

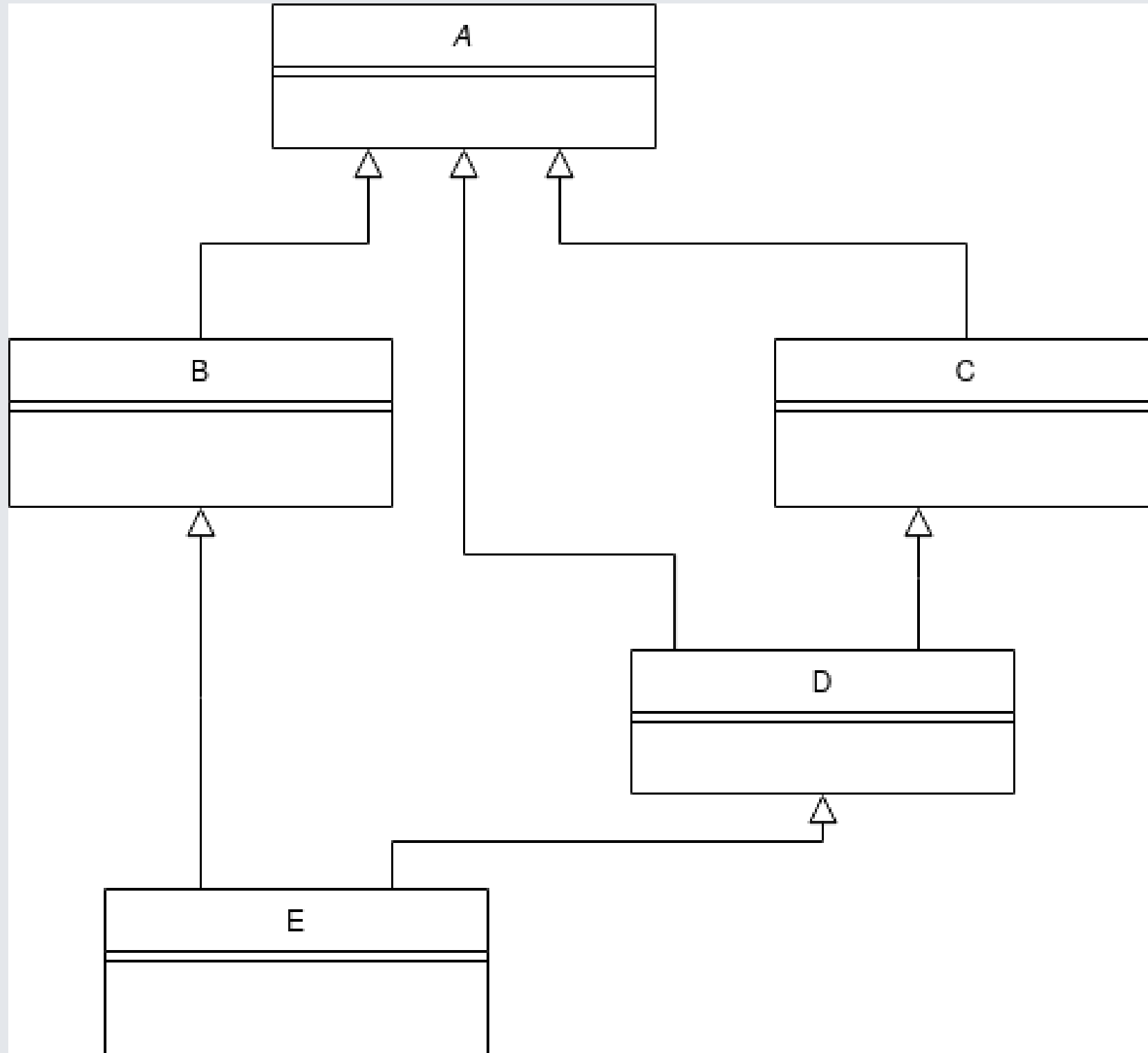
LANGUAGE OBJ. AV. (C++) MASTER 1

U.F.R. d'Informatique
Université de Paris Cité

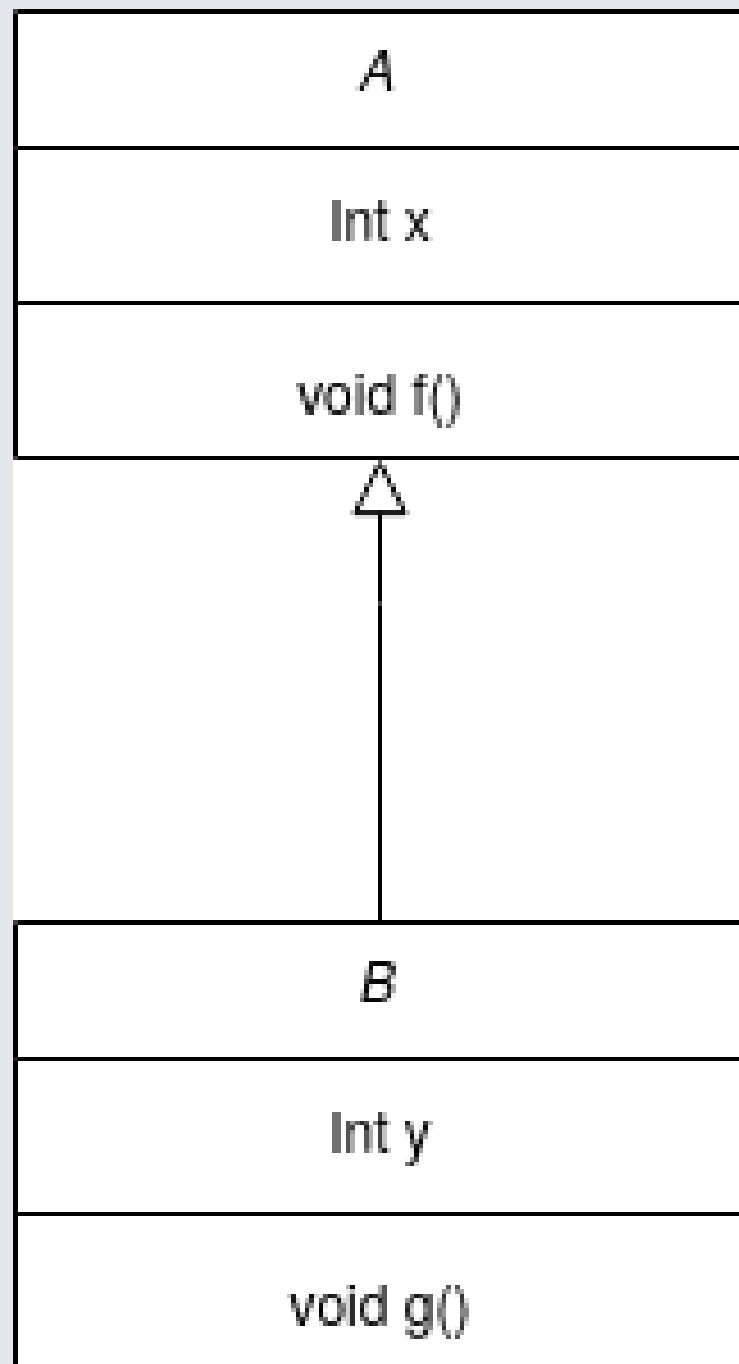
Cette semaine :

- Héritage multiple
- Correction du TP noté

L'HÉRITAGE MULTIPLE...



Revenons un peu sur l'héritage simple



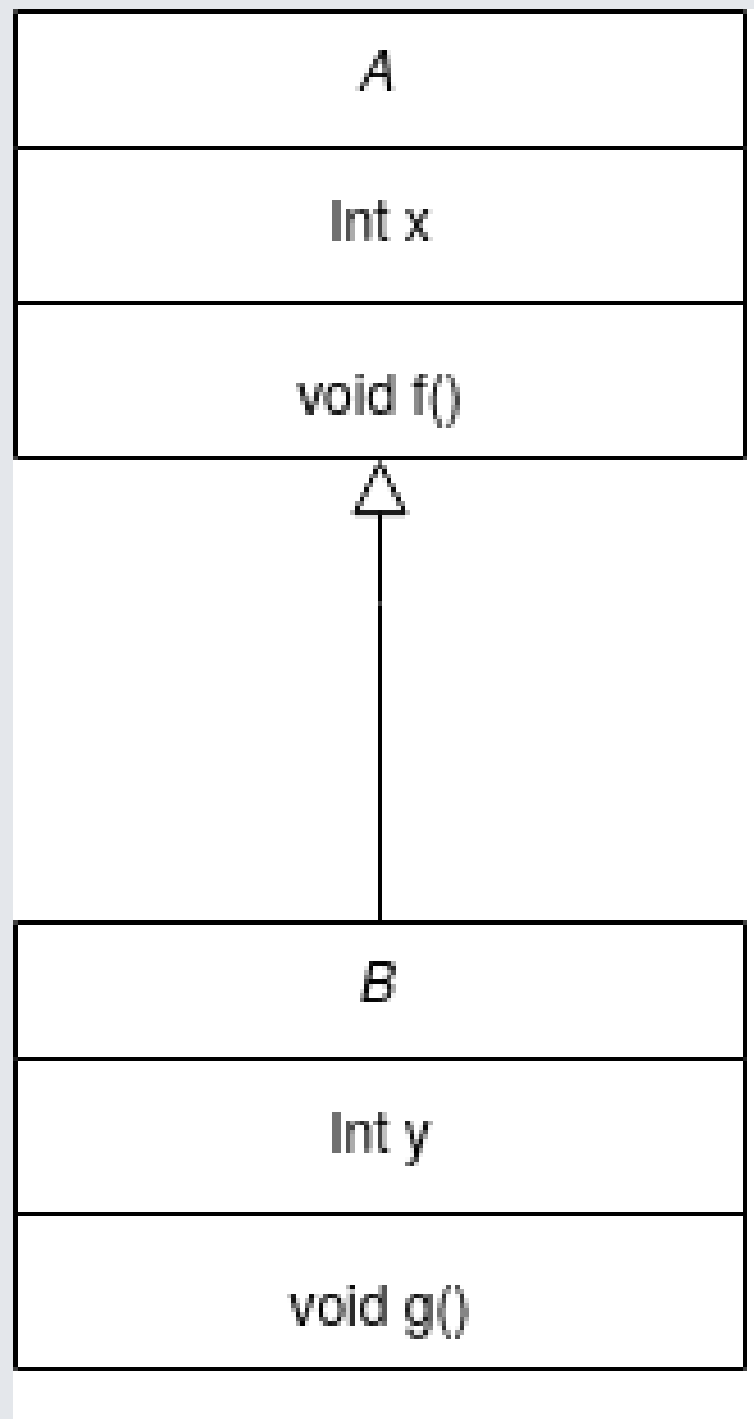
```
class A {
public :
    void f();
    int x;
    A(int =0);
};
```

```
class B : public A {
public :
    void g();
    int y;
    B(int =1);
};
```

```
void A::f() {
    cout << "A::f " << x << endl;
}
A::A(int x) :x{x} {}
```

```
void B::g() {
    cout << "B::g " << y << endl;
}
B::B(int y) :y{y} {}
```

Revenons un peu sur l'héritage simple

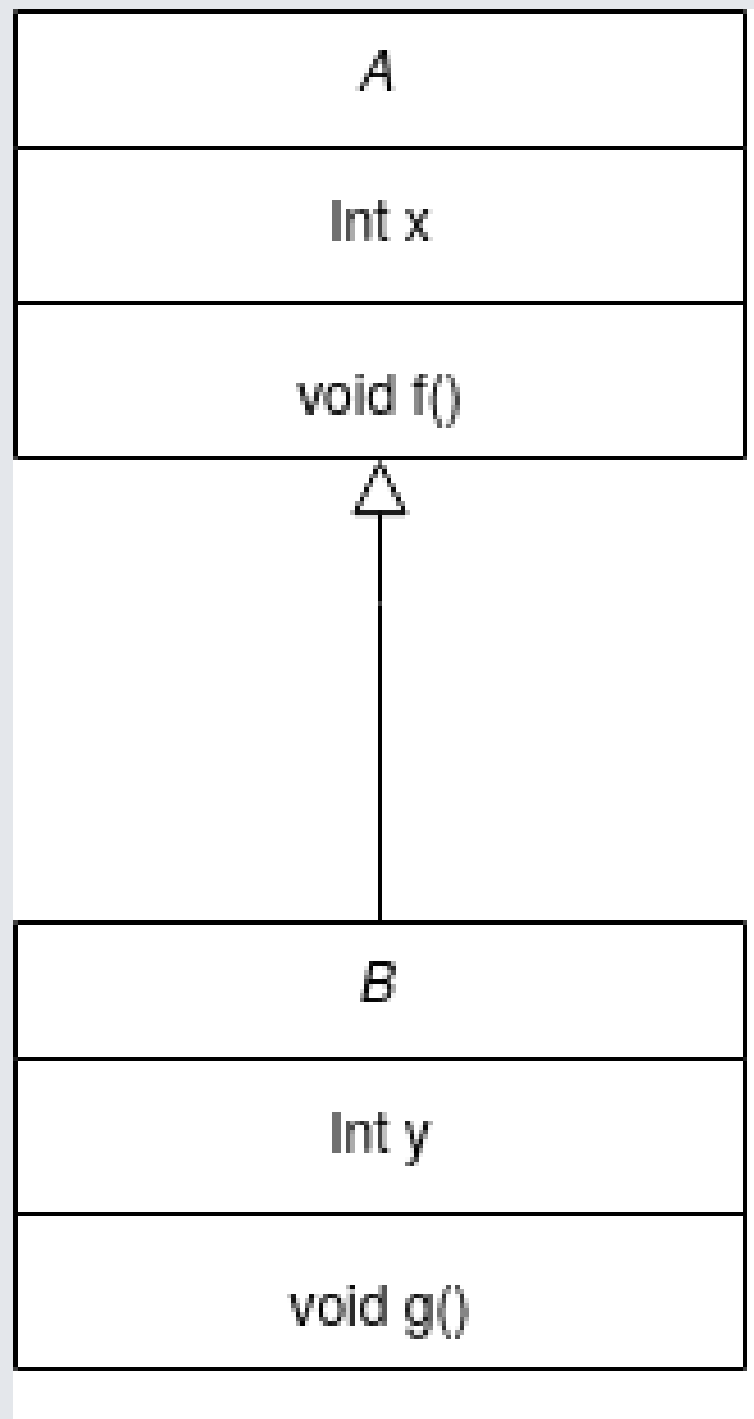


B hérite de A :

- un objet B b "est un" A
- il "possède" attributs et méthodes de A

d'une certaine façon
b "possède toute une partie
de type A"

Revenons un peu sur l'héritage simple



B hérite de A :

- un objet B b "est un" A
- il "possède" attributs et méthodes de A

d'une certaine façon
b "possède toute une partie
de type A"

pas très différent d'une composition
avec en + du sous typage

D'ailleurs, si on cache le sous typage,
il est difficile de distinguer :

```
class A {  
public :  
    void f();  
    int x;  
    A(int =0);  
};
```

```
class B : private A {  
public :  
    void g();  
    int y;  
    B(int =1);  
};
```

Un héritage privé
cache le sous typage

```
class B{  
private :  
    A a;  
    void f();  
    int &x;  
public :  
    void g();  
    int y;  
    B(int =1);  
};
```

```
void B::f() {a.f();}  
  
B::B(int y) :  
             y{y},  
             x{a.x} {}
```

D'ailleurs, si on cache le sous typage,
il est difficile de distinguer :

```
class A {  
public :  
    void f();  
    int x;  
    A(int =0);  
};
```

```
class B : private A {  
public :  
    void g();  
    int y;  
    B(int =1);  
};
```

Un héritage privé
cache le sous typage

On écrit
que B se
compose
d'un A

```
class B{  
private :  
    A a;  
    void f();  
    int &x;  
public :  
    void g();  
    int y;  
    B(int =1);  
};
```

```
void B::f() {a.f();}  
  
B::B(int y) :  
                y{y},  
                x{a.x} {}
```


D'ailleurs, si on cache le sous typage, il est difficile de distinguer :

```
class A {  
public :  
    void f();  
    int x;  
    A(int =0);  
};
```

```
class B : private A {  
public :  
    void g();  
    int y;  
    B(int =1);  
};
```

Une héritage privé cache le sous typage

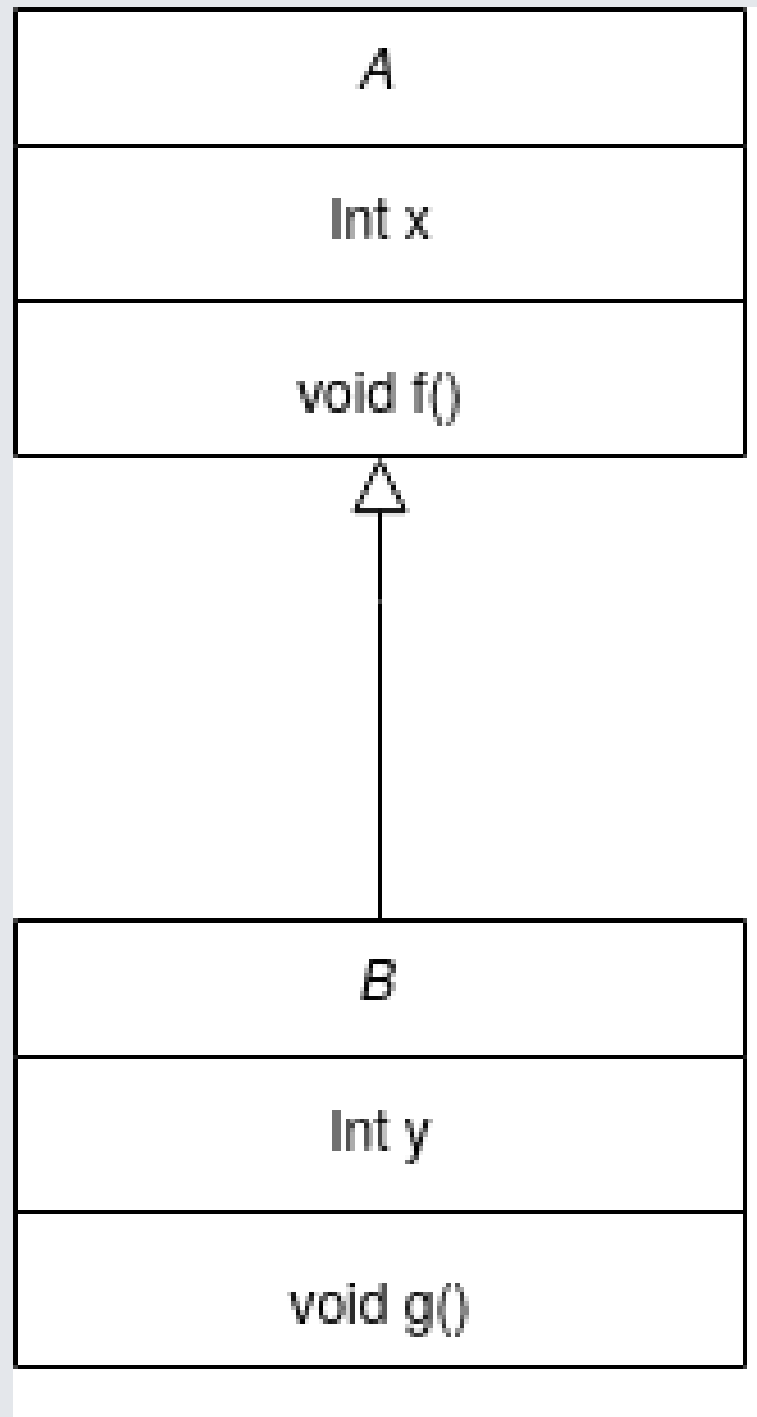
On écrit que B se compose d'un A

```
class B{  
private :  
    A a;  
    void f();  
    int &x;  
public :  
    void g();  
    int y;  
    B(int =1);  
};
```

```
void B::f() {a.f();}  
  
B::B(int y) :  
    y{y},  
    x{a.x} {}
```

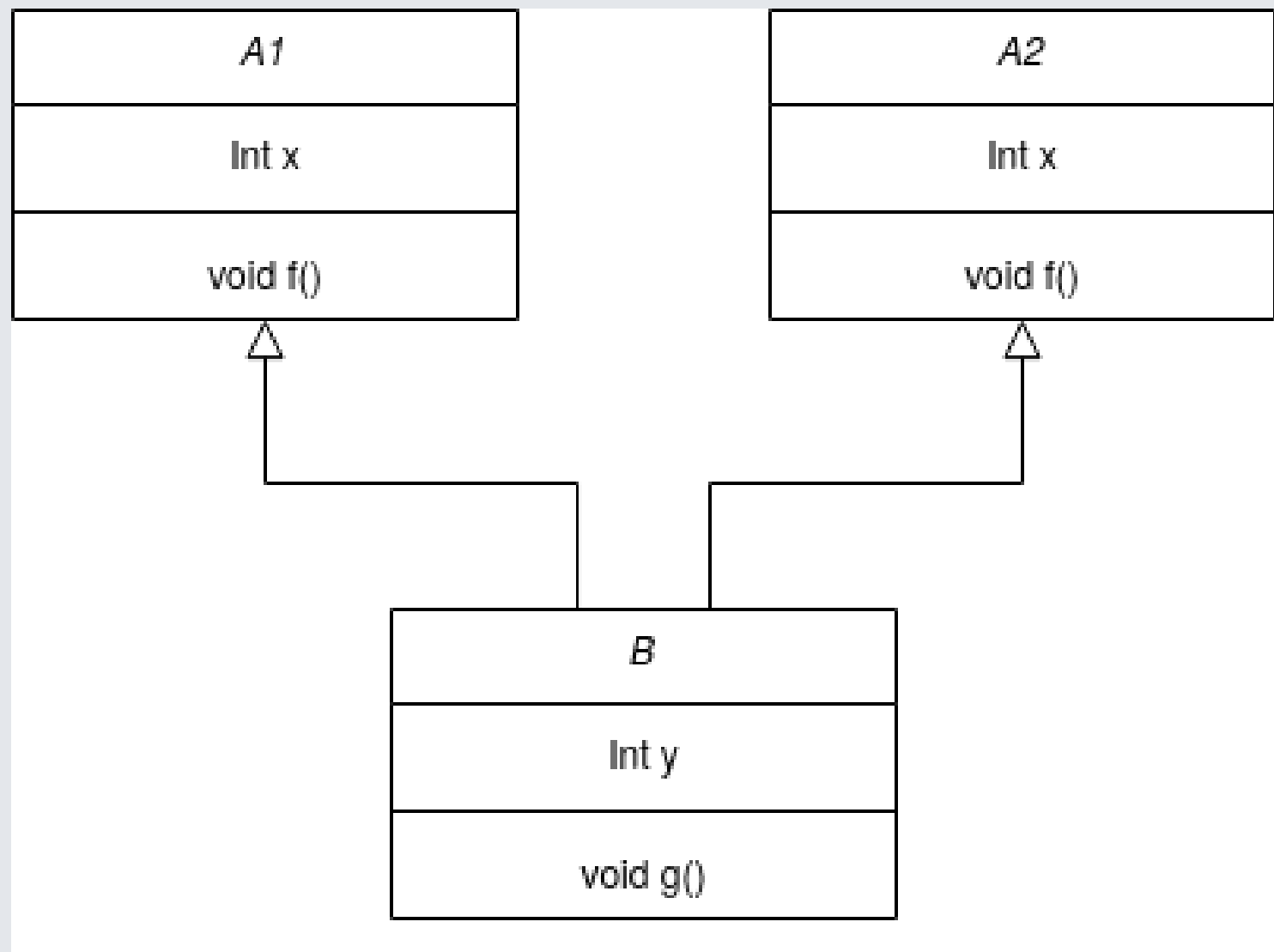
et on fait en sorte de lui donner des accès équivalents aux méthodes + attributs

Retenons que



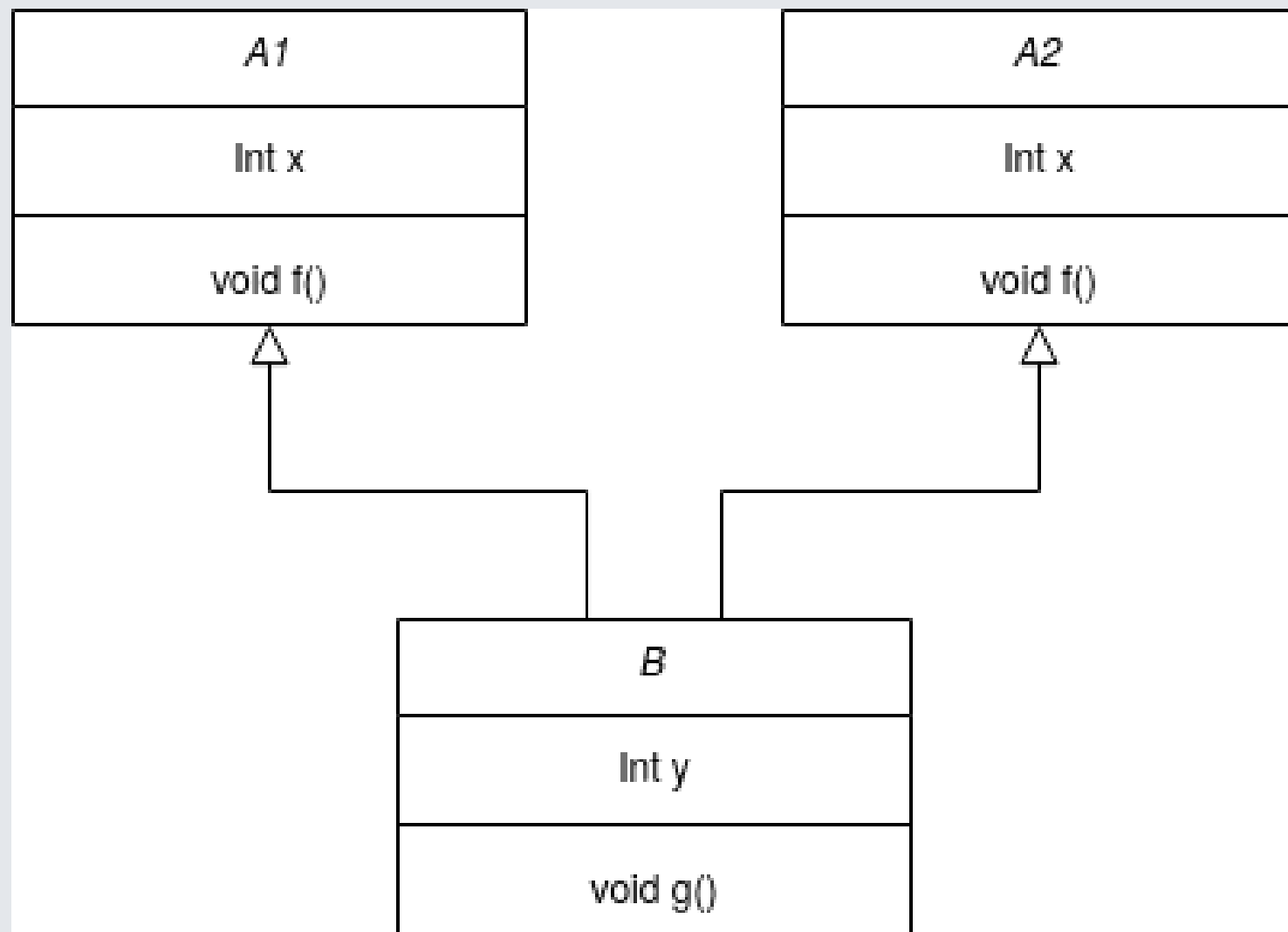
d'une certaine façon
b "possède toute une partie
de type A"

Cas de l'héritage multiple :



Il peut conduire à des ambiguïtés :
B doit pouvoir faire `f()`,
et possède un attribut `x`

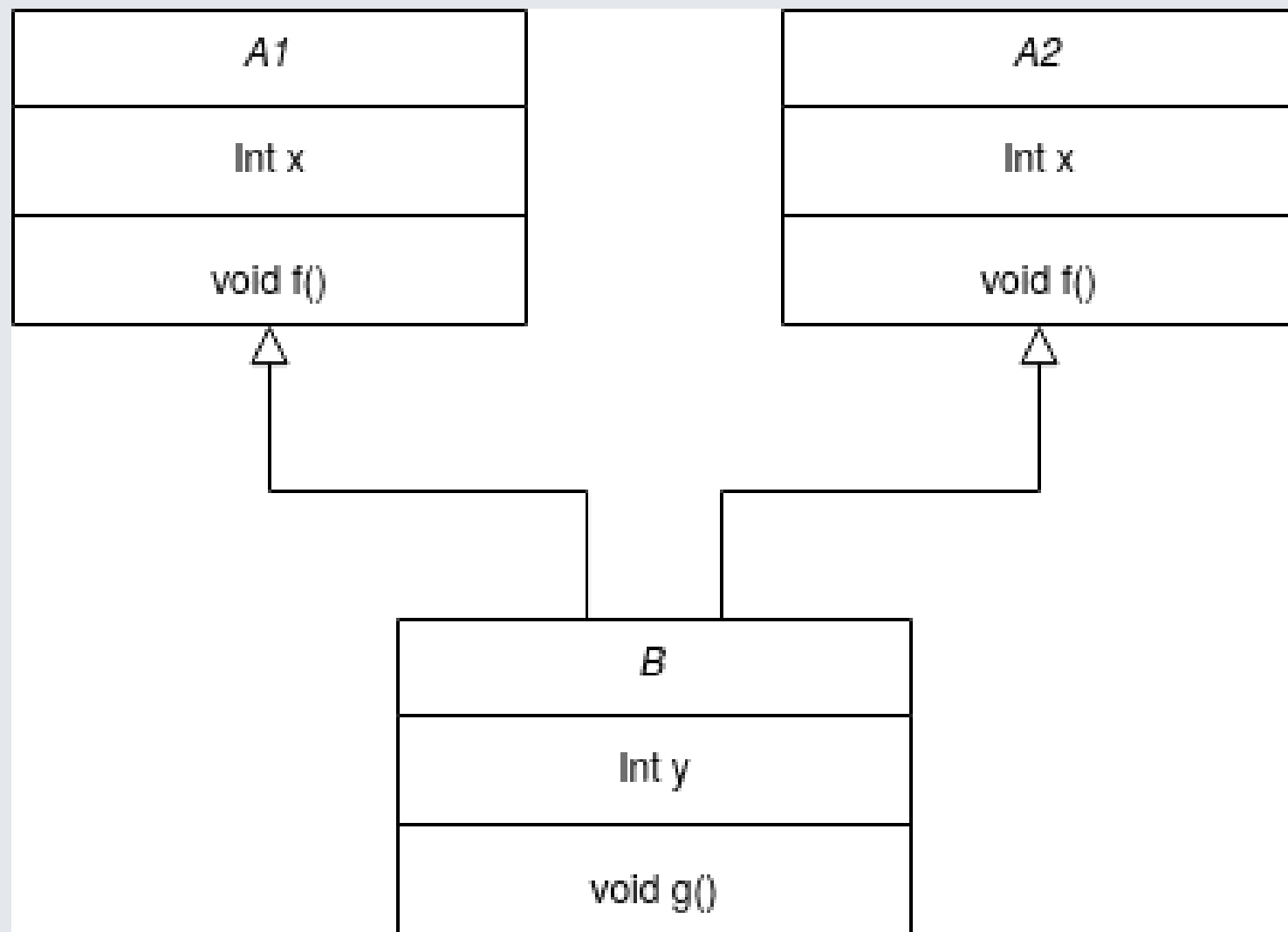
Cas de l'héritage multiple :



mais... d'une certaine façon **b** "possède toute une partie de type **A1**, et une partie de type **A2**"

Il peut conduire à des ambiguïtés :
B b doit pouvoir faire **f()**,
et possède un attribut **x**

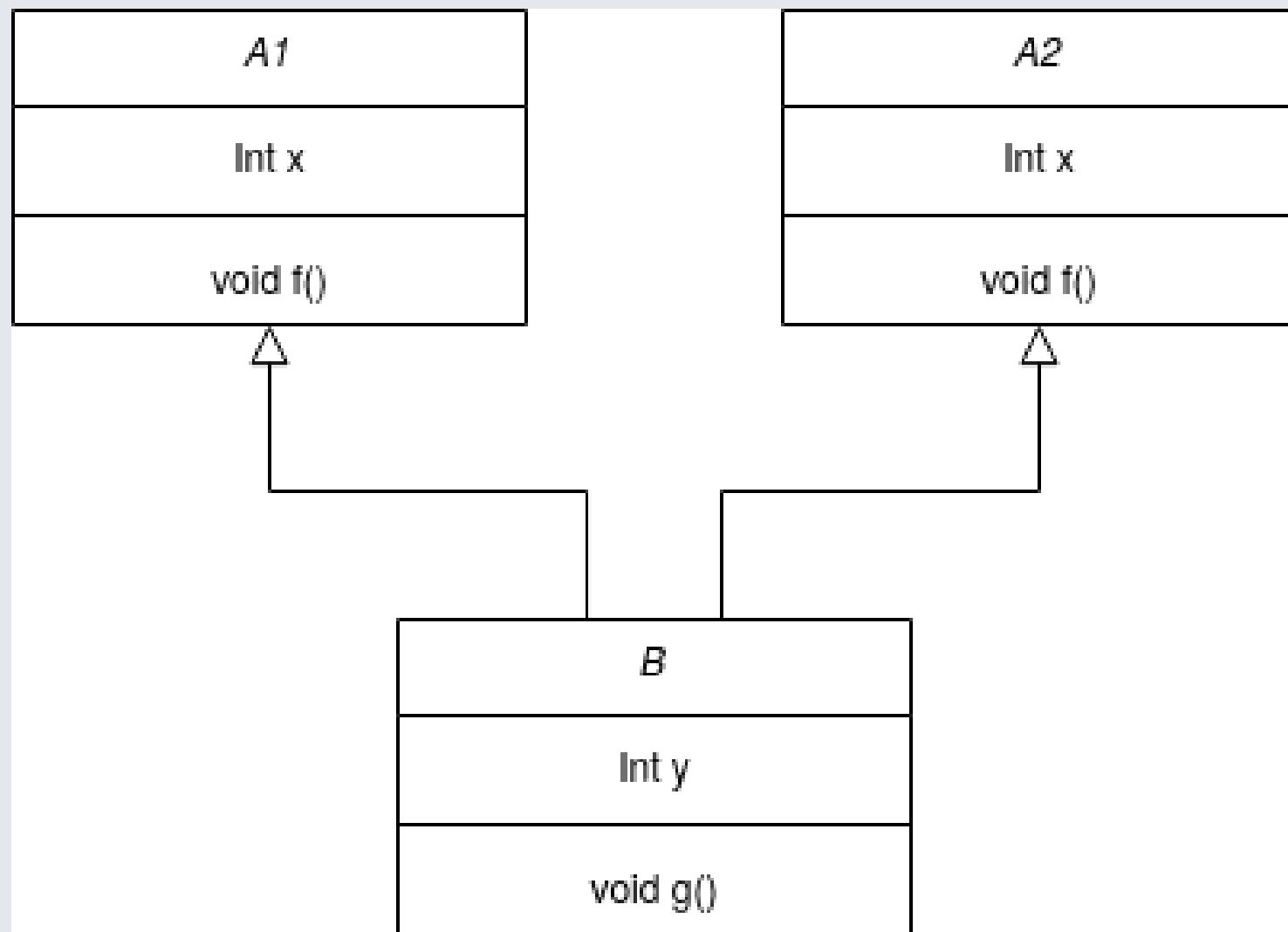
Cas de l'héritage multiple :



mais... d'une certaine façon b "possède toute une partie de type A1, et une partie de type A2"

```
void B::g() {
    // f(); // seul est ambigu
    // x; // seul est ambigu
    A1::f(); A2::f();
    y=A1::x+A2::x;
}
```

Cas de l'héritage multiple :

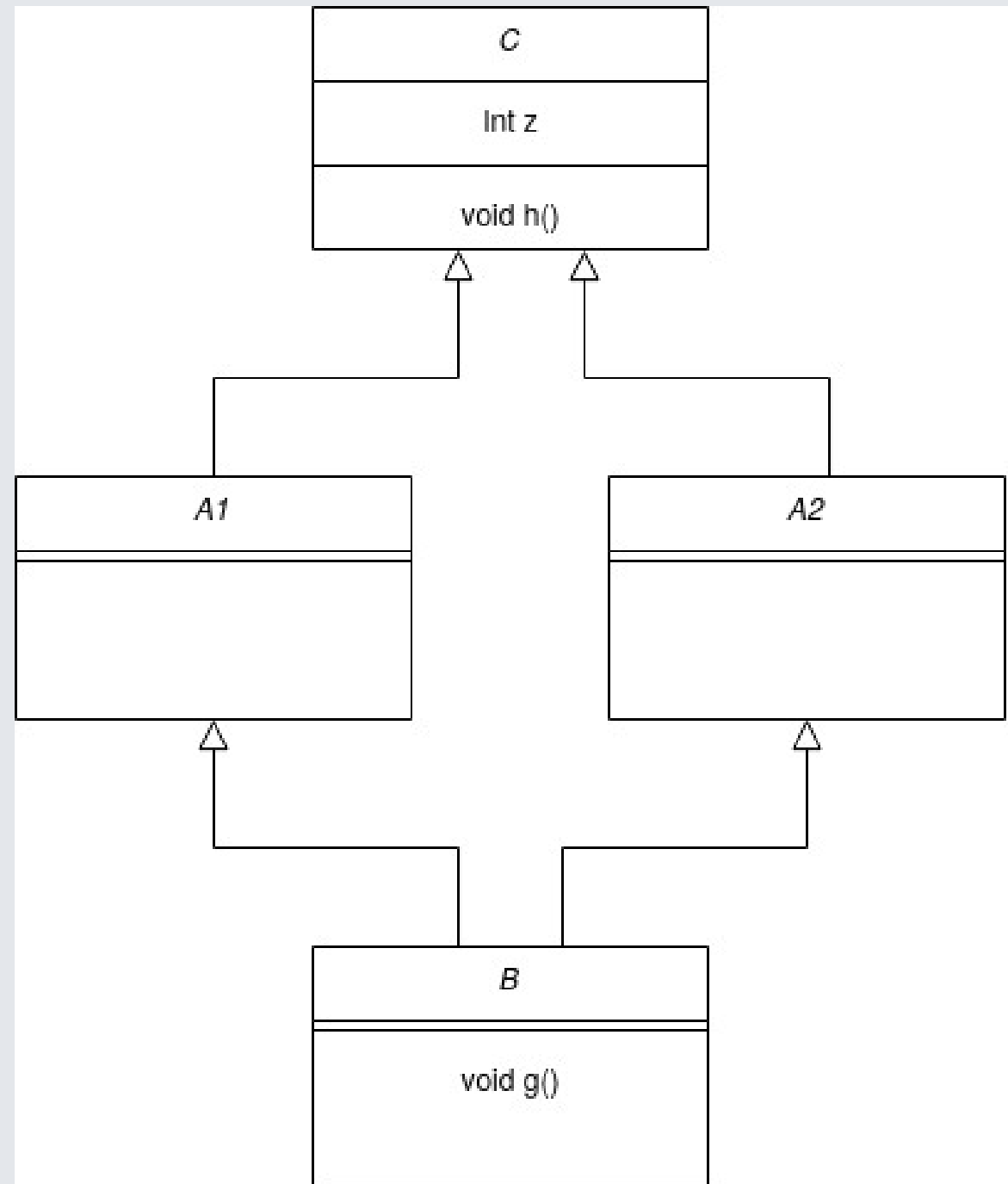


mais... d'une certaine façon b "possède toute une partie de type A1, et une partie de type A2"

```
B::B(int y) :: A1{y+1}, A2{y-1}, y{y} {}
```

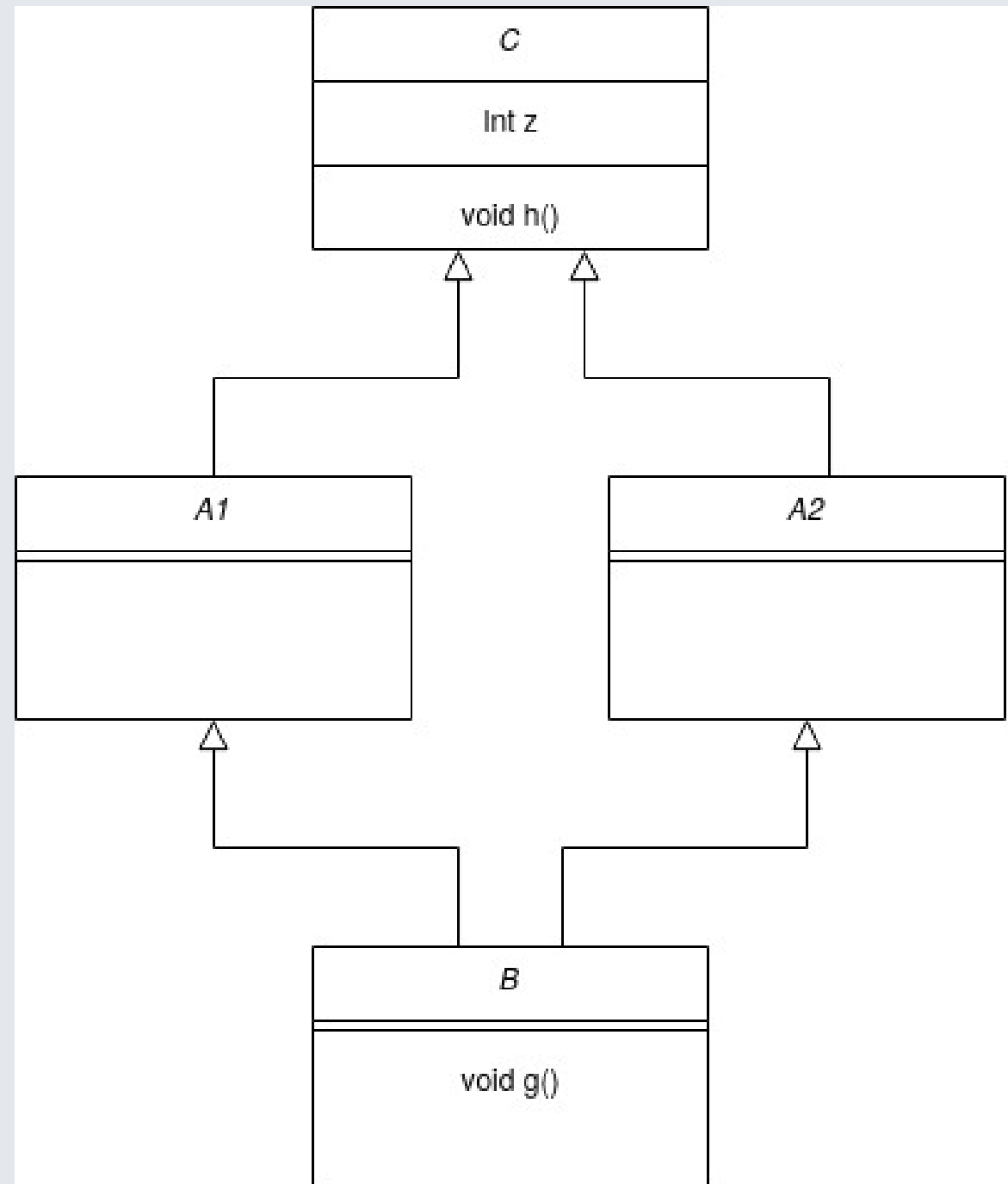
Et il faut les construire tous les deux

Cas de l'héritage multiple ... "en diamant"

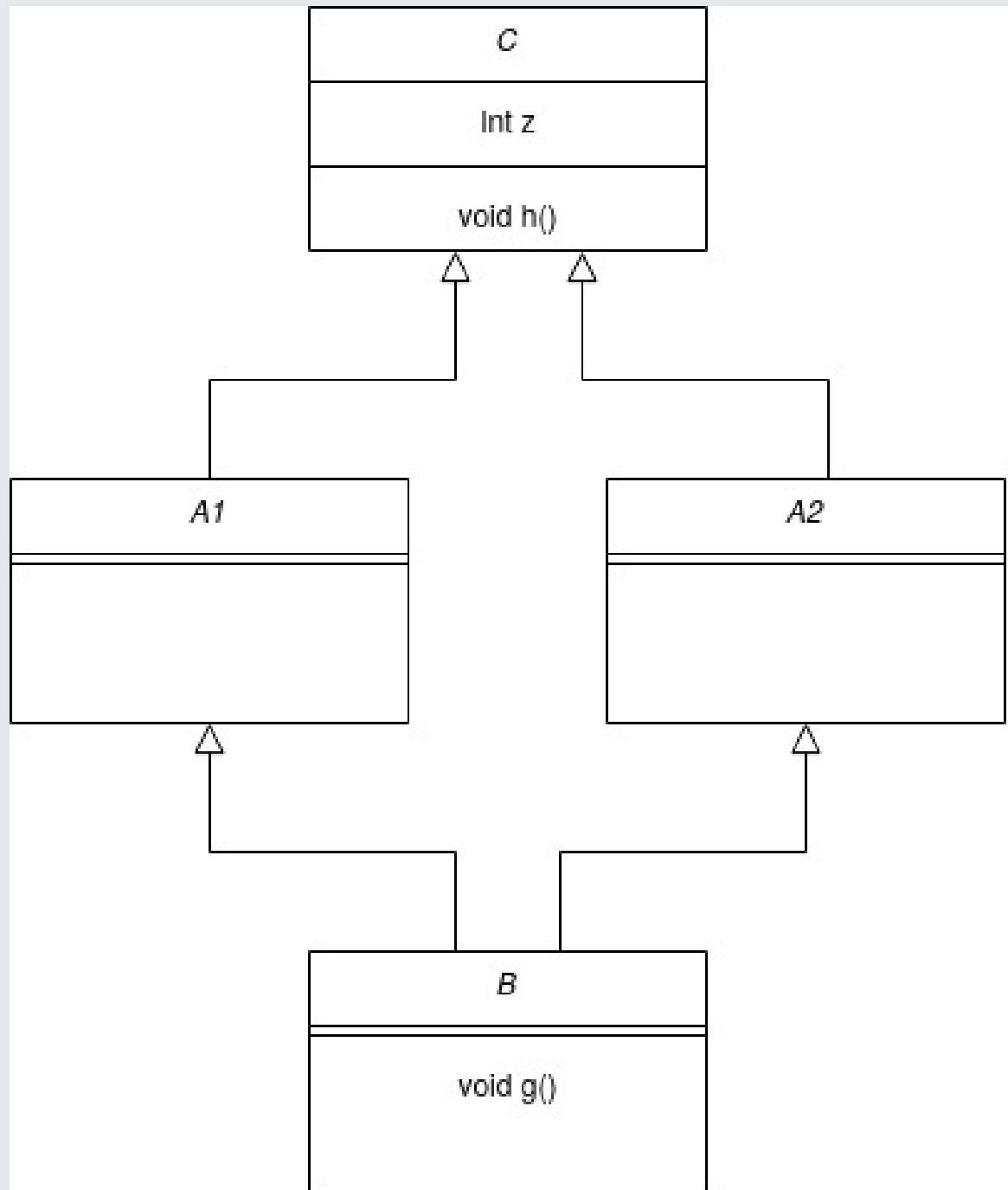


Cas de l'héritage multiple ... "en diamant"

Un B "possède une partie de type A1, et une partie de type A2"



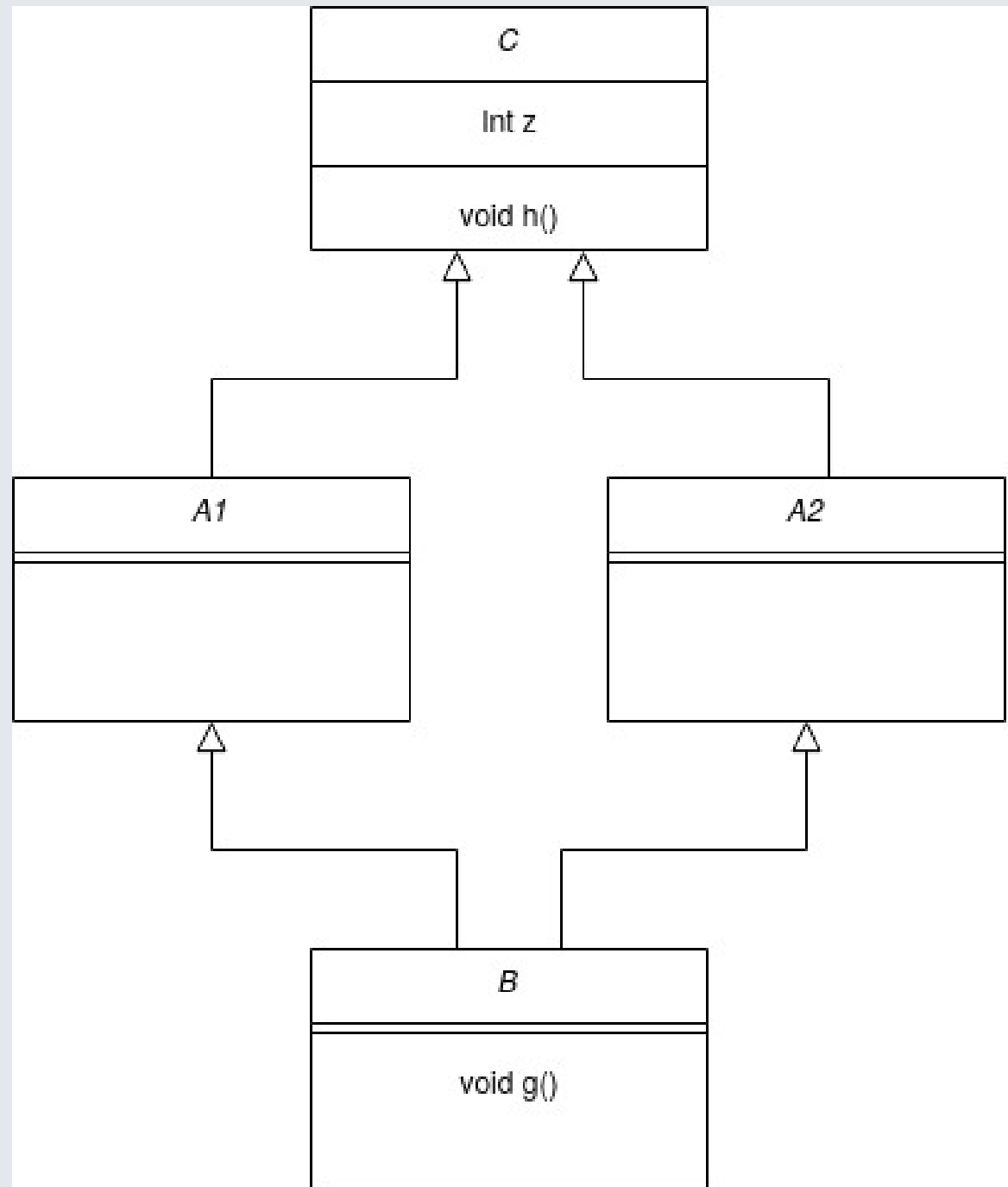
Cas de l'héritage multiple ... "en diamant"



Un B "possède une partie de type A1, et une partie de type A2"

Mais aussi : sa partie de type A1 possède une partie de type C et idem pour A2.

Cas de l'héritage multiple ... "en diamant"



Un B "possède une partie de type A1, et une partie de type A2"

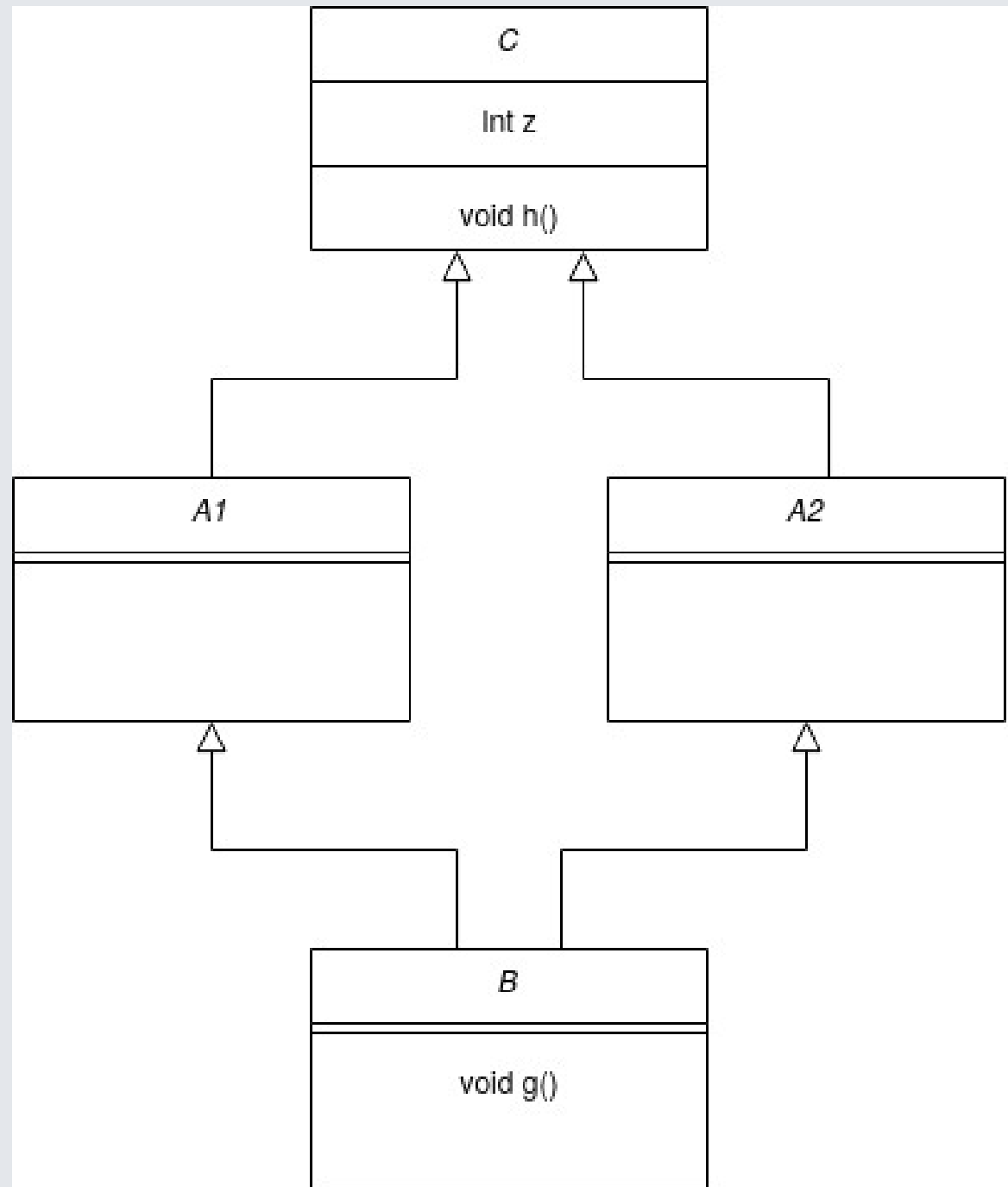
Mais aussi : sa partie de type A1 possède une partie de type C et idem pour A2.

Combien de C pour un B ?

Combien de z pour un B ?

pour un B, `h()` s'adresse à quel C ?

Cas de l'héritage multiple ... "en diamant"



Un B "possède une partie de type A1, et une partie de type A2"

Mais aussi : sa partie

de type A1 ou 2, les deux réponses sont acceptables. Il y aura une distinction syntaxique

Combien de C pour un B ?

Combien de z pour un B ?

pour un B, h() s'adresse à quel C ?

Syntaxe où les ancêtres C sont distingués

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
class A1 : public C {  
public :  
    A1();  
};
```

```
class A2 : public C {  
public :  
    A2();  
};
```

syntaxe
"normale"

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

Syntaxe où les ancêtres C sont distingués

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
class A1 : public C {  
public :  
    A1();  
};
```

```
class A2 : public C {  
public :  
    A2();  
};
```

syntaxe
"normale"

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

Syntaxe où les ancêtres C sont distingués

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1}{}  

```

```
class A1 : public C {  
public :  
    A1();  
};
```

```
class A2 : public C {  
public :  
    A2();  
};
```

syntaxe
"normale"

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

Syntaxe où les ancêtres C sont distingués

<pre>class C { public : int z; void h(); C(int); };</pre>	<pre>C::C(int z):z{z} {} void C::h() { cout << "C h : " << z << endl; }</pre>	
<pre>class A1 : public C { public : A1(); };</pre>	<pre>A1::A1():C{1} {}</pre>	<pre>A2::A2():C{2} {}</pre>
<pre>class B : public A1, public A2 { public : B(); void g(); };</pre>		

```
class A2 : public C {  
public :  
    A2();  
};
```

syntaxe
"normale"

Syntaxe où les ancêtres C sont distingués

<pre>class C { public : int z; void h(); C(int); };</pre>	<pre>C::C(int z):z{z} {} void C::h() { cout << "C h : " << z << endl; }</pre>	
	<pre>A1::A1():C{1}{} </pre>	<pre>A2::A2():C{2}{} </pre>

<pre>class A1 : public C { public : A1(); };</pre>	<pre>class A2 : public C { public : A2(); };</pre>
--	--

syntaxe
"normale"

<pre>class B : public A1, public A2 { public : B(); void g(); };</pre>	<pre>B::B():A1{},A2{} {} void B::g() { cout << A1::z << " et " << A2::z << endl; A1::h(); A2::h(); }</pre>
--	--

ok, c'est juste le
constructeur par
défaut, mais on le
fait apparaître

Syntaxe où les ancêtres C sont distingués

<pre>class C { public : int z; void h(); C(int); };</pre>	<pre>C::C(int z):z{z} {} void C::h() { cout << "C h : " << z << endl; }</pre>	
	<pre>A1::A1():C{1}{} </pre>	<pre>A2::A2():C{2}{} </pre>

<pre>class A1 : public C { public : A1(); };</pre>	<pre>class A2 : public C { public : A2(); };</pre>
--	--

syntaxe
"normale"

<pre>class B : public A1, public A2 { public : B(); void g(); };</pre>	<pre>B::B():A1{},A2{} {} void B::g() { cout << A1::z << " et " << A2::z << endl; A1::h(); A2::h(); }</pre>	<pre>int main() { B b; b.g(); }</pre>
		<pre>1 et 2 C h : 1 C h : 2</pre>

Syntaxe où les ancêtres C sont fusionnés

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B():C{12},A1{},A2{} {}  
void B::g() {  
    cout << A1::z << " et "  
         << A2::z << endl;  
    A1::h();  
    A2::h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

Syntaxe où les ancêtres C sont fusionnés

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B():C{12},A1{},A2{} {}  
void B::g() {  
    cout << A1::z << " et "  
          << A2::z << endl;  
    A1::h();  
    A2::h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

```
12 et 12  
C h : 12  
C h : 12
```

Syntaxe où les ancêtres C sont fusionnés

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B():C{12},A1{},A2{} {}  
void B::g() {  
    cout << z << " et "  
        << z << endl;  
  
    h();  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

```
12 et 12  
C h : 12  
C h : 12
```

Syntaxe où les ancêtres C sont fusionnés

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B():C{12},A1{},A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

```
12  
C h : 12
```

Syntaxe où les ancêtres C sont fusionnés

virtual exprime que la partie relative à C **doit** être construite **au plus près** du type réel

```
void h() {  
    C(int i);  
};  
A1::A1() : C{1} {}  
A2::A2() : C{2} {}  
z << endl;
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};  
B::B() : C{12}, A1{}, A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

```
12  
C h : 12
```

Syntaxe où les ancêtres C sont fusionnés

virtual exprime que la partie relative à C doit être construite au plus près du type réel

c'est toujours le cas dans la classe qu'on déclare

```
void f(C*)  
{  
  C(int i)  
};
```

```
A1::A1() : C{1} {}
```

```
A2::A2() : C{2} {}
```

```
class A1 : virtual public C {  
public :  
  A1();  
};
```

```
class A2 : virtual public C {  
public :  
  A2();  
};
```

```
class B : public A1, public A2 {  
public :  
  B();  
  void g();  
};  
  
B::B() : C{12}, A1 {}, A2 {} {}  
void B::g() {  
  cout << z << endl;  
  h();  
}
```

```
int main() {  
  B b;  
  b.g();  
}
```

```
12  
C h : 12
```

Syntaxe où les ancêtres C sont fusionnés

virtual exprime que la partie relative à C doit être construite au plus près du type réel

```
void f() {  
    C(int i);  
};
```

```
z << endl;
```

```
A1::A1() : C{1} {}
```

```
A2::A2() : C{2} {}
```

```
class A1: virtual public C {  
public :  
    A1();  
};
```

```
class A2: virtual public C {
```

```
public :  
    A1();  
};
```

C'est nouveau ici. L'ambiguïté potentielle sur les ancêtres est levée au plus près du type concerné

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B() : C{12}, A1 {}, A2 {} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

```
12  
C h : 12
```


Syntaxe où les ancêtres C sont fusionnés

virtual exprime que la partie relative à C doit être construite au plus près du type réel

les appels à C{1} et C{2} induits par A1{} et A2{} sont neutralisés

```
void h() {  
  C(int i);  
};
```

```
A1::A1() : C{1} {}
```

```
A2::A2() : C{2} {}
```

```
class A1 : virtual public C {  
public :  
  A1();  
};
```

```
class A2 : virtual public C {  
public :  
  A2();  
};
```

```
class B : public A1, public A2 {  
public :  
  B();  
  void g();  
};  
  
B::B() : C{12}, A1{}, A2{} {}  
void B::g() {  
  cout << z << endl;  
  h();  
}
```

```
int main() {  
  B b;  
  b.g();  
}
```

```
12  
C h : 12
```

Question :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class A2: virtual public C{
```

Si jamais on n'écrit pas le constructeur de C ...

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B(): A1{},A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

Question :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class A2: virtual public C{
```

Si jamais on n'écrit pas le constructeur de C ...
C++ invoque le constructeur C()

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B(): A1{},A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

error: no matching
function for call
to 'C::C()'

Question simple :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B():C{12},A1{},A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
    A1 a;  
    a.h();  
}
```

```
12  
C h : 12  
??
```

Question simple :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

Rien ne change
pour A1, ce qui est
normal : il
construit son
parent virtuel

```
int main() {  
    // B b;  
    // b.g();  
    A1 a;  
    a.h();  
}
```

C h : 1

Question 2 :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class D : public A1 {  
public :  
    // constructeur par défaut  
};
```

```
int main() {  
    D d;  
}
```

??

Question 2 :

```
class C {
public :
    int z;
    void h();
    C(int );
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

$$A1 :: A1 () : C\{1\}\{\}$$

```
class A1: virtual public C{
public :
    A1 ();
};
```

```
class D : public A1 {
public :
    // constructeur par défaut
};
```

Le constructeur par défaut est :

```
D::D() : C{ }, A1{ }{ }
```

```
int main() {
    D d;
}
```

```
error: no matching
function for call to
'C::C()'
```

Question 2 :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class D : public A1 {  
public :  
    D();  
};
```

```
D::D():C{3}, A1{} {}
```

```
int main() {  
    D d;  
    d.h();  
}
```

```
C h : 3
```


Question 3 :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: public C{  
public : Branche non virtual  
    A1();  
};
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B(): A1{}, A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

?

Question 3 :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: public C{  
public :  
    A1();  
};
```

Branche non virtual
en premier héritage

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B(): A1{}, A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

?

Question 3 :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: public C{  
public :  
    A1();  
};
```

Branche non virtual
en premier héritage
C est-il construit ?

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B(): A1{}, A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

?

Question 3 :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: public C{  
public :  
    A1();  
};
```

"la partie relative à C
doit être construite au
plus près du type réel"

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B(): A1{}, A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

```
error: no matching  
function for call  
to 'C::C()'
```

Question 3 :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: public C{  
public :  
    A1();  
};
```

"la partie relative à C
doit être construite au
plus près du type réel"

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B():C{12},A1{},A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

??

Question 3 :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: public C{  
public :  
    A1();  
};
```

"la partie relative à C
doit être construite au
plus près du type réel"

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B():C{12},A1{},A2{} {}  
void B::g() {  
    cout << z << endl;  
    h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

error: reference to
'z' is ambiguous

Question 3 :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: public C{  
public :  
    A1();  
};
```

"la partie relative à C
doit être construite au
plus près du type réel"

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B():C{12},A1{},A2{} {}  
void B::g() {  
    cout << A1::z << " et "  
         << A2::z << endl;  
    A1::h();  
    A2::h();  
}
```

```
int main() {  
    B b;  
    b.g();  
}
```

```
1 et 12  
C h : 1  
C h : 12
```

Question 3 :

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
C::C(int z):z{z} {}  
void C::h() {  
    cout << "C h : " << z << endl;  
}
```

```
A1::A1():C{1} {}
```

```
A2::A2():C{2} {}
```

```
class A1: public C{  
public :  
    A1();  
};
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
B::B():C{12},A1{},A2{} {}  
void B::g() {  
    cout << A1::z << " et "  
         << A2::z << endl;  
    A1::h();  
    A2::h();  
}
```

A1 construit son "bloc",
A2 est construit en 2 fois

```
1 et 12  
C h : 1  
C h : 12
```


Question 4 - au delà de B ?

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
class A1: virtual public C{  
public :  
    A1();  
};
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class D: public B{  
public :  
    D();  
};
```

```
D::D():B{}{}
```

```
int main() {  
    D d;  
}
```

?

Question 4 - au delà de B ?

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
class A1: virtual public C{  
public :  
    A1 "la partie relative à C  
    doit être construite au  
    plus près du type réel"
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class D: public B{  
public :  
    D();  
};
```

```
D::D():B{}{}
```

```
int main() {  
    D d;  
}
```

```
error: no matching  
function for call  
to 'C::C()'
```

Question 4 - au delà de B ?

```
class C {  
public :  
    int z;  
    void h();  
    C(int );  
};
```

```
class B : public A1, public A2 {  
public :  
    B();  
    void g();  
};
```

```
class A1: virtual public C{  
public :  
    A1 "la partie relative à C  
    doit être construite au  
    plus près du type réel"
```

```
class A2: virtual public C{  
public :  
    A2();  
};
```

```
class D: public B{  
public :  
    D();  
};
```

```
D::D():C{3},B{}{}
```

```
int main() {  
    D d;  
}
```

ok

Remarques sur les choix de conceptions faits en java :

```
// code en java  
public class B extends A implements I1,I2 {}
```

Remarque sur les héritages faits en java

un seul "héritage" entre classes

"héritages" qui concernent les interfaces

```
// code en java  
public class B extends A implements I1, I2 {}
```

Remarques et choix de syntaxe

un seul "héritage" entre classes

"héritage" qui concernent les interfaces

```
// code en java  
public class B extends A implements I1, I2 {}
```

```
public interface I1 {  
    void f();  
    void g();  
}
```

les interfaces en java ne sont que déclaratives. Elles ne contiennent pas d'attribut d'instance etc ...

L'idée derrière ces simplifications syntaxique est de permettre un héritage multiple simple, en évitant les questions que nous venons d'aborder puisque :

- pas d'ambiguïtés car les méthodes ne sont pas implémentées
- pas de pb pour dupliquer ou pas une classe de base (elle n'a pas de corps)

En c++ on écrirait la même chose ainsi :

```
class B : public A, public I1, public I2 {}
```

```
class I1 {  
    virtual void f()=0;  
    virtual void g()=0;  
}
```

les interfaces de java sont,
en c++, une forme de classe
abstraite "pure" :

- sans attributs,
- sans méthodes définies

En c++ on écrirait la même chose ainsi :

```
class B : public A, public I1, public I2 {}
```

```
class I1 {  
    virtual void f()=0;  
    virtual void g()=0;  
}
```

les interfaces de java sont,
en c++, une forme de classe
abstraite "pure" :

- sans attributs,
- sans méthodes définies

Tant qu'il y a un constructeur par défaut pour I1,
c++ construira un B sans lourdeurs syntaxiques.

(si I1 contient des attributs, la question de
l'héritage virtuel ou pas de I1 se posera)

Rq : attention aux adresses !

```
class Classe1 {};  
class Classe2 {};
```

```
class T : public Classe1, public Classe2 {};
```

```
int main() {  
    T t;  
    Classe1 *p1;  
    Classe2 *p2;  
  
    p1 = &t;  
    p2 = &t;  
  
    cout << "t " << (&t) << endl;  
    cout << "p1 " << p1 << endl;  
    cout << "p2 " << p2 << endl;  
  
    p1 = (T *) (p2);  
    cout << "p1 (from p2) " << p1 << endl;  
}
```

Rq : attention aux adresses !

```
class Classe1 {};  
class Classe2 {};
```

```
class T : public Classe1, public Classe2 {};
```

```
int main() {  
    T t;  
    Classe1 *p1;  
    Classe2 *p2;
```

```
    p1 = &t;  
    p2 = &t;
```

```
    cout << "t " << (&t) << endl;  
    cout << "p1 " << p1 << endl;  
    cout << "p2 " << p2 << endl;
```

```
    p1 = (T *) (p2);
```

```
    cout << "p1 (from p2) " << p1 << endl;
```

```
}
```

ici aucun pb

```
t 0xbff40dc3  
p1 0xbff40dc3  
p2 0xbff40dc3  
p1 (from p2) 0xbff40dc3
```

Rq : attention aux adresses !

```
class Classe1 : public virtual A{};
class Classe2 : public virtual A{};
```

```
class A {};
```

```
class T : public Classe1, public Classe2 {};
```

```
int main() {
    T t;
    A *pA;
    Classe1 *p1;
    Classe2 *p2;
    p1 = &t;
    pA = &t;
    p2 = &t;
    cout << "t " << (&t) << endl;
    cout << "p1 " << p1 << endl;
    cout << "pA " << pA << endl;
    cout << "p2 " << p2 << endl;
}
```

```
t 0xbfba75e4
p1 0xbfba75e4
pA 0xbfba75e4
p2 0xbfba75e8
```

Rq : attention aux adresses !

```
class Classe1 : public virtual A{};
class Classe2 : public virtual A{};
```

```
class A {};
```

```
class T : public Classe1, public Classe2 {};
```

```
T t;
A *pA;
Classe1 *p1;
Classe2 *p2;
p1 = &t;
pA = &t;
p2 = &t;
cout << "t " << (&t) << endl;
cout << "p1 " << p1 << endl;
cout << "pa " << pA << endl;
cout << "p2 " << p2 << endl;
pA = p2;
p1 = (Classe1*) p2;
cout << "pA (from p2) " << pA << endl;
cout << "p1 (from p2) " << p1 << endl;
```

```
t 0xbfba75e4
p1 0xbfba75e4
pA 0xbfba75e4
p2 0xbfba75e8
??
```

Rq : attention aux adresses !

```
class Classe1 : public virtual A{};  
class Classe2 : public virtual A{};
```

```
class A {};
```

```
class T : public Classe1, public Classe2 {};
```

```
T t;  
A *pA;  
Classe1 *p1;  
Classe2 *p2;  
p1 = &t;  
pA = &t;  
p2 = &t;  
cout << "t " << (&t) << endl;
```

```
cout << "p1 " << p1 << endl;
```

```
cout << "pa " << pA << endl;
```

```
cout << "p2 " << p2 << endl;
```

```
pA = p2;
```

```
p1 = (Classe1*) p2;
```

```
cout << "pA (from p2) " << pA
```

```
cout << "p1 (from p2) " << p1 << endl;
```

```
t 0xbfba75e4
```

```
p1 0xbfba75e4
```

```
pA 0xbfba75e4
```

```
p2 0xbfba75e8
```

```
pA (from p2) 0xbfba75e4
```

```
p1 (from p2) 0xbfba75e8
```

Rq : attention aux adresses !

```
class Classe1 : public virtual A{};  
class Classe2 : public virtual A{};
```

```
class A {};
```

```
class T : public Classe1, public Classe2 {};
```

```
T t;  
A *pA;  
Classe1 *p1;  
Classe2 *p2;  
p1 = &t;  
pA = &t;  
p2 = &t;  
pA = p2;  
p1 = (Classe1*) p2;  
if (pA==p1) cout << "egaux !" << endl;  
    else cout << "différents !" << endl;
```

```
t    0xbfba75e4  
p1   0xbfba75e4  
pA   0xbfba75e4  
p2   0xbfba75e8  
pA   (from p2) 0xbfba75e4  
p1   (from p2) 0xbfba75e8  
??
```

Rq : attention aux adresses !

```
class Classe1 : public virtual A{};  
class Classe2 : public virtual A{};
```

```
class A {};
```

```
class T : public Classe1, public Classe2 {};
```

```
T t;  
A *pA;  
Classe1 *p1;  
Classe2 *p2;  
p1 = &t;  
pA = &t;  
p2 = &t;  
pA = p2;  
p1 = (Classe1*) p2;
```

```
if (pA==p1) cout << "egaux !" << endl;  
else cout << "different";
```

```
t  0xbfba75e4  
p1 0xbfba75e4  
pA 0xbfba75e4  
p2 0xbfba75e8  
pA (from p2) 0xbfba75e4  
p1 (from p2) 0xbfba75e8  
egaux !
```

Heureusement

Mais à l'affichage vous pouvez être troublés :
C++ distingue (un peu) les pointeurs et les adresses des
objets, mais cela devrait rester transparent

L'explication est que dans T il y a deux parties proches qui représentent Classe1, et Classe2.

Dans le premier exemple, ces classes ayant une taille nulle, une optimisation est faite pour ne pas occuper d'espace supplémentaire.

Dans le second, la table virtuelle occupe un espace, et ces 2 adresses ne sont pas "strictement" les mêmes.

Ca aurait donc aussi été le cas si on modifiait l'exemple 1 pour ajouter un attribut (int) par ex.

Si ca vous intéresse, voir : <https://h-deb.clg.qc.ca/Sujets/TrucsScouts/Adresse-precise-objet.html>

Correction du TP noté

Correction du TP noté

Tâche : un élément atomique à réaliser au sein d'un projet. Elle possède un **nom**, et vous proposerez une façon de les **numéroter automatiquement** de sorte qu'on puisse **l'identifier par ce numéro**. Une tâche peut être **réalisée ou en attente**. Elle ne peut évoluer que dans un sens : il n'est pas possible de dé-réaliser une tâche. Une durée est prévue pour sa réalisation, on parle ici de sa **durée propre**. La tâche ne pourra être mise en oeuvre que si ses dépendances sont elles même réalisées. Ce qu'on appelle dépendances, ce sont d'autres tâches qui lui sont prioritaires. Une tâche a une **vision locale de ses dépendances**, vous utiliserez un vector pour la caractériser. Elle peut être incomplète : par exemple, si t1 dépend de t2 qui dépend de t3, la dépendance de t1 à t3 peut apparaître ou pas dans la vision locale qu'a t1.

Correction du TP noté

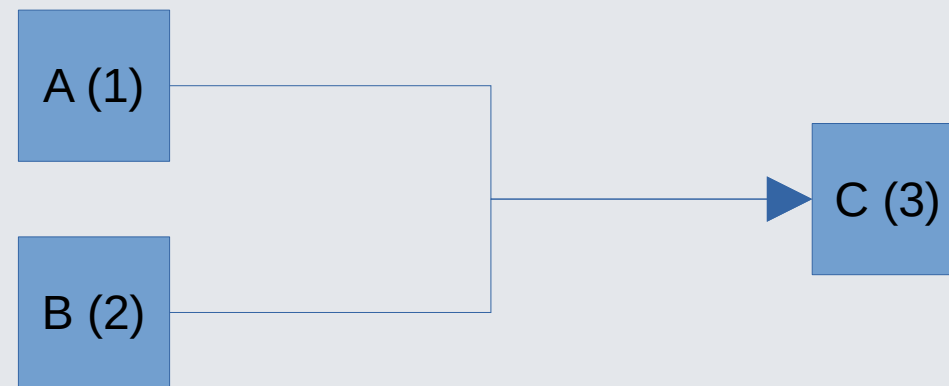
```
class Tache {  
private :  
    static int NB;  
    bool faite; // politique de chgt particulière  
    vector <Tache *> depend_de;  
  
public :  
    const string name;  
    const int num;  
    const int duree;  
  
    Tache(const string n, const int d=0);  
    virtual ~Tache();  
  
    ...  
};
```

Correction du TP noté

Pour chaque classe vous devrez pouvoir :

- avoir une gestion satisfaisante des copies/affectation/destruction.

Imaginons :



La destruction de A ne peut pas impacter C : on ne se préoccupera pas de restituer (delete) les dépendances.

La copie, l'affectation contrediraient l'identification par num : on va les interdire

```
class Tache {  
    const Tache & operator=(const Tache &) = delete;  
    Tache (const Tache &)=delete;  
private :  
    static int NB;  
    bool faite;  
    vector <Tache *> depend_de;  
  
public :  
    const string name;  
    const int num;  
    const int duree;  
  
    Tache(const string n, const int d=0);  
    virtual ~Tache();  
};
```

```
#include "Tache.hpp"
```

```
Tache::~~Tache() {}
```

```
// Projet sera responsable de la création/destruction
```

Correction du TP noté

Pour chaque classe vous devrez pouvoir :

- en afficher une instance en surchargeant l'opérateur <<

```
class Tache {
    const Tache & operator=(const Tache &) = delete;
    Tache (const Tache &)=delete;
private :
    static int NB;
    bool faite;
    vector <Tache *> depend_de;

public :
    const string name;
    const int num;
    const int duree;

    Tache(const string n, const int d=0);
    virtual ~Tache();

friend ostream & operator<<(ostream &, const Tache &);
};
```



```
ostream & operator<<(ostream & o, const Tache & x) {
    o << x.name << " (" << x.num << ")" << endl;
    o << (x.fait?" FAITE ":" NON FAITE");
    o << " duree : " << x.duree << endl;
    if (!x.depend_de.empty()) {
        o << "dépend de : " ;
        for (Tache *t : x.depend_de) o << t->num << " ";
        o << endl;
    }
    return o;
}
```

On choisi d'afficher des éléments privés.

Plutot que d'utiliser des getters disponibles pour tout le monde,
on préfère que operator<< ait le privilège "friend"

Pour le constructeur, il faut utiliser la séquence d'initialisation !
D'autant plus que des attributs sont "const"

```
#include "Tache.hpp"

int Tache::NB=0;

Tache::Tache(const string n,const int d)
    : faite{false},name{n}, num{NB++},duree{d} {}
```

remarquez qu'on n'a pas besoin de préciser qq chose pour le vecteur de dépendance : son constructeur par défaut convient, et on n'a rien à dire de plus d'utile.

En particulier il vous faudra écrire :

- **bool realise()** : déclenche la réalisation d'une tâche après avoir vérifié que c'est possible.
- **bool depends_from(const Tache & x)** : la tâche courante dépend-elle transitivement de x
- **bool ajouteDependance(Tache & x)** : ajoute la dépendance de this à x si celle-ci ne crée pas de cycle
- **int dureeParal()** : durée totale de réalisation d'une tâche et de toutes ses dépendances (au sens large) non encore réalisées.
(avec un degré de parallélisme arbitraire)

```
class Tache {  
    ...  
public :  
    bool realise();  
    bool depends_from(const Tache &x) const;  
    bool ajouteDependance(Tache &);  
    int dureeParal() const;  
    ...  
};
```

```
bool Tache::realise() {  
    for (Tache *t : depend_de)  
        if (!(t->faite)) return false;  
    faite=true;  
    return true;  
}
```

```
bool Tache::realise() {  
    for (Tache *t : depend_de)  
        if (!(t->faite)) return false;  
    faite=true;  
    return true;  
}
```

```
bool Tache::depends_from(const Tache &x) const {  
    if (this==&x) return true;  
    for (Tache *t :depend_de)  
        if (t==&x || t->depends_from(x)) return true;  
    return false;  
}
```

```
bool Tache::ajouteDependance( Tache &x) {  
    if (x.depends_from(*this)) return false; // cycle  
    if (depends_from(x)) return false; // déjà là  
    depend_de.push_back( &x);  
    return true;  
}
```

```
int Tache::dureeParal() const {  
    if (faite) return 0;  
    int d=0;  
    for (const Tache * t:depend_de)  
        d=max(d,t->dureeParal());  
    return d+duree;  
}
```

- Vous aurez besoin de **marquer** les sommets (les tâches) lors d'un parcours : ajoutez leur un attribut public booléen.
- écrivez une méthode de parcours en profondeur **void Tache::PP_postfixe** qui prenne en argument une **référence d'un vecteur de Tache**. Vous lui ajouterez une tâche au moment **postfixe** de son parcours.


```
class Tache {  
    ...  
  
public :  
    bool marquage;  
    void PP_postfixe (vector<Tache*> &v) ;  
  
    ...  
};
```

```
void Tache::PP_postfixe (vector<Tache*> &v) {  
    marquage=true;  
    for (Tache * t :depend_de)  
        if (! (t->marquage)) t->PP_postfixe (v) ;  
    v.push_back (this) ;  
}
```

Classe Projet :

conserve un **vector de l'ensemble de ses tâches**.

Nous **ferons en sorte** qu'il respecte un ordre topologique.

Fourni des méthodes utiles à ses sous-classes **uniquement** :

- **pick_2()** : retourne 2 d'id de tâches au hasard, telles que la 2ème ne dépende pas transitivement de la 1ère.
- **contains** : retourne un pointeur vers une tâche désignée par son identifiant, ou par son nom si possible ou nullptr sinon
- un **affichage** : parcourt le vecteur de tâches.
- **consult_tasks()** : une version non modifiable des tâches
- **topologicalSort()** : réordonne le vector de tâches

```
class Projet {  
protected :  
    vector <Tache *> all_tasks;  
    Projet();  
    pair<int, int> pick_two_random_task();  
    Tache * contains(int);  
    Tache * contains(string);  
    friend ostream & operator<<(ostream & , const  
Projet &);  
    virtual ~Projet();  
public :  
    vector <Tache const *> consult_Tache() const;  
};
```

Pour chaque classe vous devrez pouvoir :

- avoir une gestion satisfaisante des copies/affectation/destruction

On comprend (un peu plus loin) que ProtoProjet vont se charger de la création des tâches, et transférer cette charge aux RunProjet.

-> Projet (en général) est responsable des tâches.

autoriser affectations reviendrait à en partager la responsabilité

-> on interdira l'affectation ou on en fera une move affectation

Même raisonnement avec la copie

Des sémantiques "move" pourraient permettre de reprendre les tâches d'un RunProject dans un Proto, refaire des ajouts et revenir au Run, ce qui est plutôt contraire à l'esprit du sujet.

```
class Projet {  
    ...  
    const Projet & operator=(const Projet &) = delete;  
    Projet ( const Projet &)= delete;  
    ...  
};
```

```
class Projet {  
    ...  
    const Projet & operator=(const Projet &) = delete;  
    Projet ( const Projet &)= delete;  
    ...  
};
```

```
Projet::~~Projet() {  
    for (Tache * t:all_tasks) delete t;  
}
```

```
class Projet {  
    ...  
    const Projet & operator=(const Projet &) = delete;  
    Projet ( const Projet &)= delete;  
    ...  
};
```

```
Projet::~~Projet() {  
    for (Tache * t:all_tasks) delete t;  
}
```

```
ostream & operator<<(ostream & o, const Projet &p) {  
    for ( Tache * t: p.all_tasks) o << *t ;  
    return o;  
}
```



```
class Projet {  
    ...  
    const Projet & operator=(const Projet &) = delete;  
    Projet ( const Projet &)= delete;  
    ...  
};
```

```
Projet::~~Projet() {  
    for (Tache * t:all_tasks) delete t;  
}
```

```
ostream & operator<<(ostream & o, const Projet &p) {  
    for ( Tache * t: p.all_tasks) o << *t ;  
    return o;  
}
```

```
Tache* Projet::contains(string n) {  
    for (Tache * t:all_tasks) if (t->name==n) return t;  
    return nullptr;  
}
```

```
pair<int, int> Projet::pick_2() {  
    int t1,t2;  
    do {  
        t1=rand()%all_tasks.size();  
        t2=rand()%all_tasks.size();  
    } while(  
        (t1==t2) ||  
        (all_tasks[t2]->depends_from(*all_tasks[t1]))  
    );  
    return {all_tasks[t1]->num, all_tasks[t2]->num};  
}
```

Remarquez :

return {a,b} + typage du retour fait une construction

```
vector<Tache const*> Projet::consult_Tache() const {  
    vector<Tache const*> rep;  
    for (Tache *t : all_tasks) rep.push_back(t);  
    return rep;  
}
```

Remarquez que `return all_tasks;` ne convient pas :

car si B est un sous type de A il n'y a pas de garantie de conversion d'une collection de A vers une collection de B.

`Tache * const` est un sous type de `Tache *`

par contre lors du `push_back` la conversion individuelle est invoquée

Classe ProtoProjet :

est une classe qui sert uniquement à construire un projet.
Elle ne peut pas faire progresser la réalisation d'un projet.

Pour chaque classe vous devrez pouvoir :

- avoir une gestion satisfaisante des copies/affectation/destruction*
- prévoir un affichage*

Elle hérite de Projet :

- l'affichage de projet est suffisant
- la destruction de projet est suffisante
- copie/affectation sont interdites dans Projet, celles par défaut font appel à celles de Projet ... elles sont donc déjà interdites

-> rien à écrire !

```
class ProtoProjet : public Projet {  
  
public:  
    ProtoProjet();  
    bool ajoute(string n, int d);  
    bool ajoute(string n, int d, int t);  
    bool ajoute(string n, int d, int t1, int t2);  
private :  
    void cleanMarks();  
    void topologicalSort();  
    friend class RunProjet;  
};
```

Rq 1 :
comme l'ajout de taches peut créer de nouvelles dépendances,
l'ordre topologique peut être à reconstruire.

C'est pourquoi on les méthodes sont ici. (Dans Projet ok aussi)

Rq 2 :

nous reviendrons sur le friend class au moment de RunProjet

```
ProtoProjet::ProtoProjet() {  
    all_tasks.push_back(new Tache("Fin"));  
    all_tasks.push_back(new Tache("Debut"));  
    all_tasks[0]->ajouteDependance(*all_tasks[1]);  
}
```

```
bool ProtoProjet::ajoute (string n, int d) {  
    pair<int, int> p =pick_2();  
    return ajoute (n,d,p.first,p.second);  
}
```

```
bool ProtoProjet::ajoute(string n, int d, int t) {  
    return ajoute(n,d,all_tasks[0]->num, t);  
}
```

```
bool ProtoProjet::ajoute(string n,int d,int t1,int t2)  
{  
    ... est la plus générale : réutilisez !  
}
```

```
bool ProtoProjet::ajoute( string n, int d,
                          int t1, int t2) {
    Tache * task1= contains(t1), *task2 =contains(t2);
    if ( contains(n)!=nullptr || !task1 || !task2
        || task2->depends_from(*task1)
        ) return false;
    Tache *t=new Tache(n,d);
    // mise à jour des dépendance
    task1->ajouteDependance(*t);
    t->ajouteDependance(*task2);
    all_tasks.push_back(t); // invalide le tri
    topologicalSort();
    return true;
}
```

```
void ProtoProjet::cleanMarks() {  
    for (Tache * t : all_tasks) t->marquage=false;  
}
```

```
void ProtoProjet::topologicalSort() {  
    cleanMarks();  
    vector<Tache*> tmp;  
  
    all_tasks[0]->PP_postfixe(tmp);  
    all_tasks.clear();  
    for (int i=tmp.size()-1; i>=0; i--)  
        all_tasks.push_back(tmp[i]);  
}
```


Classe RunProjet :

Pour chaque classe vous devrez pouvoir :

- avoir une gestion satisfaisante des copies/affectation/destruction*
- prévoir un affichage*

Elle hérite de Projet qui nous convient -> rien à écrire !

```
class RunProjet : public Projet{  
  
public:  
    RunProjet (ProtoProjet &);  
    bool run (int );  
    bool run (vector<int>);  
};
```

remarquez l'absence de const devant ProtoProjet & : il sera "vidé de sa substance"

```
RunProjet::RunProjet (ProtoProjet &x) : Projet{} {  
    all_tasks = x.all_tasks;  
    x.all_tasks.clear();  
}
```

Rq :

- la construction de Projet{} est facultative (implicite sinon)
- l'affectation entre vecteurs copie les pointeurs

Rq plus importante :

on a besoin d'accéder/modifier x.all_tasks.

Bien qu'il soit protected dans Projet, les classes RunProjet et ProtoProjet sont dans des branches différentes et n'ont donc pas accès à ce champs.

C'est pour cela que dans ProtoProjet on avait prévu la déclaration
friend class RunProjet

```
bool RunProjet::run(int n) {  
    Tache * t {contains(n)};  
    return (t != nullptr) && (t->realise());  
}
```

```
bool RunProjet::run (vector<int> v) {  
    for (int x:v)  
        if (!run (x)) return false;  
    return true;  
}
```

Rq :certains se sont préoccupé de pouvoir faire un "rollback" dans le second run, au cas où l'une des tâches échouait.

C'était une bonne idée, mais le cadre n'était pas assez détaillé pour le faire facilement.

Classes Gestionnaire, Consultant, Expert :

On voulait voir une méthode virtuelle pure dans Gestionnaire

```
class Gestionnaire {  
public:  
    virtual pair<vector<int>, int>  
        avis(const RunProjet &p)=0;  
};
```

```
class Consultant : public Gestionnaire {  
public:  
    virtual pair<vector<int>, int>  
        avis(const RunProjet &p);  
};
```

```
class Expert : public Gestionnaire {  
public:  
    virtual pair<vector<int>, int>  
        avis(const RunProjet &p);  
};
```

```
pair<vector<int>, int>
    Consultant::avis(const RunProjet &p) {
    // on parcours les tâches à partir de la fin
    // on ne compte que les taches non réalisées
    int duree=0;
    vector<int> rep;
    vector<const Tache *> tasks {p.consult_Tache()};
    for (int i=tasks.size()-1; i>=0 ; i--) {
        const Tache & t {*(tasks[i])};
        if (!(t.isDone())) {
            rep.push_back(t.num);
            duree += t.duree;
        }
    }
    return {rep, duree};
}
```

```
pair<vector<int>, int>
    Expert::avis(const RunProjet &p) {
    // j'ai utilisé un consultant pour récupérer la
    trace qui est la même ...
    Consultant c;
    vector<int> rep {c.avis(p).first};

    vector <Tache const *> tasks {p.consult_Tache()};
    int duree= tasks[0]->dureeParal();
    return {rep,duree};
}
```