

Programmation Orientée Objet : Héritage multiple

Jean-Cédric Chappelier

Faculté I&C

Organisation du travail (semestre)

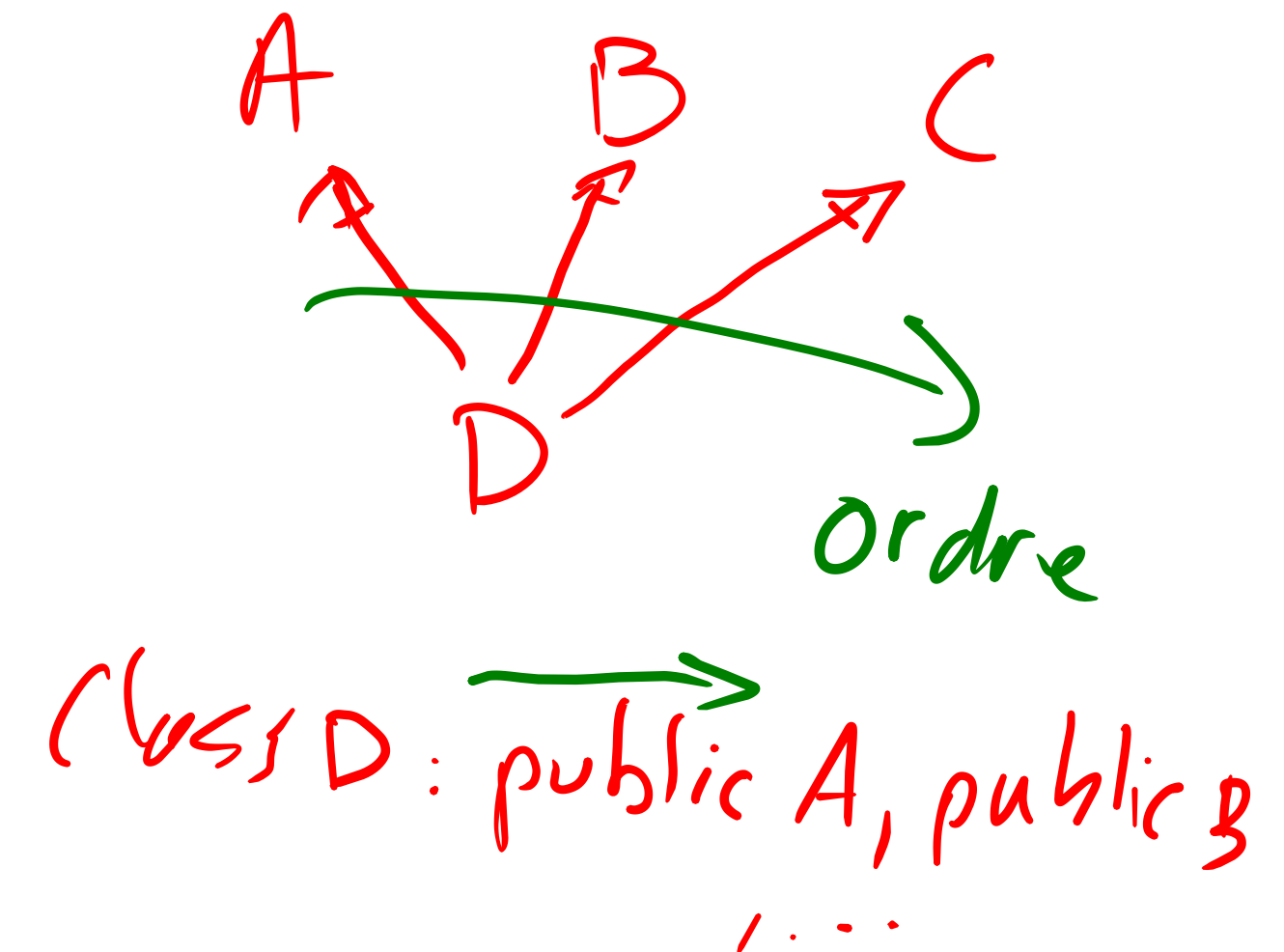
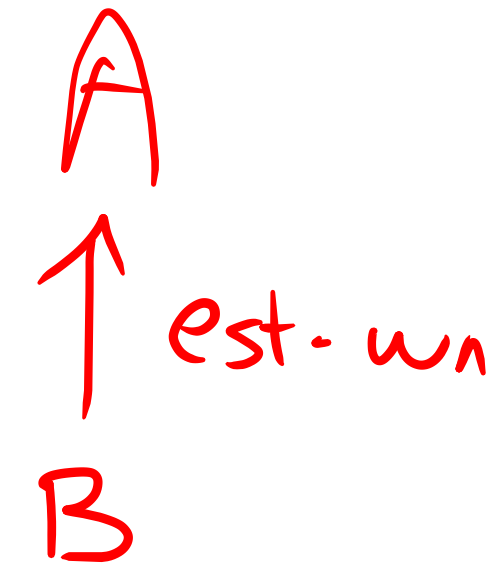
	MOOC	déc.	cours 1 h Jeudi 8-9	exercices 2 h Jeudi 9-11
1	19.02.26		0	Intro + compil. séparée
2	26.02.26	1. Intro POO	0	Intro POO
3	05.03.26	2. Constructeurs/Desi	0	Constructeurs
4	12.03.26	3. Surcharge des opé	0	Surcharge
5	19.03.26	4. Héritage	0	Héritage
6	26.03.26	5. Polymorphisme	0	Polymorphisme 1
7	02.04.26		1	Polymorphisme 2 / Collections hétérogènes
-	09.04.26		-	vacances Pâques
8	16.04.26	6. Héritage multiple	1	Héritage multiple
9	23.04.26		-	Midtem
10	30.04.26	(7. Etude de cas)	-	Templates
11	07.05.26		-	Structure de données abstraites ; Bibliothèques
12	14.05.26		-	(Ascension)
13	21.05.26	(7. Etude de cas)	-	Bibliothèques (fin) + Révisions
14	28.05.26		-	Examen

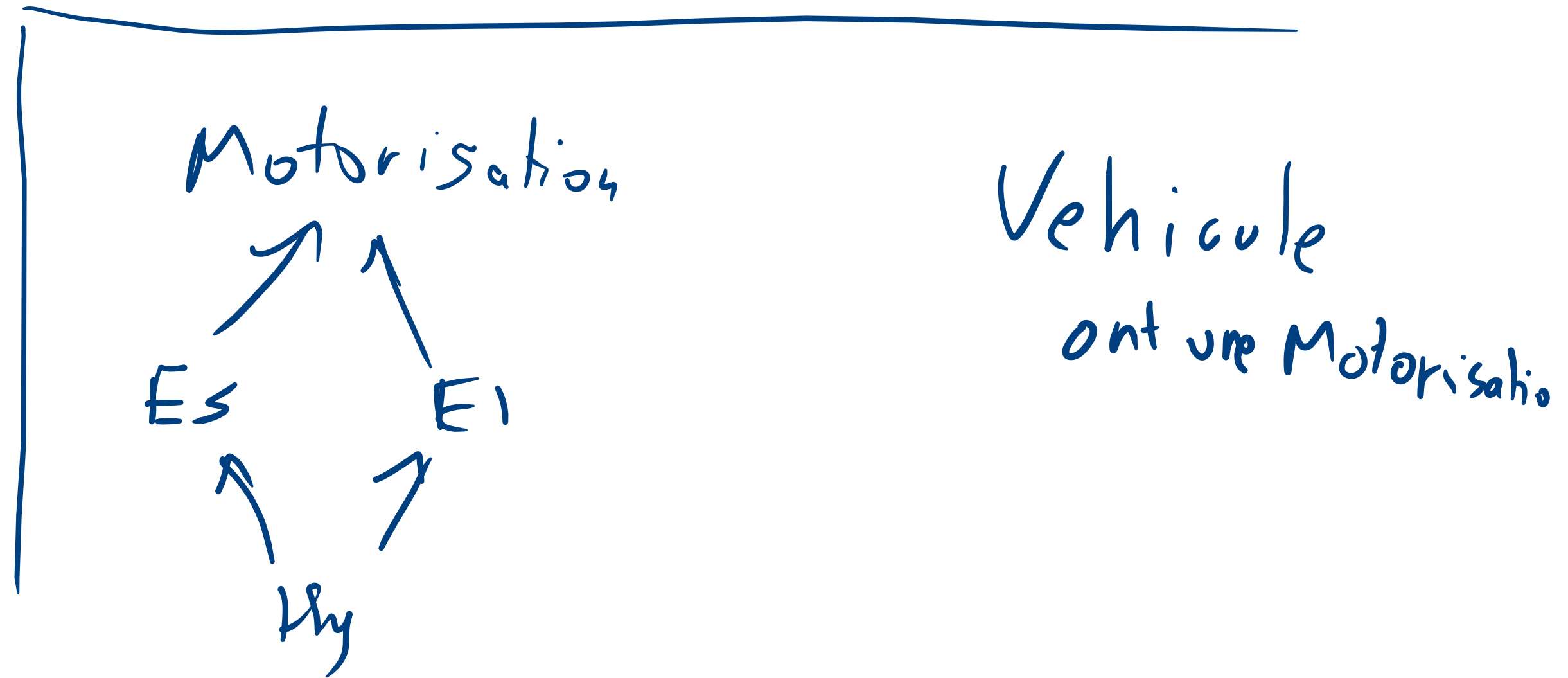
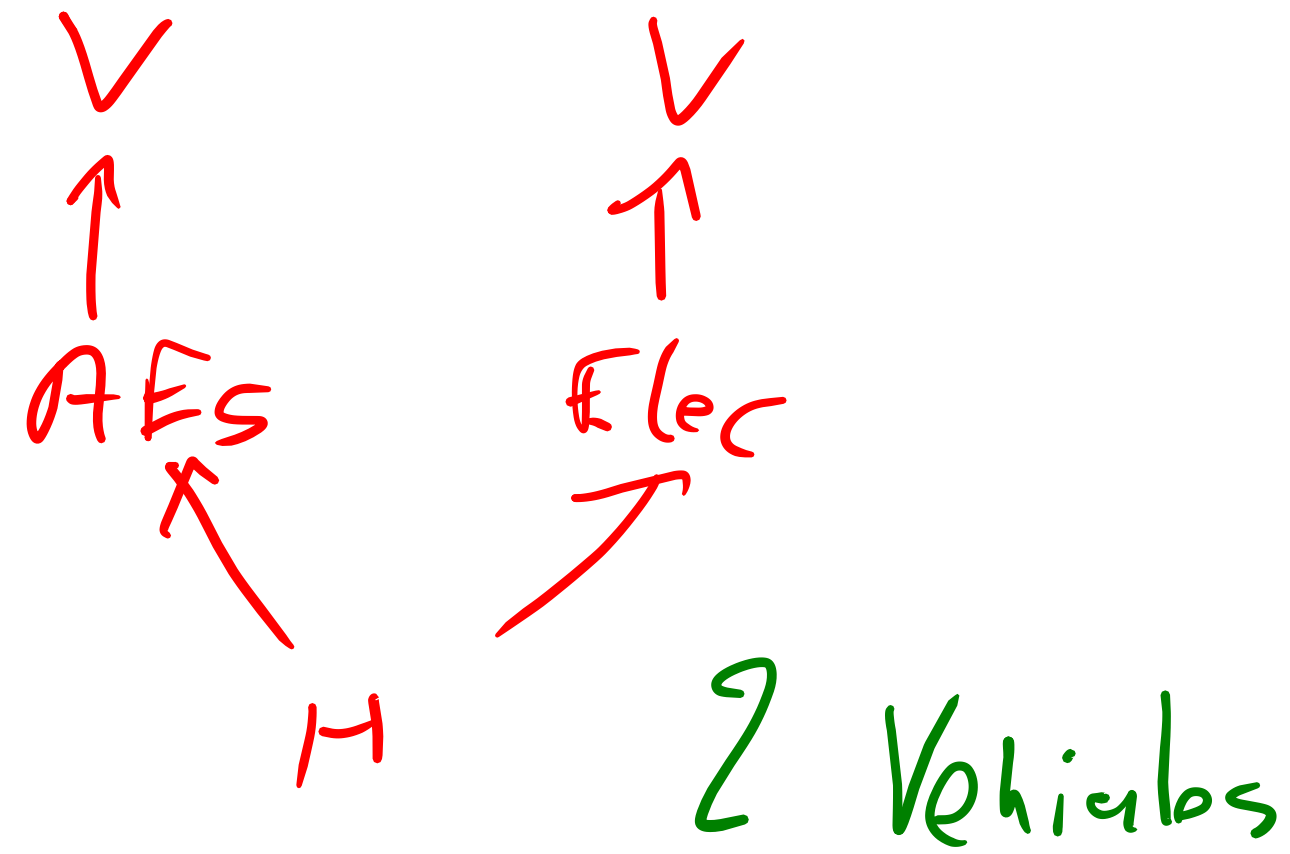
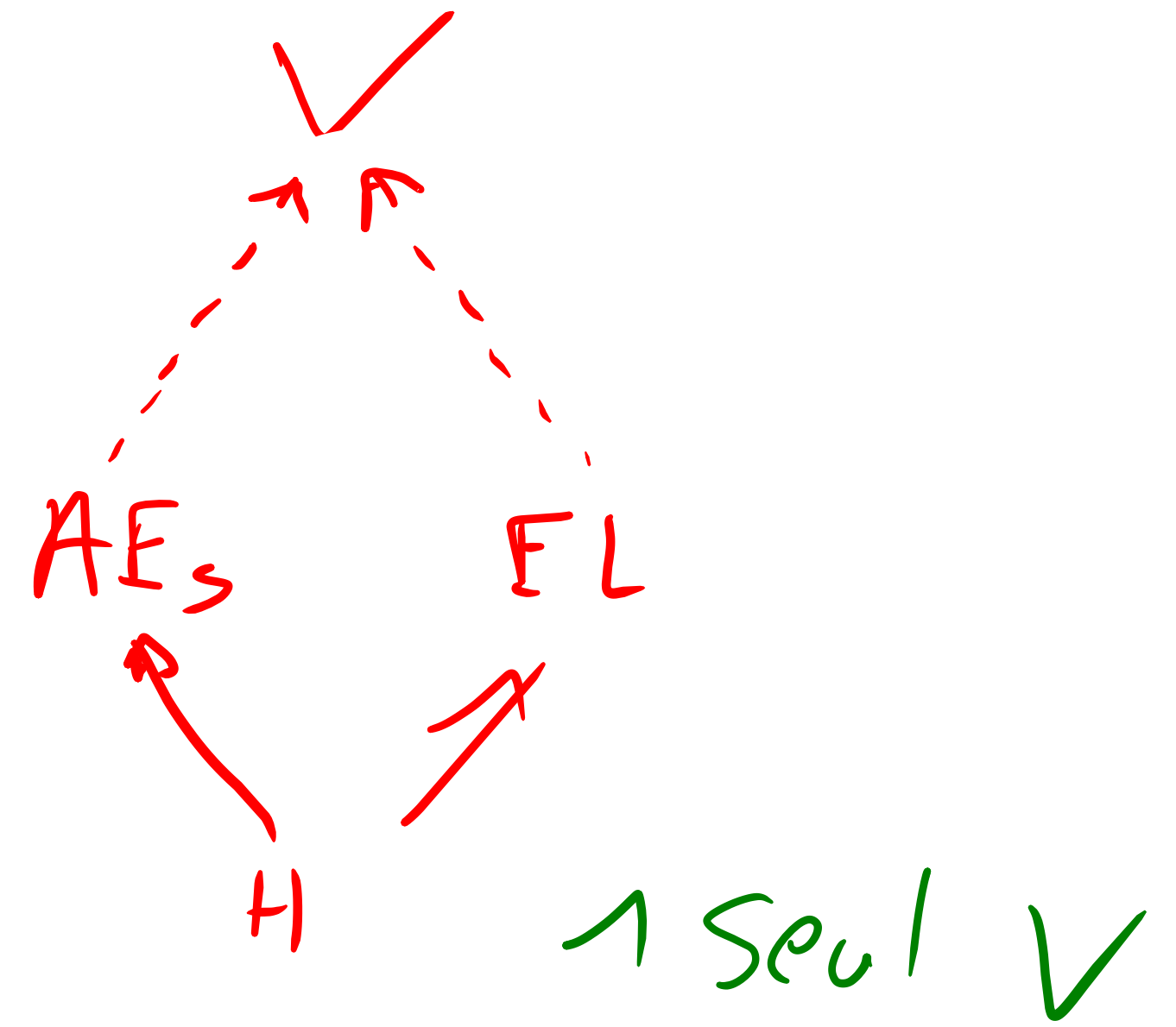
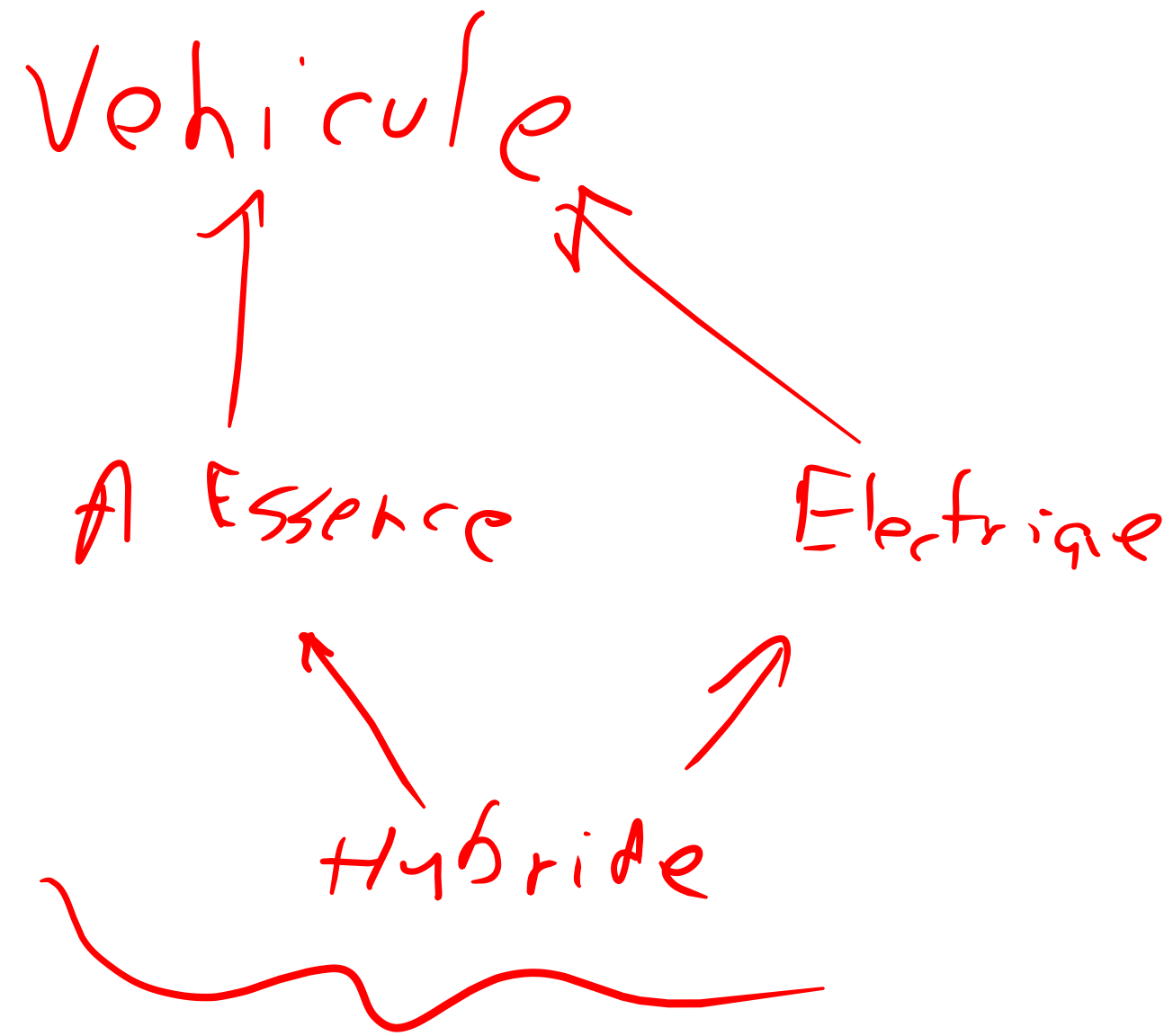
Objectifs de la leçon d'aujourd'hui

- ▶ Concepts fondamentaux
- ▶ Étude de cas

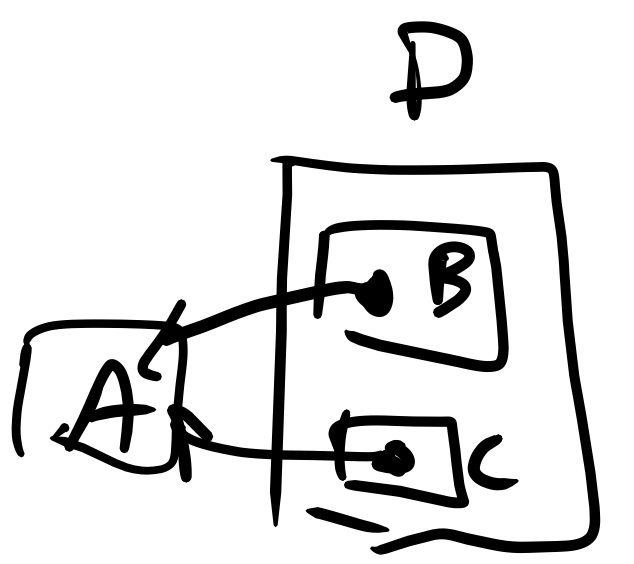
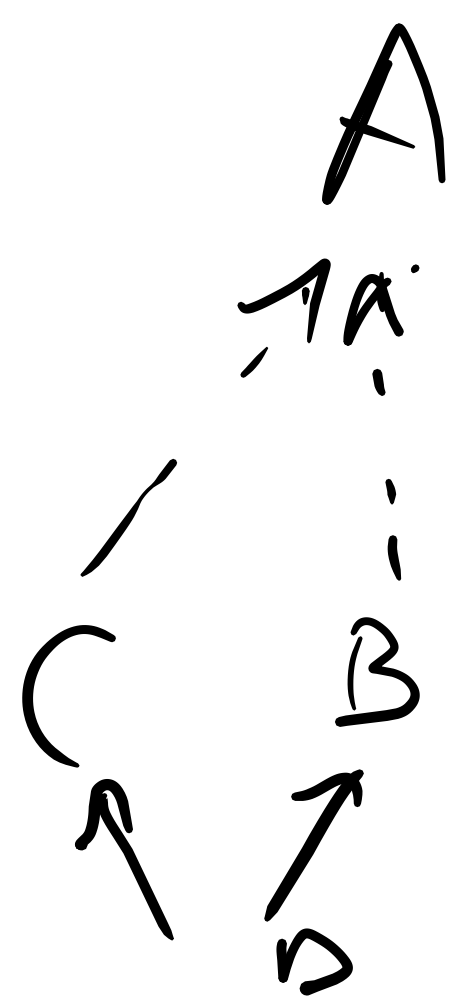
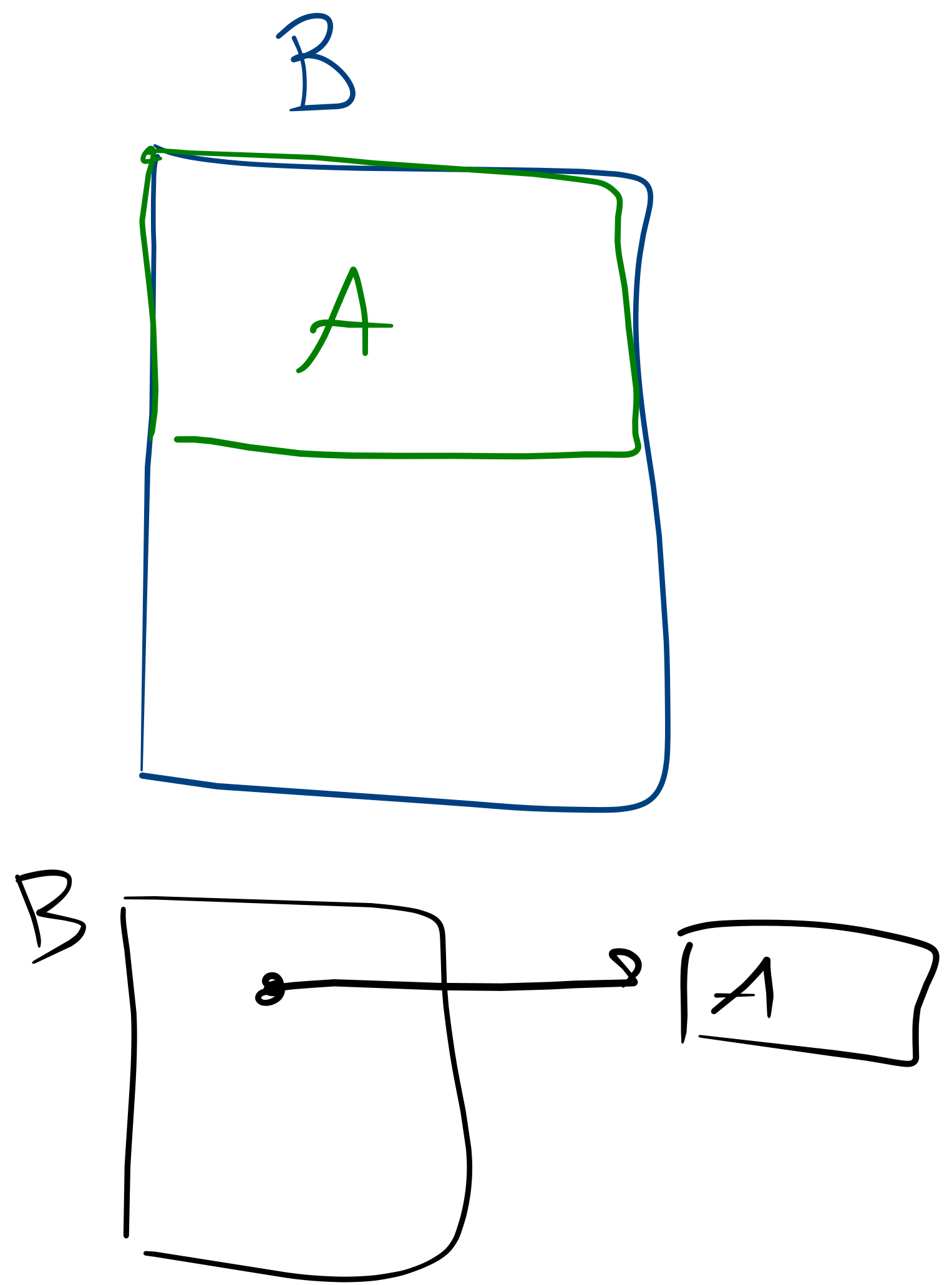
Concepts fondamentaux

- ▶ Buts, syntaxe (aucune difficulté)
- ▶ Ordre d'appel des constructeurs/destructeurs
 - ☞ ordre de déclaration *d'héritage*
- ▶ **Sens (= sémantique)** de l'héritage multiple ?
 - ▶ diagramme en losange
 - ▶ héritage et classes *virtuel(les)*
 - ▶ appel du constructeur de la classe virtuelle





A
↑
B



Etude de cas (simples)

Que faut-il corriger pour que le code suivant compile :

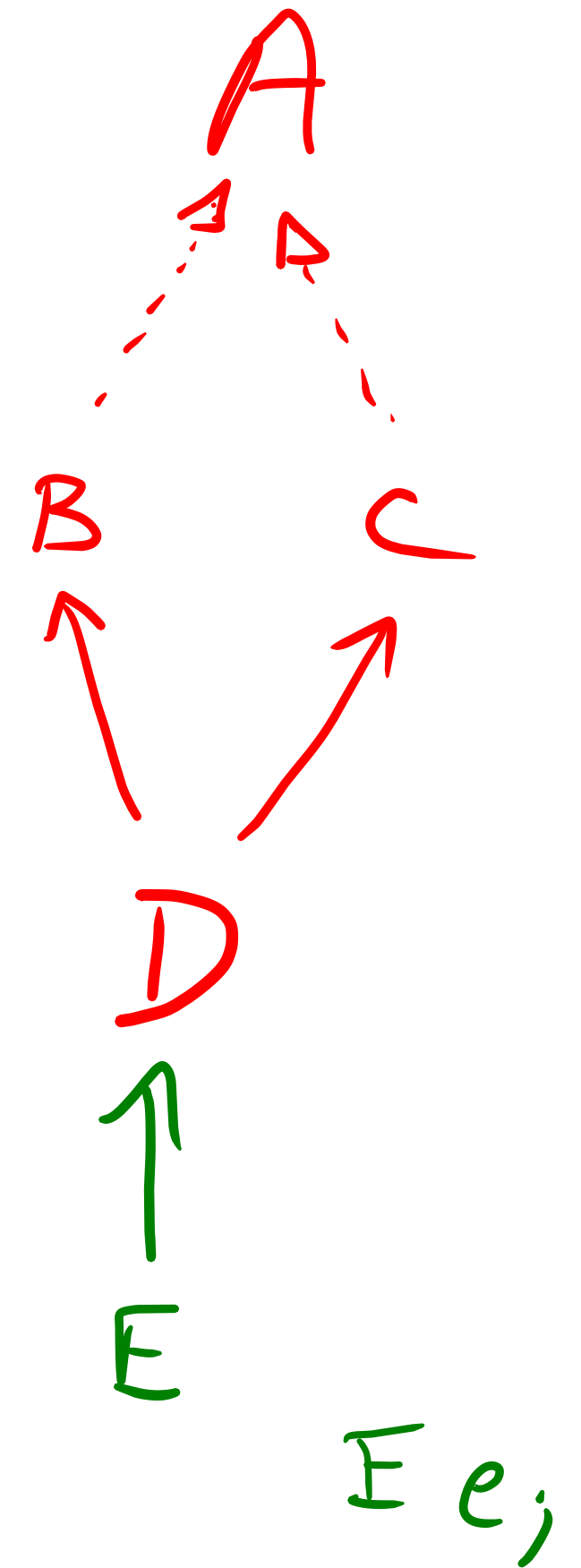
```
class A          { public: A(int x) : a(x) {}
                  private: int a;          };

class B : public virtual A { public: B() : A(0) {} };

class C : public virtual A { public: C() : A(1) {} };

class D : public B, public C { };

int main()
{
    D d1;
    return 0;
}
```



TROIS solutions

Il y a *trois* possibilités :

- ▶ ajouter un constructeur par défaut à D avec appel explicite au constructeur de A :

```
class A          { public: A(int x) : a(x) {}  
                  private: int a;          };  
  
class B : public virtual A { public: B() : A(0) {} };  
  
class C : public virtual A { public: C() : A(1) {} };  
  
class D : public B, public C { public: D() : A(42) {} };  
  
int main()  
{  
    D d1;  
    return 0;  
}
```

TROIS solutions

Il y a *trois* possibilités :

- ▶ ajouter un constructeur par défaut à A :

```
class A { public: A(int x = 42) : a(x) {}  
        private: int a;                };  
  
class B : public virtual A { public: B() : A(0) {} };  
  
class C : public virtual A { public: C() : A(1) {} };  
  
class D : public B, public C { };  
  
int main()  
{  
    D d1;  
    return 0;  
}
```

TROIS solutions

Il y a *trois* possibilités :

- ▶ supprimer les héritages virtuels

```
class A { public: A(int x) : a(x) {}  
        private: int a;           };  
  
class B : public A { public: B() : A(0) {} };  
  
class C : public A { public: C() : A(1) {} };  
  
class D : public B, public C { };  
  
int main()  
{  
    D d1;  
    return 0;  
}
```

Cas numéro 2

Le code suivant compile-t-il ?

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

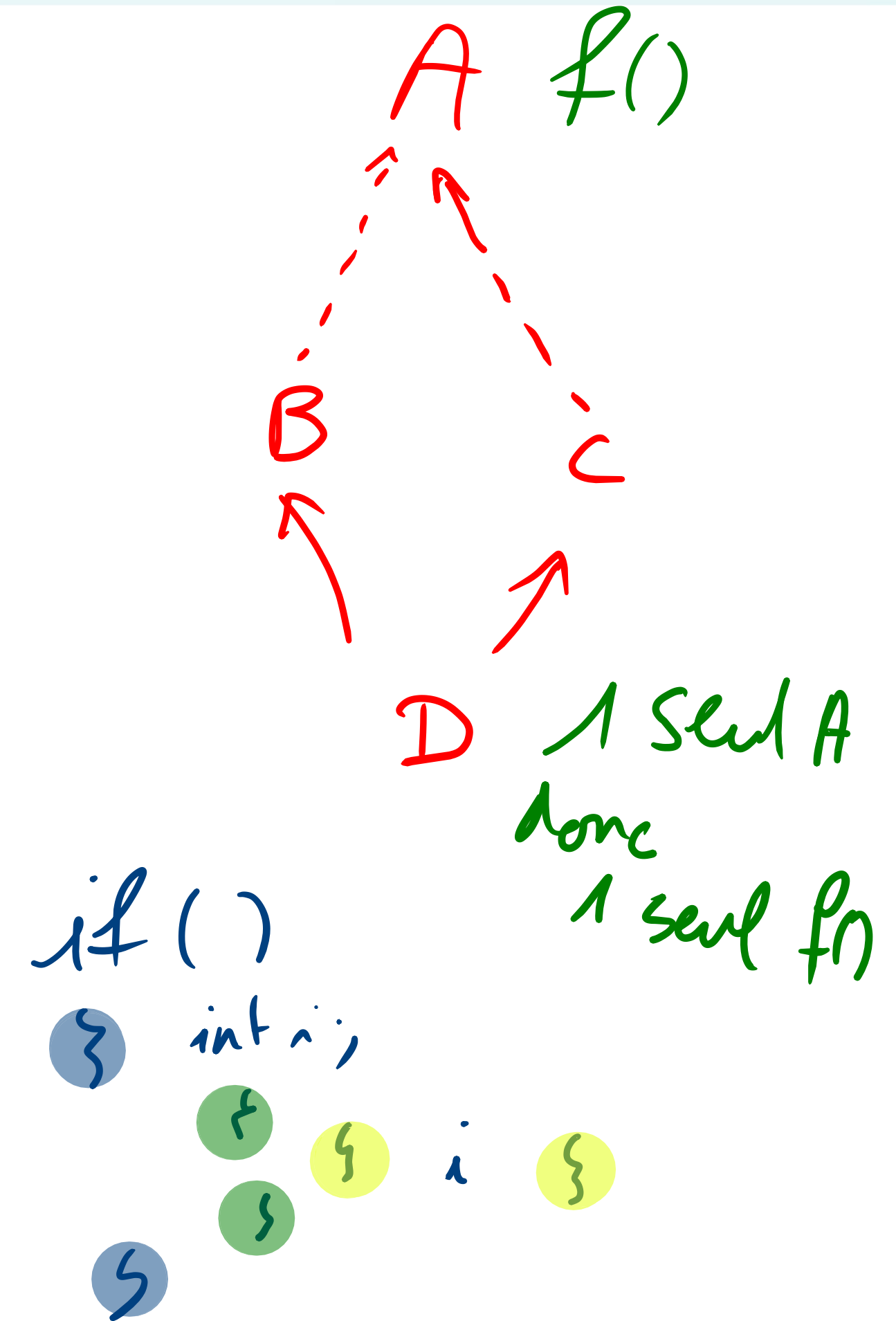
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ };

class C : public virtual A
{ };

class D : public B, public C
{ };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```



Cas numéro 2

Le code suivant compile-t-il ?  **OUI!** ...et sans les `virtual` ? non ! (ambiguïté)

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

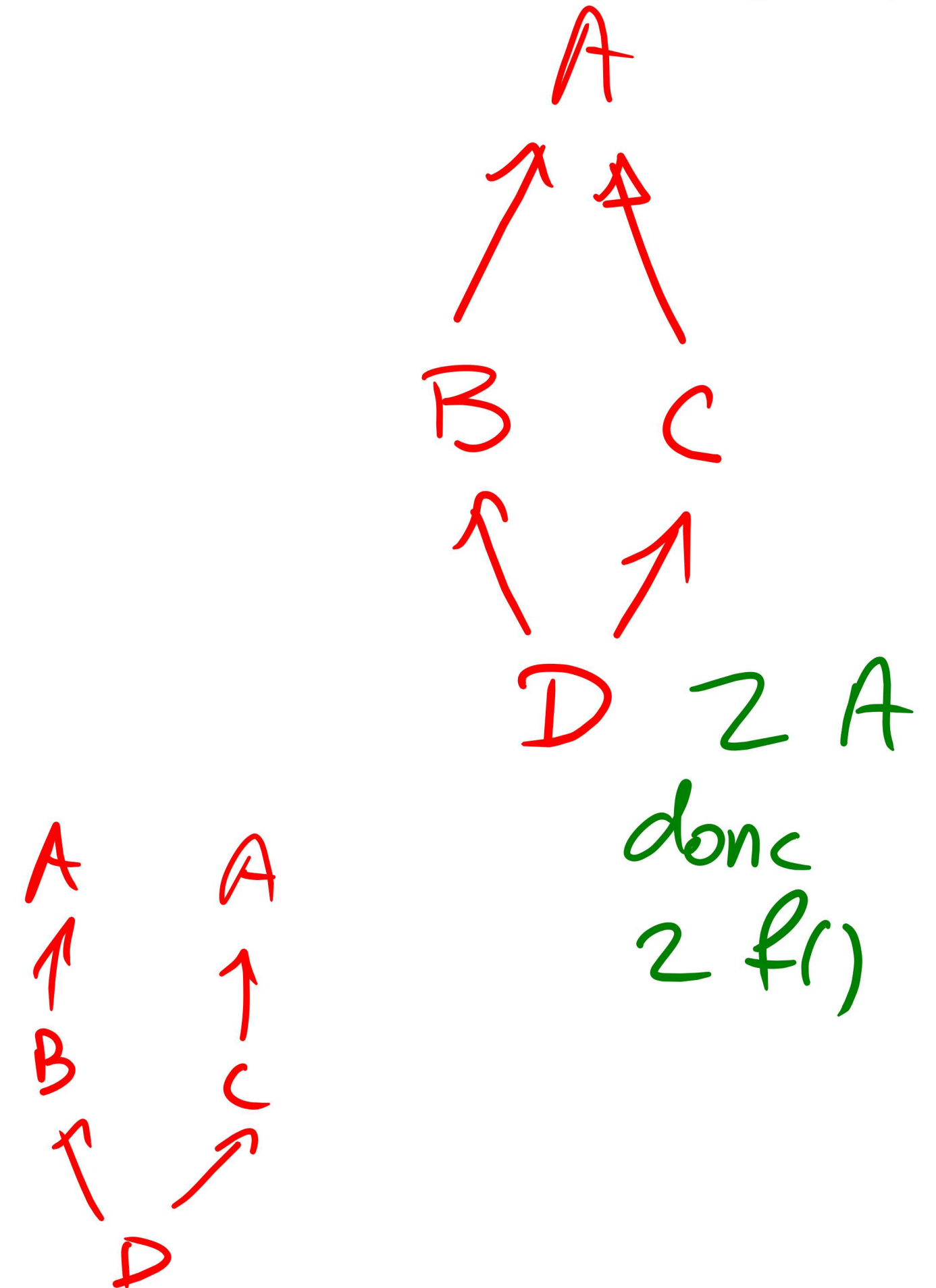
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ };

class C : public virtual A
{ };

class D : public B, public C
{ };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```



Cas numéro 3

Le code suivant compile-t-il ?

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

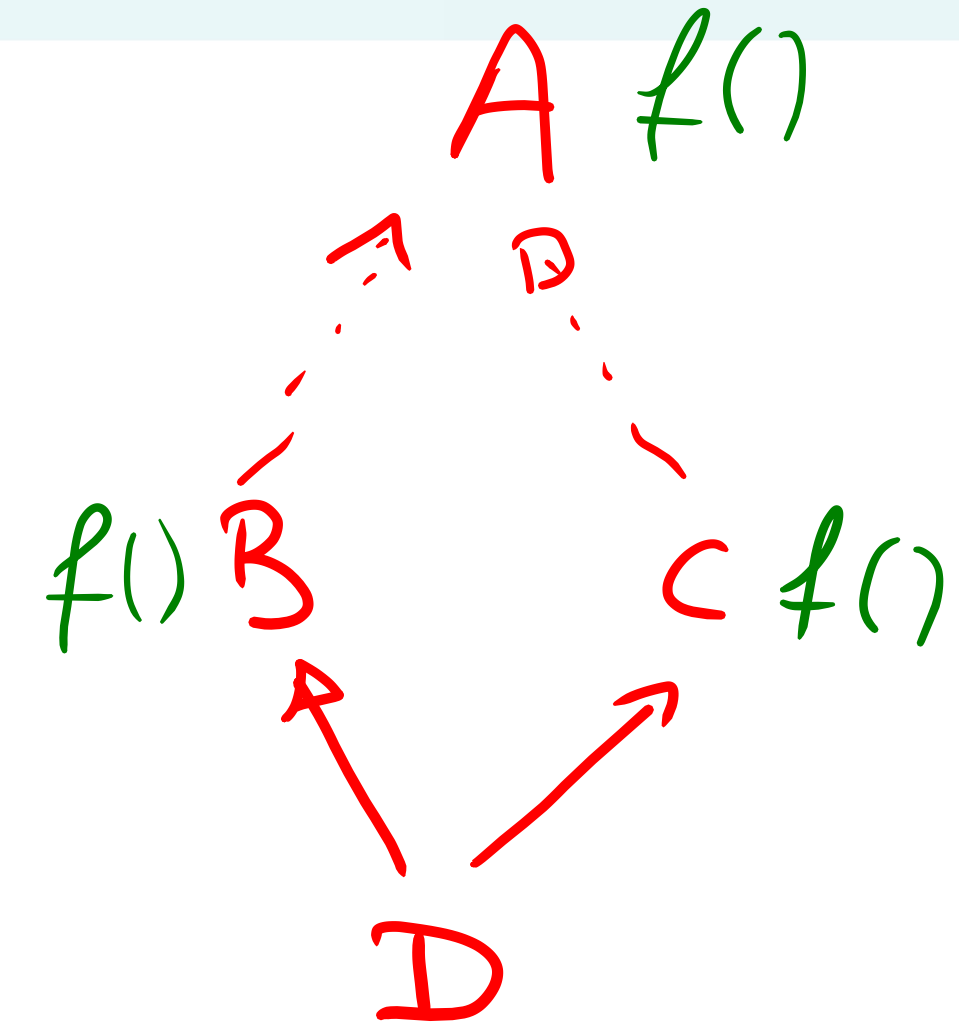
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ public: void f() const { cout << "C "; } };

class D : public B, public C
{ };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```



Cas numéro 3 : QUATRE solutions

Il y a *quatre* corrections possibles : ① **supprimer une des ambiguïtés** :

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

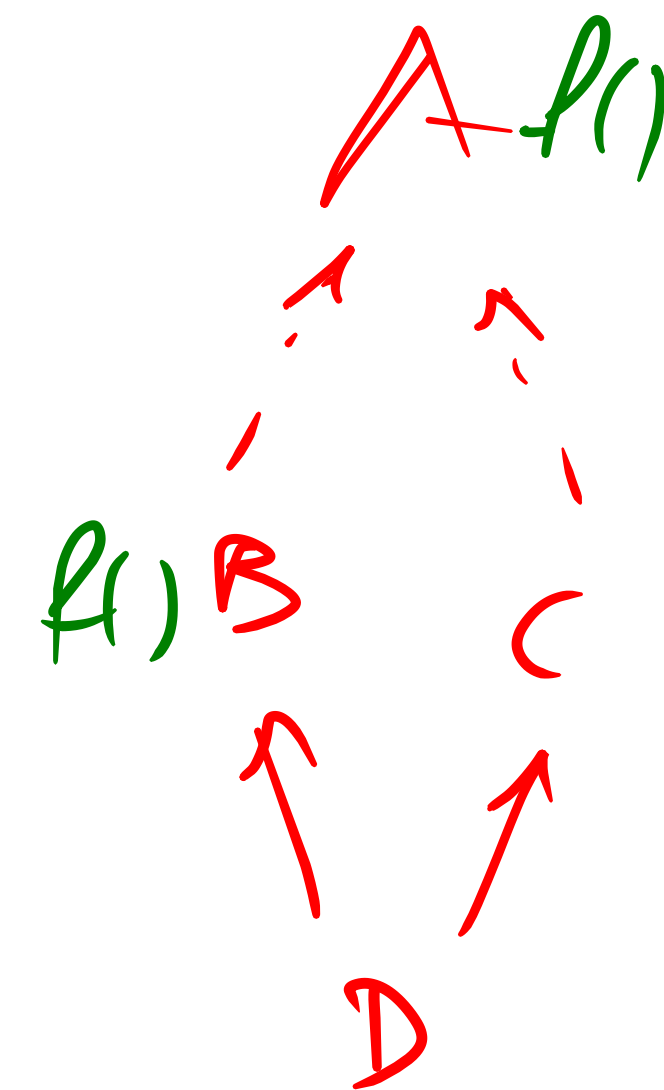
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ };

class D : public B, public C
{ };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```



Cas numéro 3 : QUATRE solutions

Il y a *quatre* corrections possibles : ② **désambiguiser l'appel** :

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

class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ public: void f() const { cout << "C "; } };

class D : public B, public C
{ };

int main()
{
    D d1;
    d1.A::f(); // ou B:: ou C::
    return 0;
}
```

Cas numéro 3 : QUATRE solutions

Il y a *quatre* corrections possibles : ③ **désambiguiser à l'aide de `using`** :

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

class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ public: void f() const { cout << "C "; } };

class D : public B, public C
{ public: using A::f; }; // ou B::f ou C::f

int main()
{
    D d1;
    d1.f();
    return 0;
}
```

Cas numéro 3 : QUATRE solutions

Il y a *quatre* corrections possibles : ④ **désambigüiser en redéfinissant :**

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

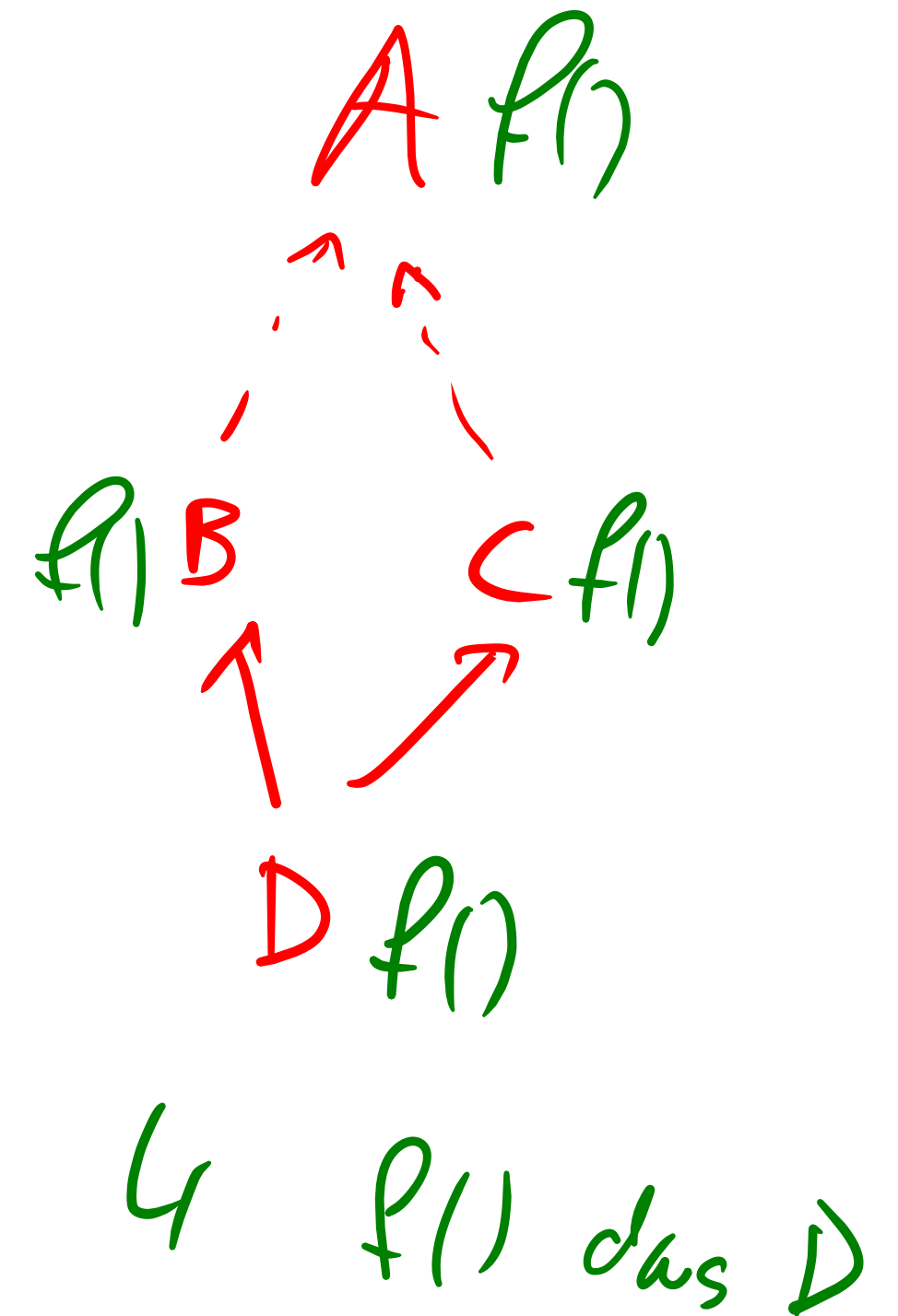
class A
{ public: void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };

class C : public virtual A
{ public: void f() const { cout << "C "; } };

class D : public B, public C
{ public: void f() const { cout << "D "; } };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```



Cas numéro 4

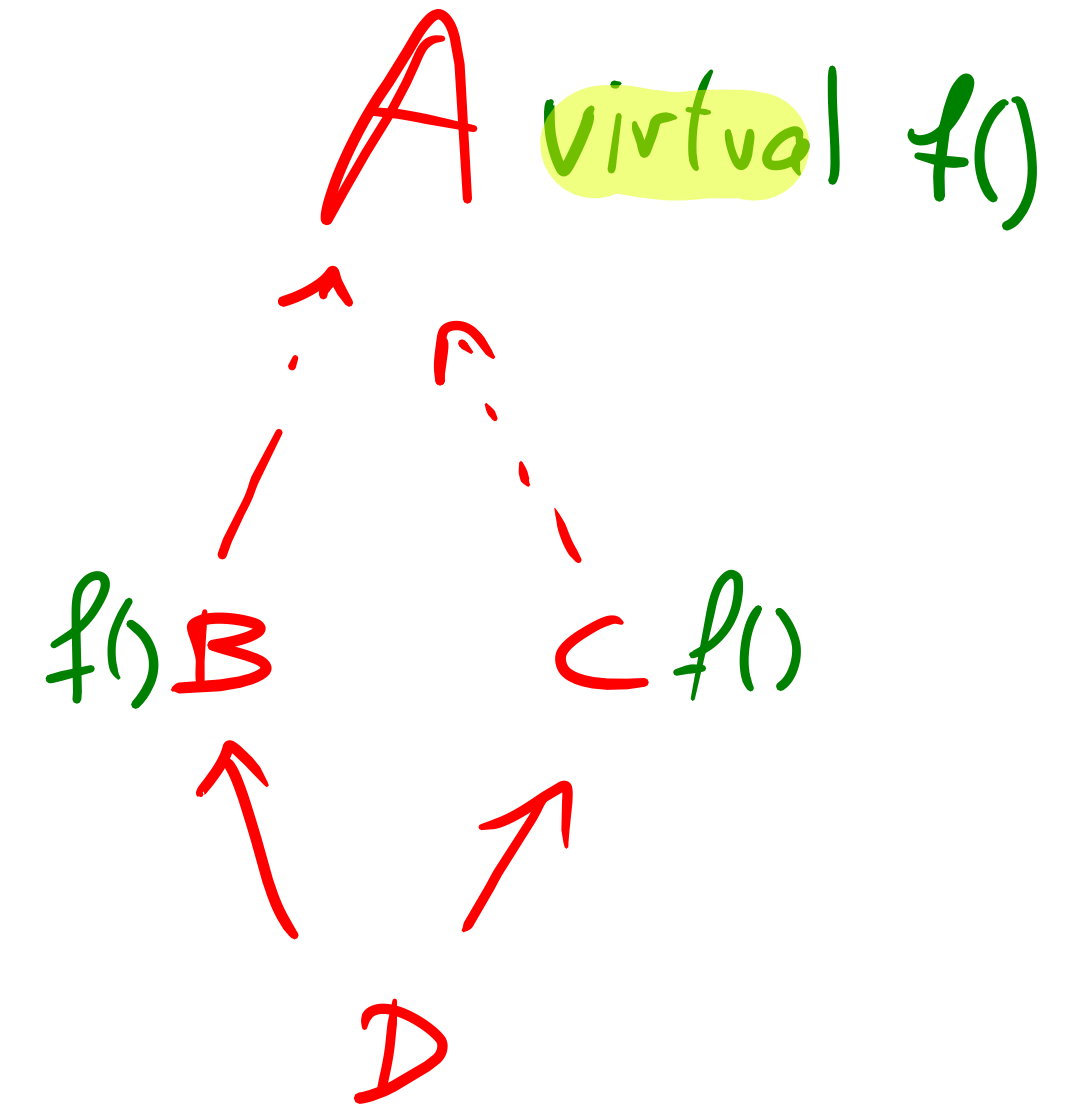
Le code suivant compile-t-il ? Si oui, qu'affiche-t-il ?

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

class A
{ public: virtual void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const { cout << "B "; } };
      virtual      ^      override
class C : public virtual A
{ public: void f() const { cout << "C "; } };
      ^
class D : public B, public C
{ };

int main()
{
    D d1;
    d1.f();
    return 0;
}
```



Cas numéro 4 : DEUX solutions



Il n'y a ici que *deux* corrections possibles :

les solutions

- ▶ 2 (désambigüiser l'appel) :

```
d1.A::f(); // ni B:: ni C::
```

- ▶ et 3 (utiliser `using`) :

```
class D : public B, public C  
{ public: using A::f; }; // ni B::f ni C::f
```

NE fonctionnent **PAS** :

no unique final override for
'virtual void A::f() const' in 'D'

De la norme C++ :

« *In a derived class, if a virtual member function of a base class subobject has more than one final override the program is ill-formed.* »

Cas numéro 4 : DEUX solutions

Ce qui ne nous laisse que *deux* corrections possibles :

- ▶ supprimer l'ambiguïté (ce qui n'est souvent pas possible) ;
- ▶ redéfinir la méthode :

```
class A
{ public: virtual void f() const { cout << "A "; } };

class B : public virtual A
{ public: void f() const override { cout << "B "; } };

class C : public virtual A
{ public: void f() const override { cout << "C "; } };

class D : public B, public C
{ public: void f() const override { cout << "D "; } };
```

Dernière question

Quelle différence entre les cas 3 et 4 ?
Que change le `virtual` ?
Donnez un exemple *illustratif*.

```
D un_d;  
A* ptr(&un_d);  
ptr->f();
```