UNIVERSITATEA BABEŞ-BOLYAI
Facultatea de Matematică şi Informatică

# Programare orientată obiect

Curs 07

Laura Dioşan

# Cuprins

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

# Relaţia de derivare (moştenire)

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

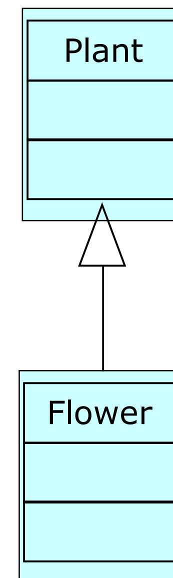
# Relaţia de derivare (moştenire)

- Permite crearea unei noi clase (numită *clasă derivată*, *clasă moştenită*, *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) şi B (clasa fiu) există o relaţie de derivare (moştenire) dacă:
  - B are toate datele şi metodele lui A
  - B poate redefini/suprascrie metode din A
  - B poate adăuga date şi metode noi celor moştenite din A

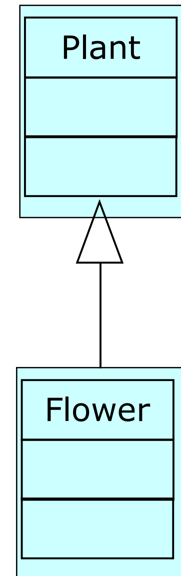
# Relaţia de derivare (moştenire) - UML

- Clasa fiu moşteneşte 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

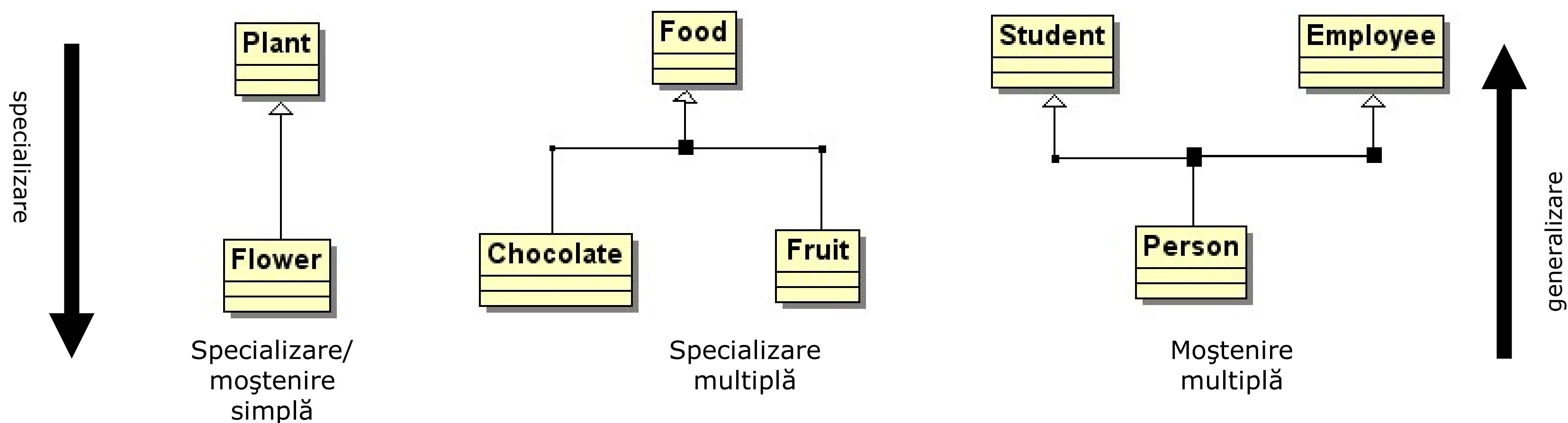
# Relaţia de derivare – concepte noi




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

# Tipologie

- Relaţia de derivare ➔ ierarhii de clase
  - Nr. de părinţi
    - Derivare (moştenire) simplă
      - O singură clasă de bază ➔ ierarhia este un arbore
    - Derivare (moştenire) 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_clase_bază{
        @date şi metode – noi şi redefinite/suprascrise
}
unde:
Listă_clase_bază =
       [<protecţie>] clasă_bază1, [<protecţie>] clasă_bază2, …, [<protecţie>] clasă_bazăn
```

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

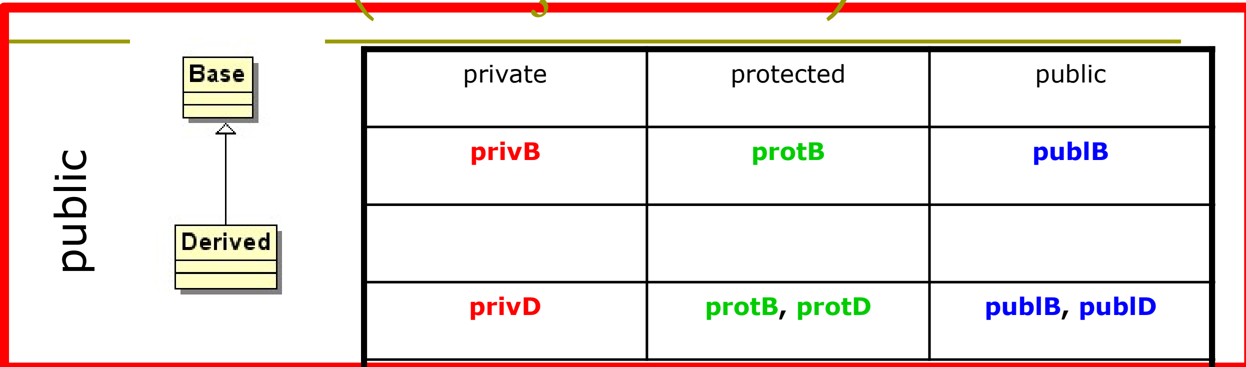
- a se consulta exemplul din directorul 06/inheritanceType
  - *Base.h, Derived.h, test.cpp*

# Protecția derivării (moștenirii)

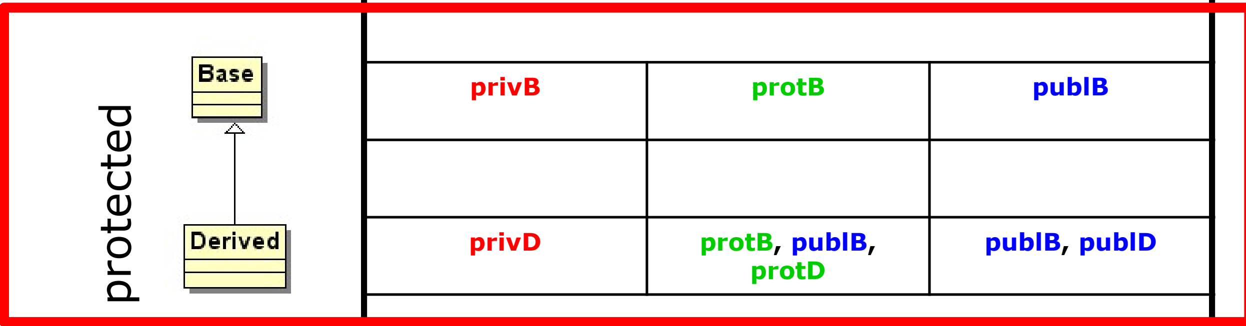
```
class Bază{
 private:
     tip privB;
protected:
     tip protB;
public:
     tip publB;
}
```

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

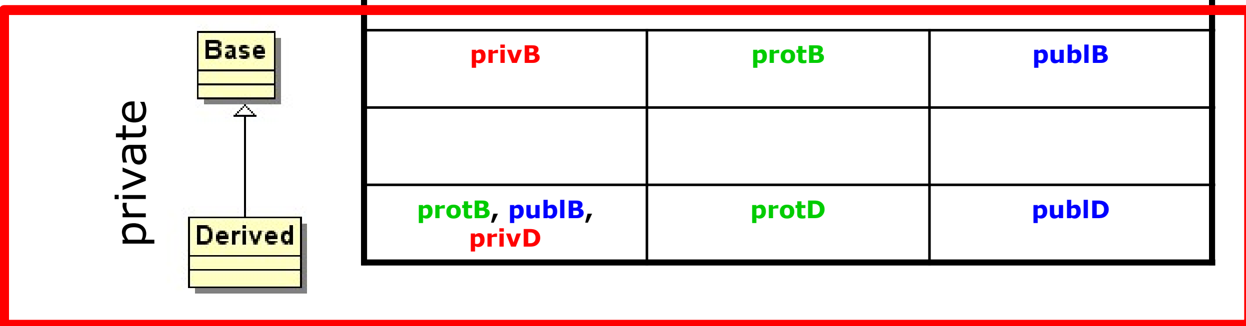
| Bază → ↓Derivată | - | # | + |
|---|---|---|---|
| - | X | - | - |
| # | X | # | # |
| + | x | # | + |




| private | protected | public |
|---|---|---|
| privB | protB | publB |
| | | |
| privD | protB, protD | publB, publD |




| private | protected | public |
|---|---|---|
| privB | protB | publB |
| | | |
| privD | protB, publB, protD | publB, publD |




| private | protected | public |
|---|---|---|
| privB | protB | publB |
| | | |
| protB, publB, privD | protD | publD |

Un atribut protected din Baza devine privat in Derivata

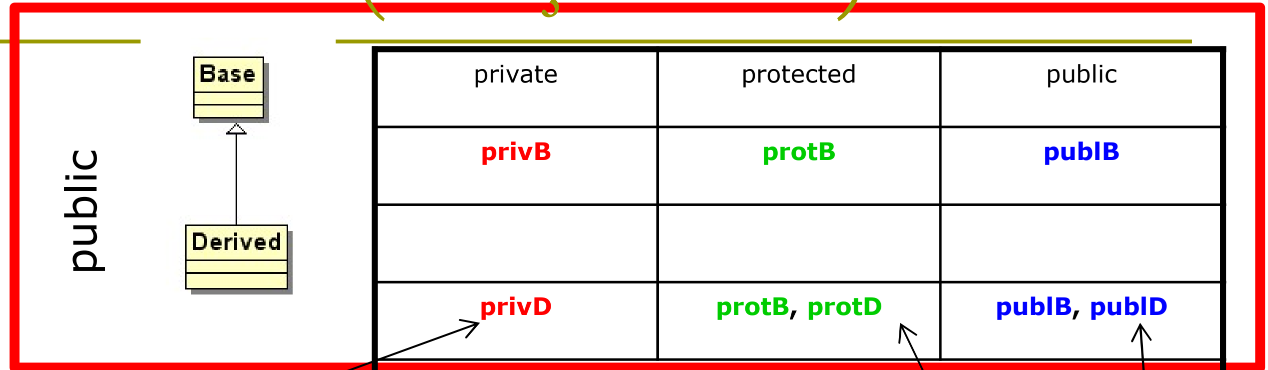
Un atribut privat din Baza nu este vizibil in Derivata

# Protecţia derivării (moştenirii)

```
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 | # | + |




| private | protected | public |
|---|---|---|
| privB | protB | publB |
| | | |
| privD | protB, protD | publB, publD |

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: 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

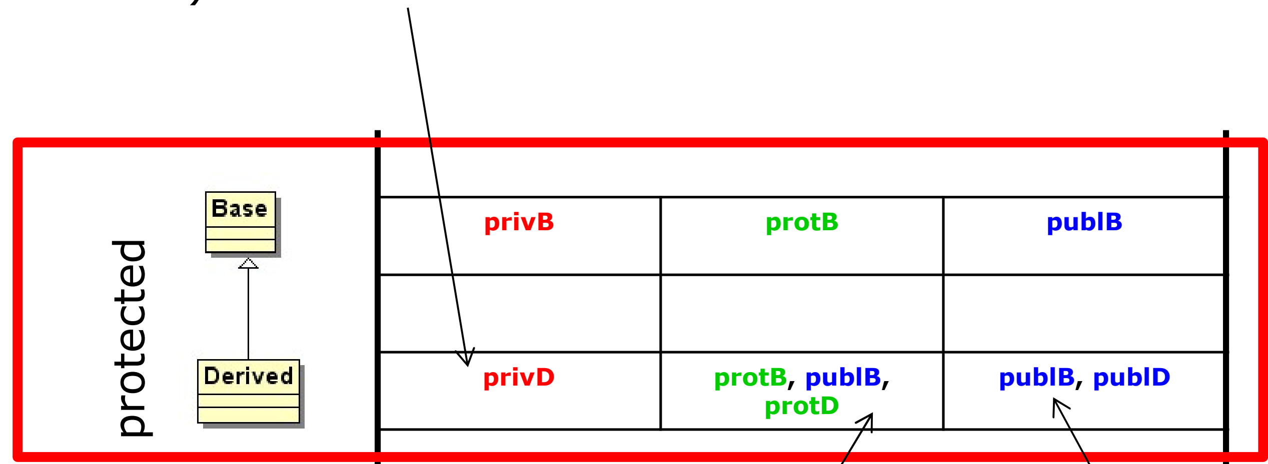
# Protecţia derivării (moştenirii)

```
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 | # | + |

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), iar accesul la atributul public publB mostenit din Baza este de tip protected.

In Derivata se pot accesa ambele atribute 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

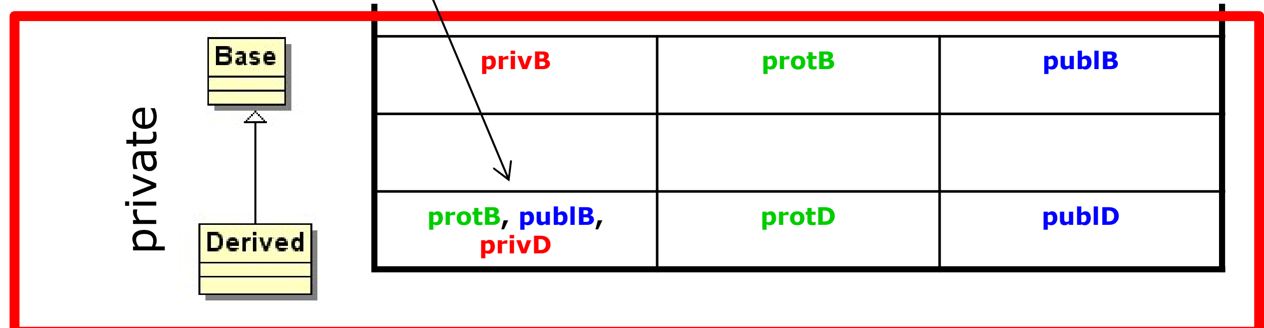
# Protecția derivării (mostenirii) /

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

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

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

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

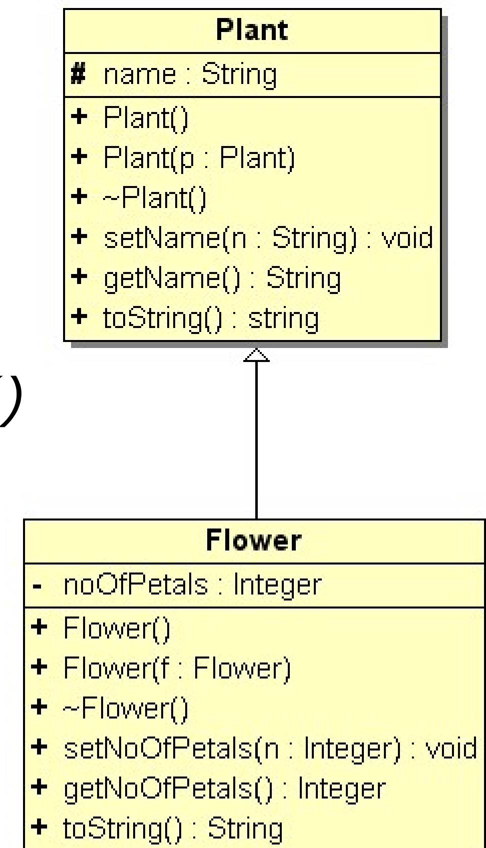



Un atribut protected din Baza devine privat in Derivata

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 moştenite în clasa derivată (CD)

- date
  - Datele *public* şi *protected* din CB
- metode (publice)
  - Metodele ordinare
    - Metode definite în CB şi care nu sunt definite în CD
  - Metodele redefinite
    - Metode definite în CB şi care sunt definite şi în CD, având acelaşi antet, dar comportament diferit
  - Metodele suprascrise
    - Metode definite în CB şi care sunt definite şi in CD, având antete diferite
  - Operatorii supraîncărcaţi (cu excepţia operatorului de atribuire)
  - Elementele prietene (*friend*)

# Elemente ne-moştenite

- date
  - datele private din CB
- metode
  - constructorii
  - destructorul
  - operatorul de atribuire supraîncărcat (*operator*=)

# Elemente ne-moştenite – constructor

- Ordinea de execuţie a constructorilor
  - mai întâi, constructorul CB,
    - creează cadrul general,
  - apoi, constructorul CD,
    - creează şi adaugă informaţie specifică

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

- Când se crează o instanţă a CD:
  - trebuie apelaţi (explicit, în lista de iniţializare a constructorului) constructorii tuturor CB (dacă sunt mai multe CB) respectând ordinea de declarare,
  - iar apoi, sunt iniţializate noile date ale CD
- Dacă o $CB_i$ nu are cel puţin un constructor, atunci respectivul apel va lipsi din listă
- Constructorul CD trebuie să apeleze constructorii tuturor CB
- CD trebuie să aibă cel puţin un constructor dacă:
  - una dintre CB nu are constructor implicit sau
  - una dintre CB nu are constructor general cu argumente implicite

# Elemente ne-moştenite – constructor de copiere

- Dacă CD nu are constructor de copiere, compilatorul va apela constructorii de copiere ai tututror CB şi 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 iniţializare 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-moştenite – destructor

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

- Când se distruge o instanţă 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-moştenite – *operator=*

- similar constructorului de copiere

- Dacă CD nu are *operator=,* compilatorul va apela *operator=* al CB şi 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 şi 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*

# Relaţia de substituţie (*subtyping*)

- **Principiul substituţiei al lui Liskov** (**LSP**)
  - Substituţie 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*
  - Substiţia obiectelor mutabile

- Relaţia de derivare ➔ ierarhii de clase

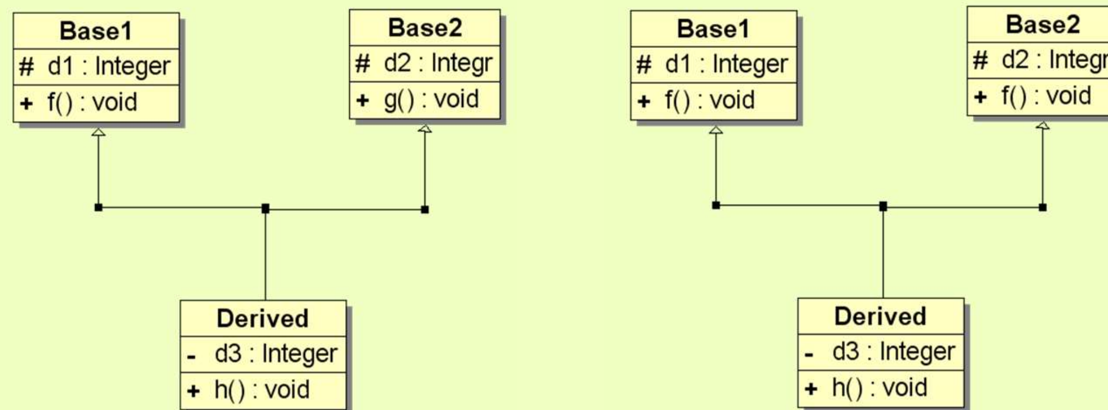
- 3 posibilităţi:
  - obiectul de tip CD este convertit automat într-un obiect de tip CB
    - CD ➔ CB, CB **!**➔ CD
  - un pointer/o referinţă la CD poate înlocui un pointer/o referinţă la CB
    - *CD ➔ *CB, *CB **!**➔ *CD
  - un pointer la o metodă din CB poate înlocui un pointer la o metodă din CD
    - *metodăCB ➔ *metodăCD, *metodăCD **!**➔ *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()*

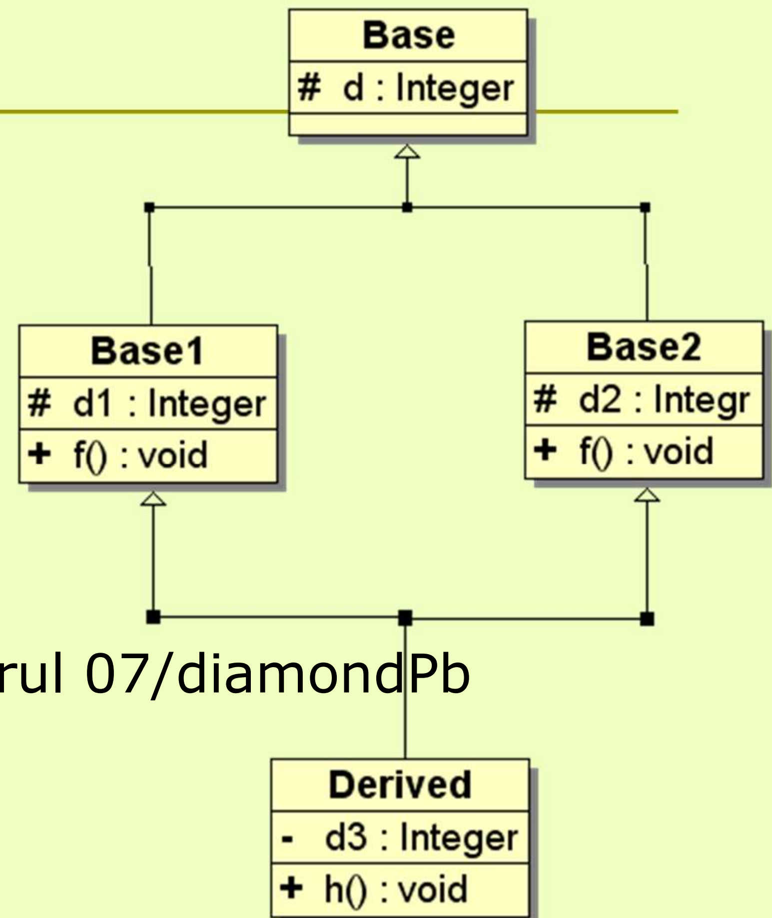
# Derivare multiplă

- 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

- Implementaţi următoarele clase (prin folosirea relaţiei de derivare):
  - Animal
    - nume,
    - greutate
  - Câine
    - nume,
    - greutate,
    - tip (Beagle, Coker, …),
    - culoareOchi
  - Peşte
    - nume,
    - Greutate,
    - tip (Calcan, Crap, Somn, Pastrav),
    - lungime

  şi folosiţi-le.

# Cursul următor

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