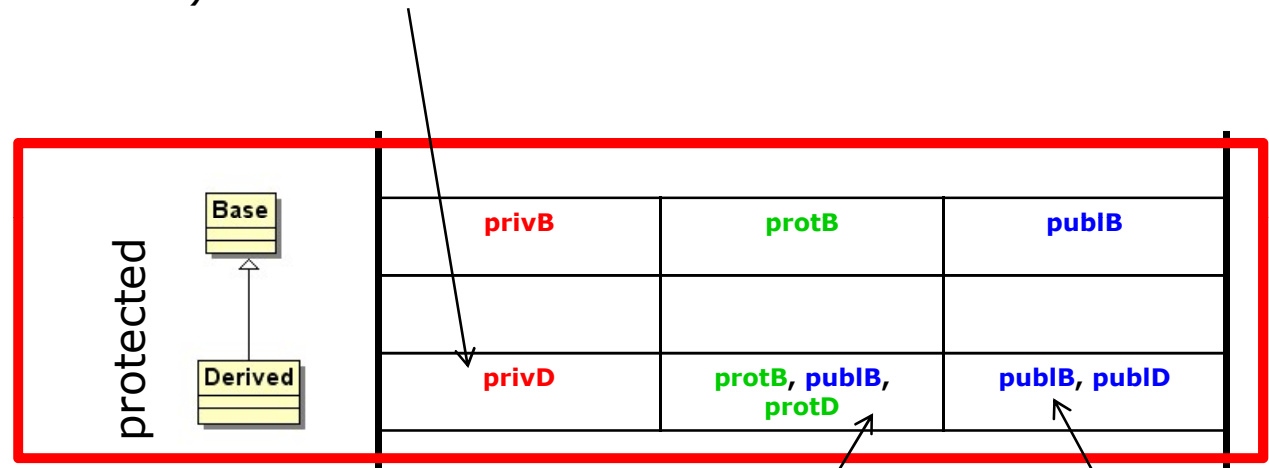
```

class Bază{
private:
    tip privB;
protected:
    tip protB;
public:
    tip publB;
}
    
```

```

class Derivată{
private:
    tip privD;
protected:
    tip protD;
public:
    tip publD;
}
    
```

In Derivata exista acces doar la atributul privat propriu (privD); privB nu se poate accesa (fiind atribut privat in clasa Baza)



Bază →	-	#	+
↓Derivată			
-	X	-	-
#	X	#	#
+	x	#	+

In Derivata se pot accesa ambele attribute protected: protD (conform definirii clasei Derivata) si protB (conform mostenirii publice din clasa Baza), iar accesul la atributul public publB mostenit din Baza este de tip protected.

In Derivata se pot accesa ambele attribute public: publD (conform definirii clasei Derivata) si plubB (conform mostenirii publice din clasa Baza)

Un atribut protected din Baza devine privat in Derivata
 Un atribut privat din Baza nu este vizibil in Derivata

Protecția derivării (mostenirii) /

```

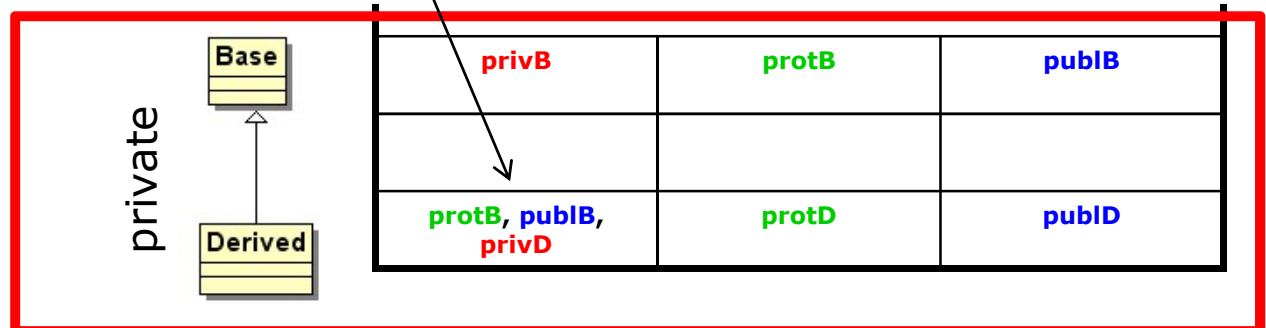
class Bază{
  private:
    tip privB;
  protected:
    tip protB;
  public:
    tip pubB;
}
    
```

```

class Derivată{
  private:
    tip privD;
  protected:
    tip protD;
  public:
    tip pubD;
}
    
```

Bază →	-	#	+
↓Derivată			
-	X	-	-
#	X	#	#
+	x	#	+

In Derivata accesul la attributele mostenite din Baza (protB si pubB) si la atributul propriu privD este privat



Un atribut protected din Baza devine privat in Derivata

Un atribut privat din Baza nu este vizibil in Derivata

Relația de derivare – exemplu simplu

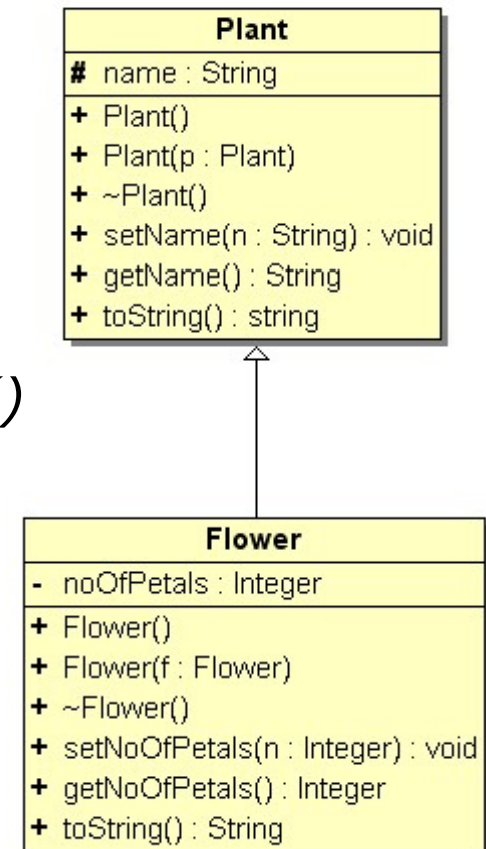
□ a se consulta exemplul din directorul 07/simple

■ *Plant.h, Plant.cpp*

■ *Flower.h, Flower.cpp*

■ *test.cpp*

□ subalgoritmul *constructorDestructor()*



Elemente moștenite în clasa derivată (CD)

- date
 - Datele *public* și *protected* din CB

- metode (publice)
 - Metodele ordinare
 - Metode definite în CB și care nu sunt definite în CD
 - Metodele redefinite
 - Metode definite în CB și care sunt definite și în CD, având același antet, dar comportament diferit
 - Metodele suprascrise
 - Metode definite în CB și care sunt definite și în CD, având antete diferite
 - Operatorii supraîncărcați (cu excepția operatorului de atribuire)
 - Elementele prietene (*friend*)

Elemente ne-moștenite

□ date

- datele private din CB

□ metode

- constructorii
- destructorul
- operatorul de atribuire supraîncărcat (*operator=*)

Elemente ne-moștenite – constructor

- Ordinea de execuție a constructorilor
 - mai întâi, constructorul CB,
 - creează cadrul general,
 - apoi, constructorul CD,
 - creează și adaugă informație specifică
- Când se crează o instanță a CD:
 - trebuie apelați (explicit, în lista de inițializare a constructorului) constructorii tuturor CB (dacă sunt mai multe CB) respectând ordinea de declarare,
 - iar apoi, sunt inițializate noile date ale CD
- Dacă o CB_i nu are cel puțin un constructor, atunci respectivul apel va lipsi din listă
- Constructorul CD trebuie să apeleze constructorii tuturor CB
- CD trebuie să aibă cel puțin un constructor dacă:
 - una dintre CB nu are constructor implicit sau
 - una dintre CB nu are constructor general cu argumente implicite

```
class CD : public CB1, public CB2, ..., public CBn{  
    ...  
    public:  
    CD() : CB1(), CB2(), ..., CBn() {  
        //inițializarea noilor date ale CD  
    }  
}
```


Elemente ne-moștenite – constructor de copiere

- Dacă CD nu are constructor de copiere, compilatorul va apela constructorii de copiere ai tuturor CB și va face o copie bit cu bit a noilor date ale CD
- Dacă CD are constructor de copiere, el trebuie să apeleze (în mod explicit, în lista de inițializare a sa) constructorul de copiere al tuturor CB

```
class CD : public CB1, public CB2, ..., public CBn{  
    ...  
    public:  
    CD(const CD &o) : CB1(o), CB2(o), ..., CBn(o) {  
        //copiere a noilor date din o  
    }  
}
```

Elemente ne-moștenite – destructor

- Ordinea de execuție a destructorilor
 - mai întâi, destructorul CD,
 - distruge datele specifice,
 - apoi destructorul CB,
 - distruge cadrul general

- Când se distruge o instanță a clasei derivate
 - se apelează destructorul CD,
 - iar apoi se apelează automat destructorii tuturor CB în ordinea inversă declarării

```
class CD : public CB1, public CB2, ..., public CBn{  
    ...  
    public:  
    ~CD() {  
        //distrugerea noilor date ale CD  
    }  
}
```

Elemente ne-moștenite – *operator=*

- similar constructorului de copiere
- Dacă CD nu are *operator=*, compilatorul va apela *operator=* al CB și va face o copie bit cu bit a noilor date ale CD
- Dacă CD are *operator=*, el trebuie să apeleze (în mod explicit, în corpul metodei) *operator=* al tuturor CB și apoi să copieze noile date ale CD

```
class CD : public CB1, public CB2, ..., public CBn{  
    ...  
    public:  
    CD& operator=(const CD &o) {  
        ...  
        CB1::operator=(o);  
        CB2::operator=(o);  
        ...  
        CBn::operator=(o);  
        //copierea noilor date ale lui o  
    }  
}
```

Exemplu

- A se consulta directorul 07/dataAndMethods
 - *Base.h*
 - *Base2.h*
 - *Derived.h*
 - *test.cpp*

Relația de substituție (*subtyping*)

□ Principiul substituției al lui Liskov (LSP)

- Substituție comportamentală (puternică)
- Fie $q(x)$ o proprietate demonstrabilă pentru obiecte x de tipul T . Atunci $q(v)$ este adevărată pentru obiecte z de tipul S , unde S este un sub-tip al lui T
- Substiția obiectelor mutabile

□ Relația de derivare → ierarhii de clase

□ 3 posibilități:

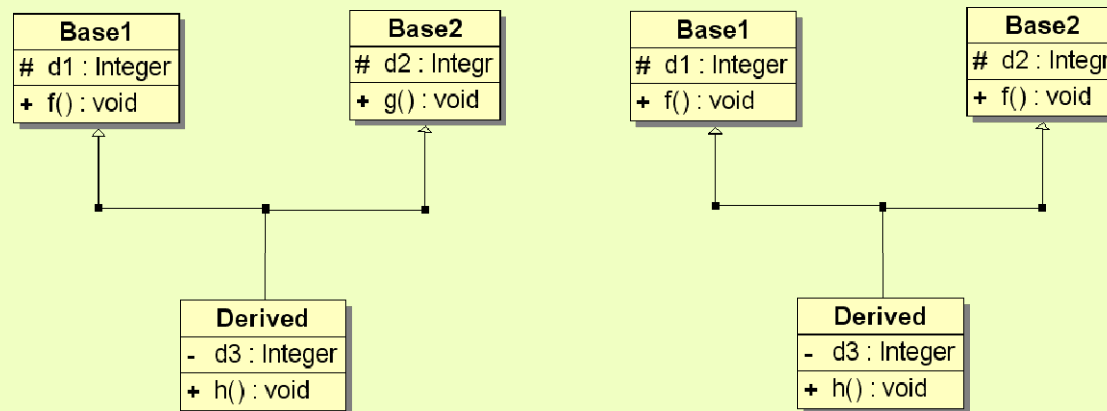
- obiectul de tip CD este convertit automat într-un obiect de tip CB
 - $CD \rightarrow CB, CB \nrightarrow CD$
- un pointer/o referință la CD poate înlocui un pointer/o referință la CB
 - $*CD \rightarrow *CB, *CB \nrightarrow *CD$
- un pointer la o metodă din CB poate înlocui un pointer la o metodă din CD
 - $*metodăCB \rightarrow *metodăCD, *metodăCD \nrightarrow *metodăCB$

Exemplu

- A se consulta exemplul din directorul 07/simple
 - *Plant.h, Plant.cpp*
 - *Flower.h, Flower.cpp*
 - *test.cpp*
 - Subalgoritmul *assignmentsBaseDerived()*
 - Subalgoritmul *references()*
 - Subalgoritmul *pointers()*

Derivare multiplă

- Mai multe clase de bază



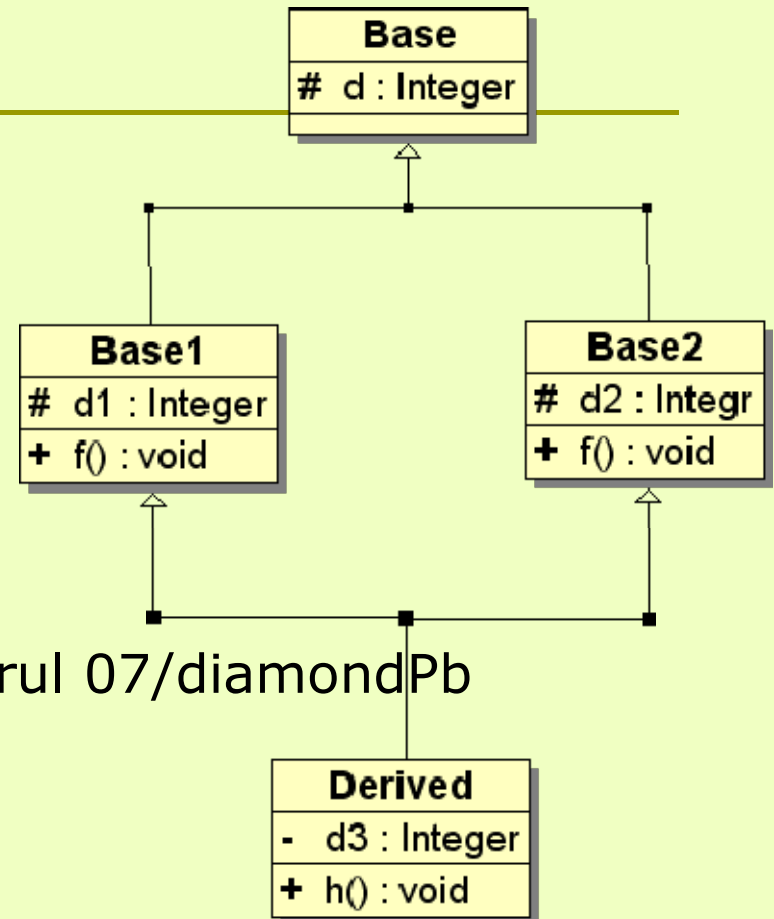
- A se consulta exemplul din directorul 07/multipleInheritance
 - *Base1.h*,
 - *Base2.h*,
 - *Derived.h*,
 - *test.cpp*

Derivare multiplă

□ *Diamond problem*

□ A se consulta exemplul din directorul 07/diamondPb

- *Base.h,*
- *Base1.h,*
- *Base2.h,*
- *Derived.h,*
- *test.cpp*



Temă

- Implementați următoarele clase (prin folosirea relației de derivare):
 - Animal
 - nume,
 - greutate
 - Câine
 - nume,
 - greutate,
 - tip (Beagle, Coker, ...),
 - culoareOchi
 - Pește
 - nume,
 - Greutate,
 - tip (Calcan, Crap, Somn, Pastrav),
 - lungime
- și folosiți-le.

Cursul următor

- Clase
 - Clase abstracte
 - Polimorfism
 - *Early and late binding*
 - Mecanismul virtual
 - Metode pure
 - UML
 - Avantajele polimorfismului