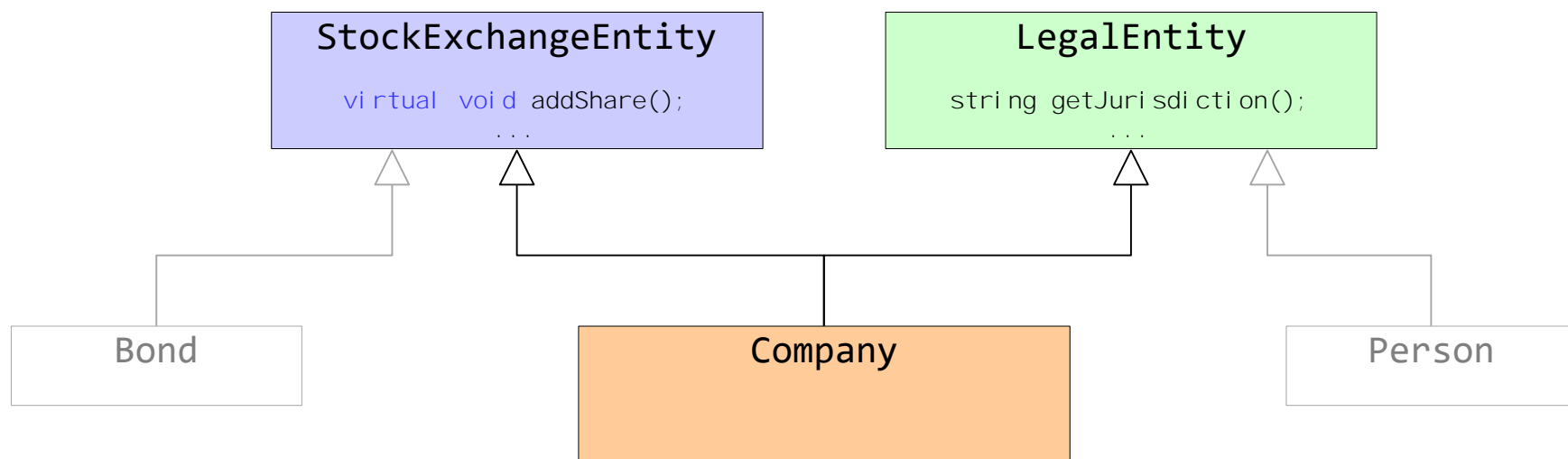


Héritage multiple

Et polymorphisme, ordre de construction,
ambiguïté de nom, problème du diamant,
attention, type vs interface

Motivation

- Une compagnie peut être partagée en actions pouvant être en bourse (« stock exchange »)
- et est aussi une entité juridique (qui peut être menée en procès, ...) comme une personne
- Des obligations ou une personne humaine ne sont que l'un ou l'autre, donc ce sont des choses indépendantes



Héritage multiple en C++

- En C++, on peut faire dériver une classe de plus d'une classe:

```
class Company
    : public StockExchangeEnti ty,
      public Legal Enti ty
{
    /* ... */
};
```

Héritage multiple et polymorphisme

- Le polymorphisme fonctionne de la même manière avec l'héritage multiple
- La différence avec l'héritage simple est qu'un **même objet peut être pointé (ou référé) par deux pointeurs (ou références) de deux classes de base différentes**

Héritage multiple et polymorphisme (exemple)

```
int main() {  
    vector<StockExchangeEntity*> stock;  
    vector<LegalEntity*> legal;  
    /* ... */  
    Company* c = new Company(/* ... */);  
    /* ... */  
    stock.push_back(c);  
    legal.push_back(c);  
    /* ... */  
}
```

Le même objet mis dans deux vecteurs de types différents. (Les deux types sont parents mais n'ont pas d'héritage entre eux.)

```
int nbShares = dynamic_cast<StockExchangeEntity*>  
    (legal[0]) -> getNbShares();
```

Peut `dynamic_cast` entre les bases.

Ne peut pas `static_cast` entre les bases (erreur compilation).

Héritage multiple et polymorphisme (exemple)

Polymorphisme par héritage: fonctions compilées une seule fois.

```
bool hasShares(const StockExchangeEnti ty& s) {
    return (s.getNbShares() > 0);
}
bool isAtMontreal (const Legal Enti ty& l) {
    return (l.getJuri sdi cti on() == "Montreal ");
}
int mai n() {
    Company c(/* ... */);
    if (hasShares(c)) {
        /* ... */
    }
    if (isAtMontreal (c)) {
        /* ... */
    }
    /* ... */
}
```

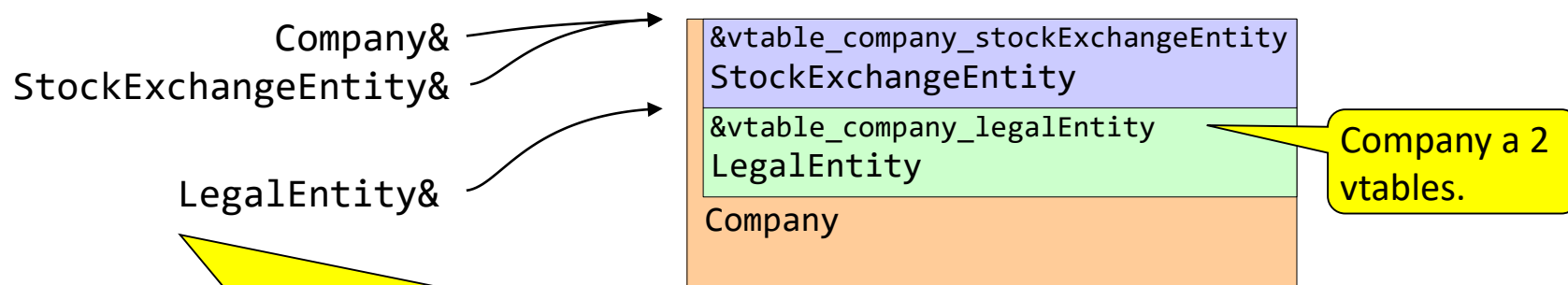
Le même objet passé à des fonctions dont les paramètres sont de types différents.

Héritage multiple et polymorphisme:

le même objet a deux types sans lien?

- hasShare/isAtMontreal peuvent être compilées sans la définition des autres classes (fichiers .cpp/.hpp séparés).
- C'est donc le « upcast » (implicite) lors de l'appel qui doit s'assurer de passer la bonne chose, **sans modifier ou construire de nouvel objet, juste en ajustant l'adresse.**

```
class Company : public StockExchangeEntity, public LegalEntity { ... };
```



L'objet dérivé a la **même adresse que la première base**, mais pas que les autres bases; le « cast » change donc l'adresse. Une méthode virtuelle change aussi l'adresse de **this** (on ne voit pas les détails dans ce cours).

Ordre de construction

Les arguments de constructions sont évalués juste avant de commencer à construire le sous-objet concerné.

1. Bases dans l'ordre de la déclaration après class
2. Attributs dans l'ordre de leur déclaration
3. Corps du constructeur de la classe est exécuté

L'ordre est indépendant de l'ordre dans la liste d'initialisation du constructeur. **Pour éviter toute confusion, on écrit toujours la liste d'initialisation dans l'ordre des déclarations** (ordre de construction ci-dessus). Donc aussi indépendant de quel constructeur est utilisé.

```
class Dérivée : public Base1, public Base2 {
```

```
...
```

```
Type1 attribut1;
```

```
Type2 attribut2;
```

```
};
```

Chaque base et attribut est un sous-objet qui suivra aussi cet ordre de construction, récursivement.

1.1. Base1
1.2. Base2
3. Corps constructeur Dérivée
2.1. attribut1
2.2. attribut2

Ordre de construction, exemple à éviter

```
class MaClasse : public Base1, public Base2 {
```

```
public:
```

```
    MaClasse(int n) :
```

```
        nValeurs(n),
```

nValeurs n'est pas encore initialisé puisque sa construction est après

```
        Base2("bonjour"),
```

```
        valeurs(new int[nValeurs]),
```

```
        Base1(getTexteBase2())
```

```
    {}
```

Appel d'une méthode de Base2 qui n'est pas encore construite

```
private:
```

```
    int* valeurs;
```

```
    int nValeurs;
```

```
};
```

On pouvait déclarer nValeurs avant valeurs, pour l'initialiser dans le bon ordre, ou faire new int[n]

Ordre de construction, exemple correct

```
class MaClasse : public Base1, public Base2 {
```

```
public:
```

```
    MaClasse(int n) :
```

Liste dans l'ordre de construction, pour éviter la confusion

```
        Base1("bonjour"),
```

```
        Base2(getTexteBase1()),
```

Base1 est bien construite lors de l'appel de la méthode

```
        nValeurs(n),
```

```
        valeurs(new int[nValeurs])
```

```
    {}
```

nValeurs est bien initialisé (on a changé l'ordre de déclaration des attributs)

```
private:
```

```
    int nValeurs;
```

```
    int* valeurs;
```

```
};
```

Si on voulait garder l'ordre des attributs, on pouvait mettre « new int[n] » ou mettre le « new int[nValeurs] » dans le corps du constructeur au lieu de la liste d'initialisation.

Ordre d'appel des constructeurs, exemple

```
class A
{
public:
    A();
private:
    B att_;
};
```

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

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

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

```
class E :
    public A,
    public D
{
public:
    E();
private:
    C att_;
};
```

```
int main()
{
    E objet;
}
```

On veut construire un objet de type E

Construit dans l'ordre:

A D C E()

Le corps du constructeur de E sera exécuté en dernier

En souligné ce qu'il reste à voir comment ces objets sont construits

Ordre d'appel des constructeurs, exemple

```
class A
{
public:
    A();
private:
    B att_;
};
```

1.2

1.1

```
int main()
{
    E objet;
}
```

Construit dans l'ordre:

A D C E()

B A()

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

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

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

```
class E :
    public A,
    public D
{
public:
    E();
private:
    C att_;
};
```

1

2

4

3

Ordre d'appel des constructeurs, exemple

```
class A
{
public:
    A();
private:
    B att_;
};
```

1.2

1.1

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

1.1.1

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

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

```
class E :
    public A,
    public D
{
public:
    E();
private:
    C att_;
};
```

1

2

4

3

```
int main()
{
    E objet;
}
```

Construit dans l'ordre:

A D C E()



└─┬─┘

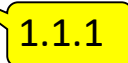
B A()

└─┘


B()





Ordre d'appel des constructeurs, exemple

```
class A
{
public:
    A(); 
private:
    B att_; 
};
```

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

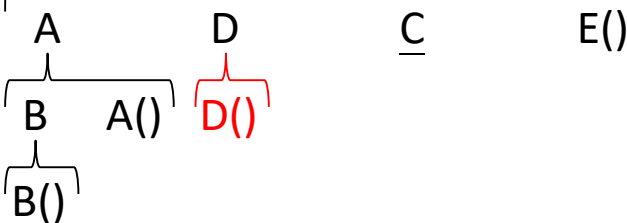
```
class C
{
public:
    C();
};
```

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

```
class E :
    public A, 
    public D 
{
public:
    E(); 
private:
    C att_; 
};
```

```
int main()
{
    E objet;
}
```

Construit dans l'ordre:



Ordre d'appel des constructeurs, exemple

```
class A
{
public:
    A(); 1.2
private:
    B att_; 1.1
};
```

```
class B
{
public:
    B(); 1.1.1
};
```

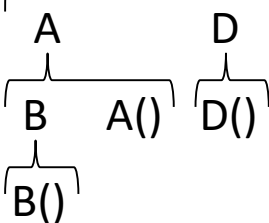
```
class C
{
public:
    C(); 3.1
};
```

```
class D
{
public:
    D(); 2.1
};
```

```
class E :
    public A, 1
    public D 2
{
public:
    E(); 4
private:
    C att_; 3
};
```

```
int main()
{
    E objet;
}
```

Construit dans l'ordre:



L'ordre de début de construction (évaluation des arguments s'il y en avait): E A B D C

Ordre de fin de construction = exécution des corps des constructeurs: B() A() D() C() E()

Ordre de destruction

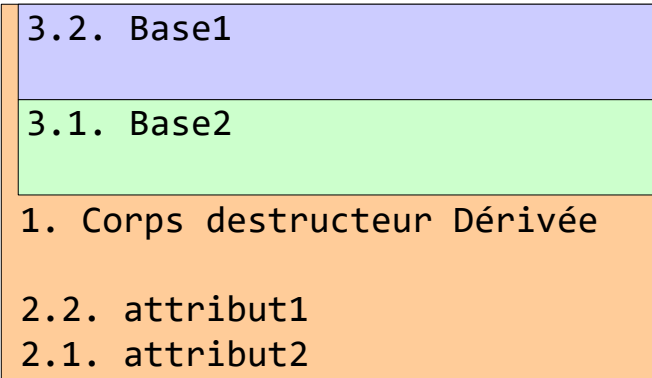
L'ordre de début de destruction (exécution des destructeurs) est l'inverse de l'ordre de fin de construction (exécution des constructeurs)

● Ordre inverse de la construction

1. Corps du destructeur de la classe est exécuté
2. Attributs dans l'ordre inverse de leur déclaration
3. Bases dans l'ordre inverse de la déclaration après class

```
class Dérivée : public Base1, public Base2 {
    ...
    Type1 attribut1;
    Type2 attribut2;
};
```

Chaque base et attribut est un sous-objet qui suivra aussi cet ordre de destruction, récursivement.



Ambiguïté de nom

- Si StockExchangeEntity et LegalEntity ont toutes les deux une méthode int getId()
 - `int id = myCompany.getId();` — **Erreur, quelle méthode appeler?**
- Une manière de régler le problème consiste à **indiquer explicitement la classe** de la méthode appelée :

```
int legalId = myCompany.LegalEntity::getId();
```

Objet sur lequel on appelle la méthode

Classe d'origine de la méthode

Méthode

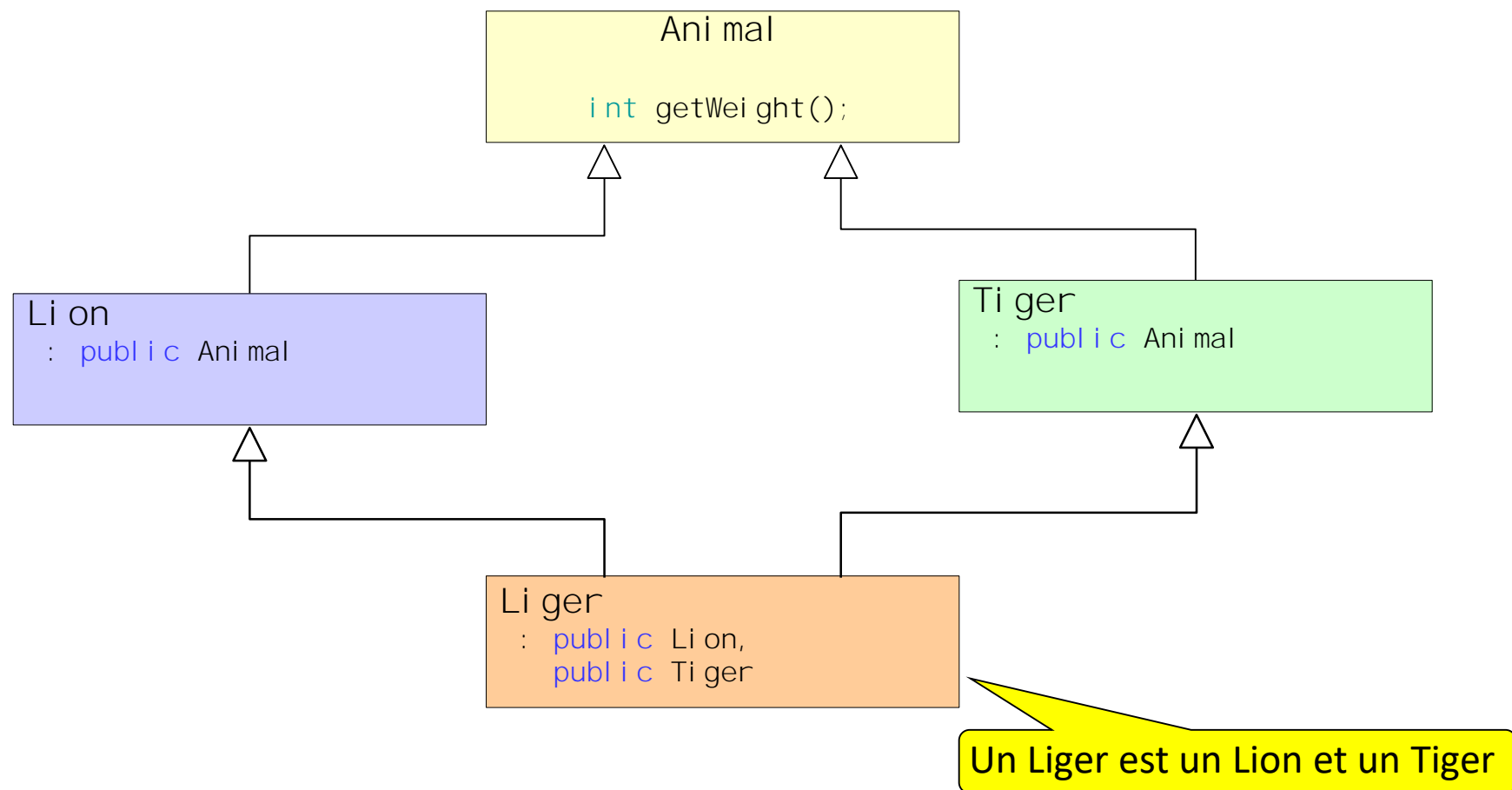
Ambiguïté de nom (suite)

- Une **meilleure solution** consiste à **cacher l'ambiguïté** et définir deux méthodes pour la classe `Company`:

```
int Company::getLegalId() const {  
    return LegalEntity::getId();  
}
```

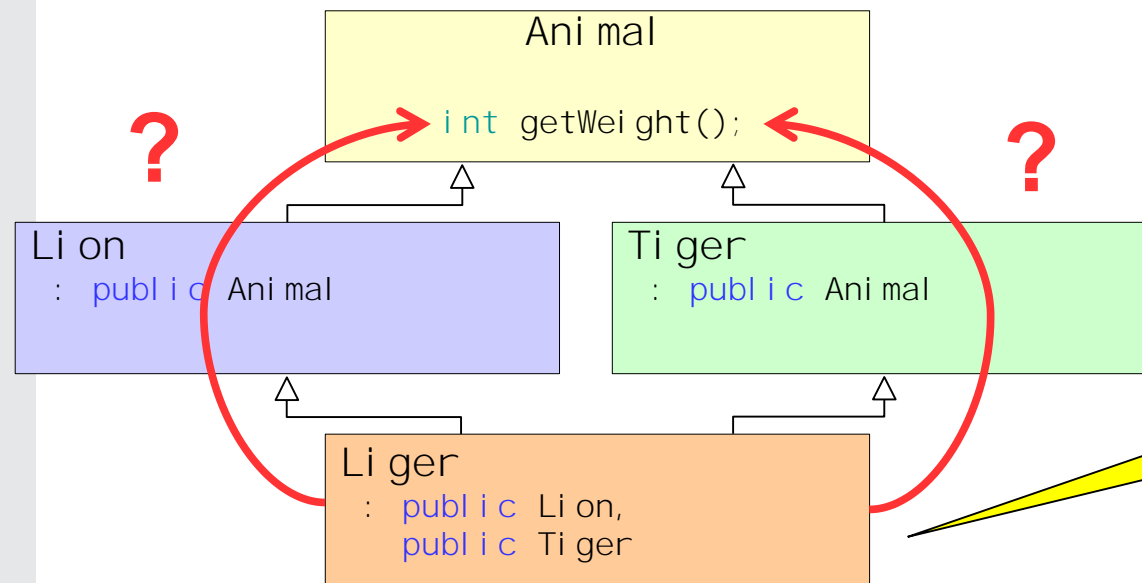
```
int Company::getStockId() const {  
    return StockExchangeEntity::getId();  
}
```

Problème du diamant



Problème du diamant (suite)

```
int main() {
    Liger lg;
    cout << "Liger weight: " << lg.getWeight();
}
```



Erreur, est-ce le poids de
sons **Animal Lion** ou **Tiger** ?

Ces déclarations disent que
Liger est **deux** animaux.

```
int weight = lg.Lion::getWeight();
```

Possible, mais un Liger devrait-
il être deux animaux?

Problème du diamant (suite)

héritage virtuel

- Un Liger n'étant pas deux animaux, on devrait utiliser l'héritage virtuel.
- Ça assure qu'un objet unique d'une classe multiplement dérivée existe: dans notre cas, un seul objet Animal, malgré que Tiger **et** Lion héritent de cette classe.
- Il suffit d'ajouter le mot clé **virtual** lorsqu'on hérite de la classe commune.

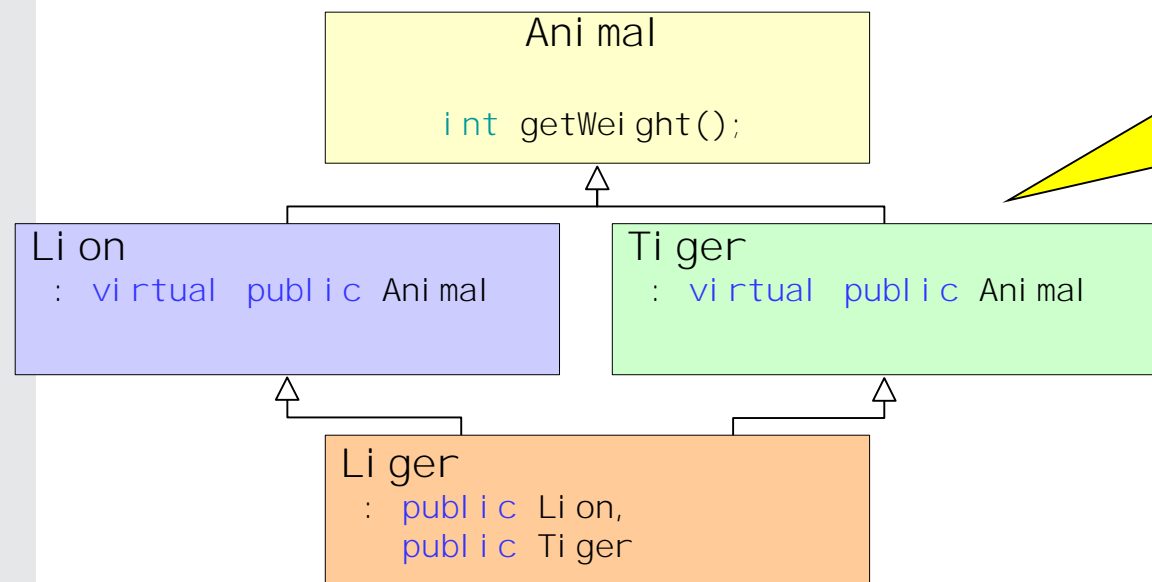
Il faut donc que cet héritage soit prévu par Lion et Tiger

```
class Lion : virtual public Animal /* ... */  
class Tiger : virtual public Animal /* ... */
```

Problème du diamant (suite)

héritage virtuel

```
int main() {
    Liger lg;
    cout << "Liger weight: " << lg.getWeight();
}
```



Ces déclarations disent que **Liger** est **un** Animal. Un seul objet Animal est créé, pas d'ambiguïté pour `getWeight()`

Problème du diamant (suite)

héritage virtuel et construction

- Lion et Tiger ont une base Animal et peuvent spécifier des arguments de construction
 - Quand Animal est construit? Avec quels arguments?
- Toutes les bases virtuelles de la classe et ses ancêtres sont construites avant toutes les autres constructions des bases
- On doit spécifier les arguments des bases virtuelles dans toutes les classes dérivées
 - Liger::Liger(...) : Animal(...), Lion(...), Tiger(...) {}

Dans l'héritage non virtuel on peut seulement donner des arguments aux classes de base directes

Attention avec l'héritage multiple

- D'autres problèmes peuvent être liés à l'héritage multiple
- L'héritage multiple de classes complètes peut compliquer la compréhension d'un code, et si un code n'est pas clair, il est difficile à partager et réutiliser
- Il ne faut pas utiliser inutilement l'héritage (multiple)
 - Est-ce que l'agrégation/composition serait préférable?
- L'héritage multiple d'interface est plus simple
 - Possiblement une classe de base complète + des interfaces

Attention avec l'héritage multiple (suite)

Basé sur <https://isocpp.org/wiki/faq/multiple-inheritance>

- Utiliser l'héritage seulement si ça enlève un if / switch / map de fonctions
 - L'héritage est principalement pour l'aiguillage dynamique, qui est généralement préférable d'utiliser celui du compilateur que de tenter de le faire à la main
- Utiliser des interfaces (classes abstraites pures) au dessus du point d'héritage multiple
 - Pour ne pas hériter de code ou donnée par deux chemins

Distinction entre type et interface

- Souvent une classe appartiendra à une seule hiérarchie de base
- à laquelle peuvent s'ajouter des interfaces

