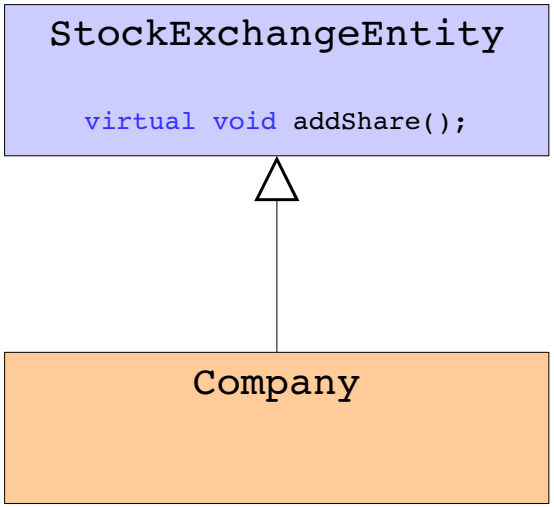
Programmation orientée objet

Héritage multiple

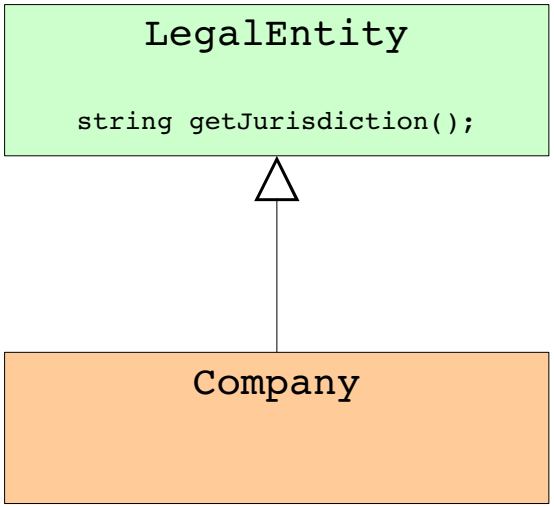
Motivation

- Reprenons notre exemple de `StockExchangeEntity`
- Nous savons qu'une `Company` est une `StockExchangeEntity`:



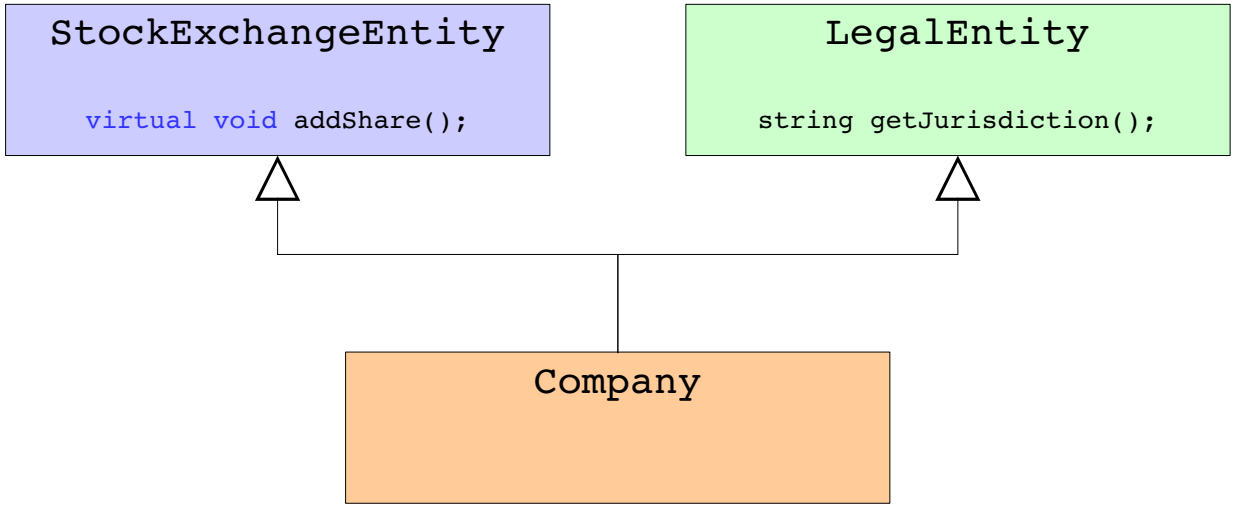
Motivation

- Mais une `Company` est aussi une entité juridique (qui peut être menée en procès, ...)



Motivation

- Cela signifie donc que la classe `Company` dérive de deux classes
- Certaines méthodes seront héritées d'une classe, d'autres seront héritées de l'autre classe



Héritage multiple en C++

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

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

Héritage multiple et polymorphisme

- Le polymorphisme fonctionne de la même manière avec l'héritage multiple
- La seule différence avec l'héritage simple est qu'un **même objet peut être pointé (ou référé**, s'il s'agit d'une référence**) 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 peut être placé dans deux vecteurs de types différents.

Héritage multiple et polymorphisme (exemple)

```
bool hasShares(const StockExchangeEntity& s) {
    return (s.getNbShares() > 0);
}
bool atMontreal(const LegalEntity& l) {
    return (l.getJurisdiction() == "Montreal");
}
int main() {
    Company c(/* ... */);
    if (hasShares(c)) {
        /* ... */
    }
    if (atMontreal(c)) {
        /* ... */
    }
    /* ... */
}
```

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

Ambiguïté de nom

- Supposons par exemple que les classes `StockExchangeEntity` et `LegalEntity` ont toutes les deux une méthode `getId()`, qui retourne un identificateur numérique

- Si un objet est de la classe `Company`, on ne pourra pas appeler la méthode `getId()` sur cet objet, puisque le compilateur ne saura pas quelle méthode appeler

Ambiguïté de nom (suite)

- Une manière de régler le problème consiste à **indiquer explicitement la classe** de la méthode appelée
- Supposons par exemple que `c` est un objet de la classe `Company` et que l'on veuille son identificateur d'entité légale, on utilisera alors l'expression suivante:

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

Objet sur lequel on appelle la méthode

Classe d'origine de la méthode

Méthode

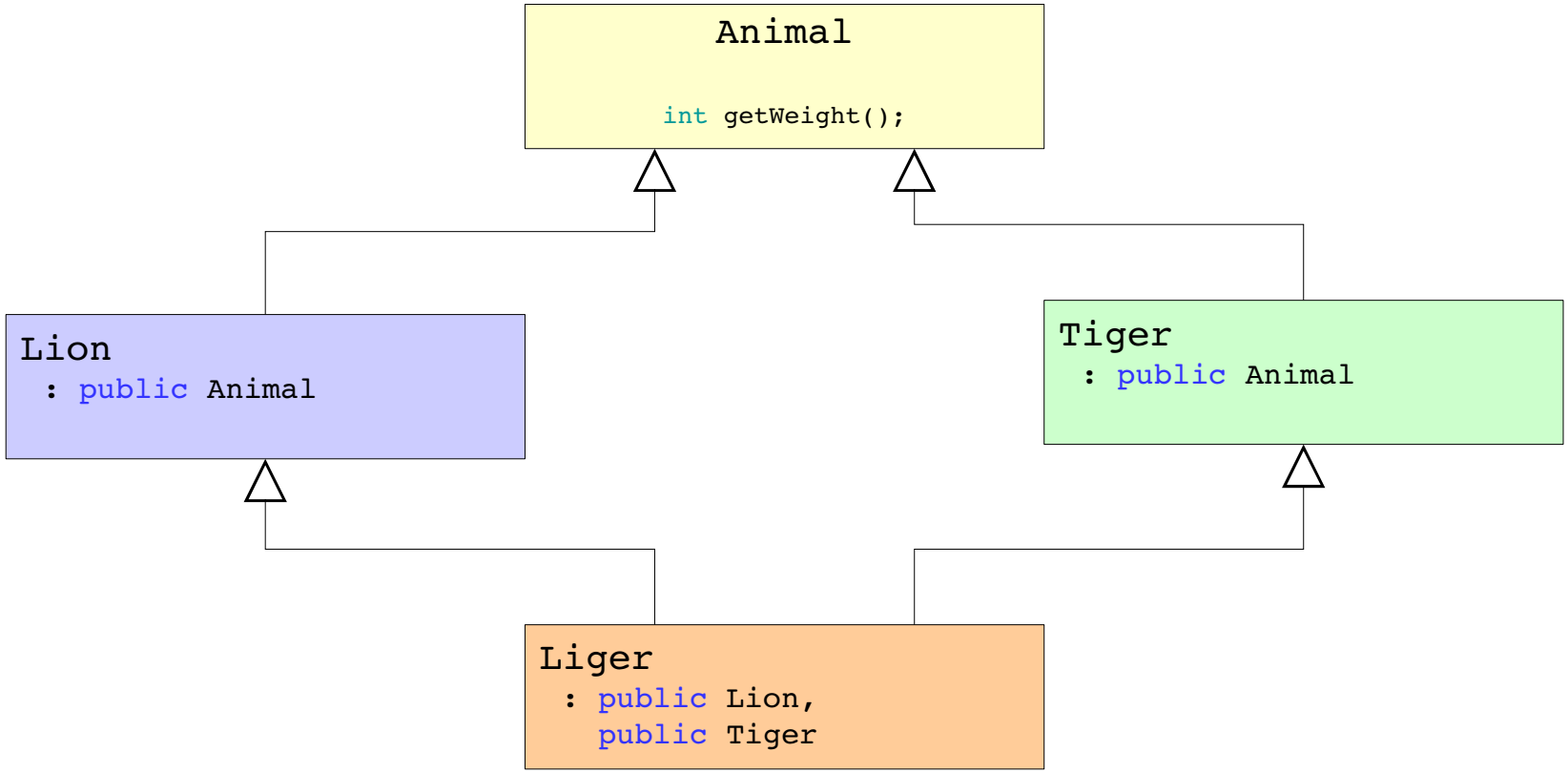
Ambiguïté de nom (suite)

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

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

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

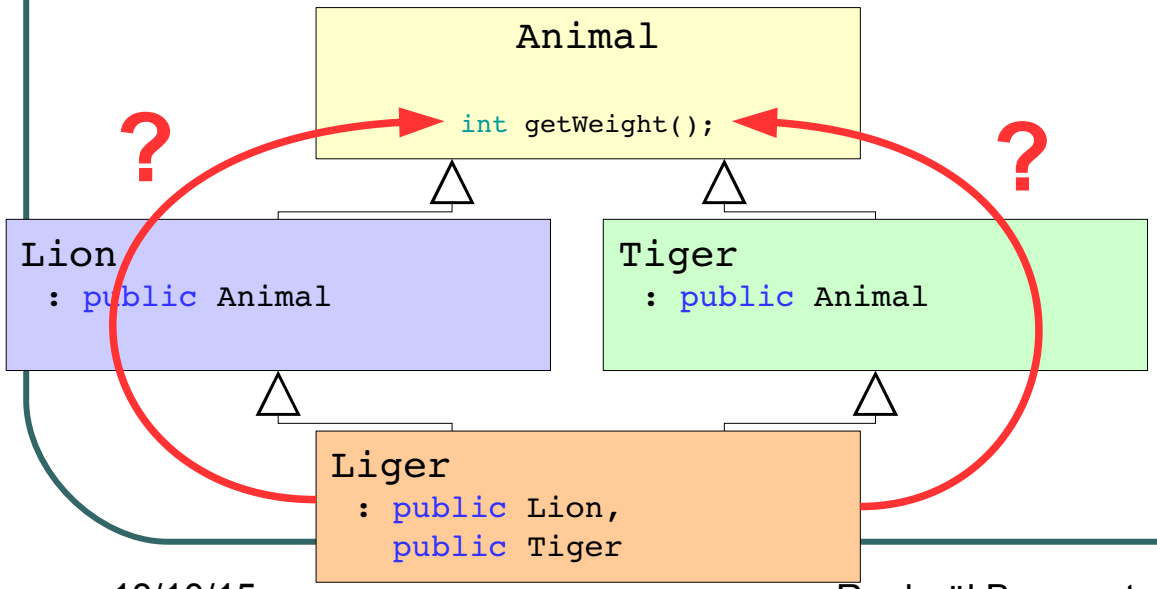
Problème du diamant



Problème du diamant (suite)

```
int main() {
   Liger lg;

   /* Erreur de compilation! */
   cout << "Liger weight: " << lg.getWeight();
}
```



On sait que l'on doit appeler **getWeight()** de la classe **Animal**, mais on ne sait pas quel chemin prendre: on passe par **Lion** ou par **Tiger** ?

Problème du diamant (suite)

- Ce problème est dû au fait que chacune des classes `Lion` comme `Tiger` hérite de `Animal`, et contient donc les attributs et méthodes de la classe `Animal`
- On pourrait régler le problème en précisant quel chemin prendre comme vu précédemment:

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

... mais ce n'est pas une bonne façon de faire dans ce cas, puisque `getWeight()` hérite directement de `Animal` dans ce cas précis.

Problème du diamant (suite)

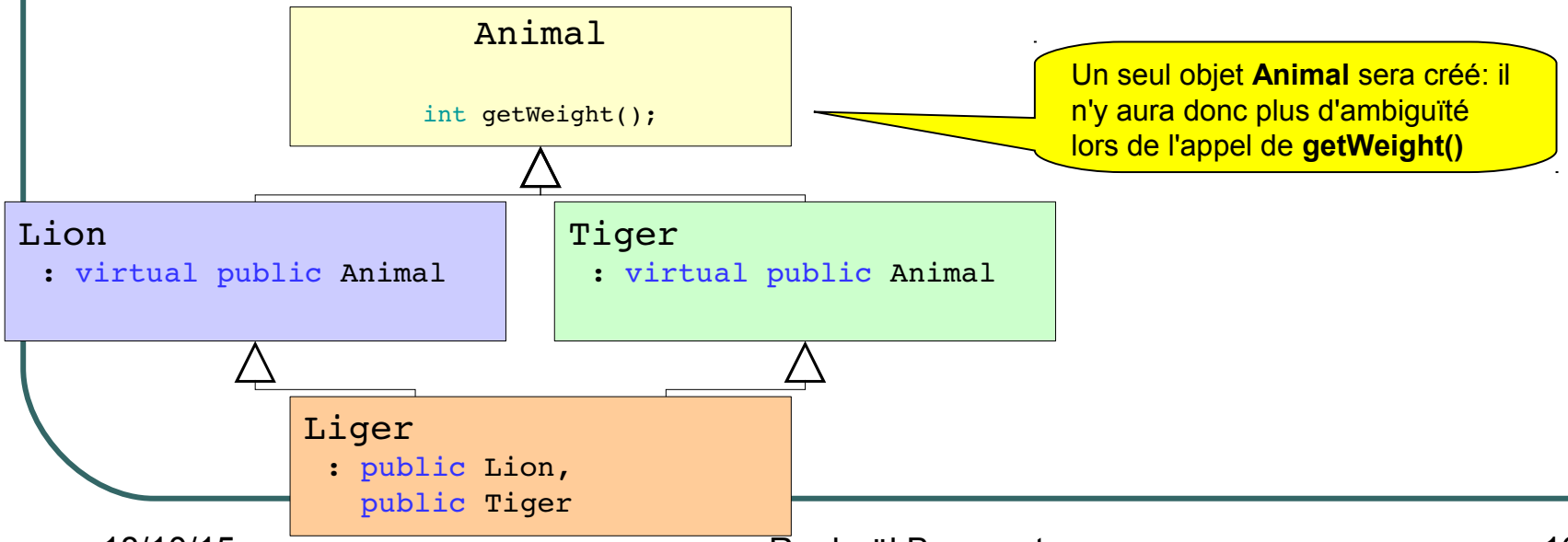
- À la place, pour régler ce genre de situation, on préfèrera utiliser l'**héritage virtuel**.
- Ce type d'héritage assure qu'un objet unique d'une classe multiplement dérivée existe: dans notre cas, un seul objet `Animal`, et ce bien que `Tiger` **ET** `Lion` héritent de cette classe.
- Pour cela, il suffit d'ajouter le mot clé `virtual` lorsqu'on hérite d'une classe.

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

Problème du diamant (suite)

```
int main() {
   Liger lg;

   /* Plus d'erreur! */
   cout << "Liger weight: " << lg.getWeight();
}
```

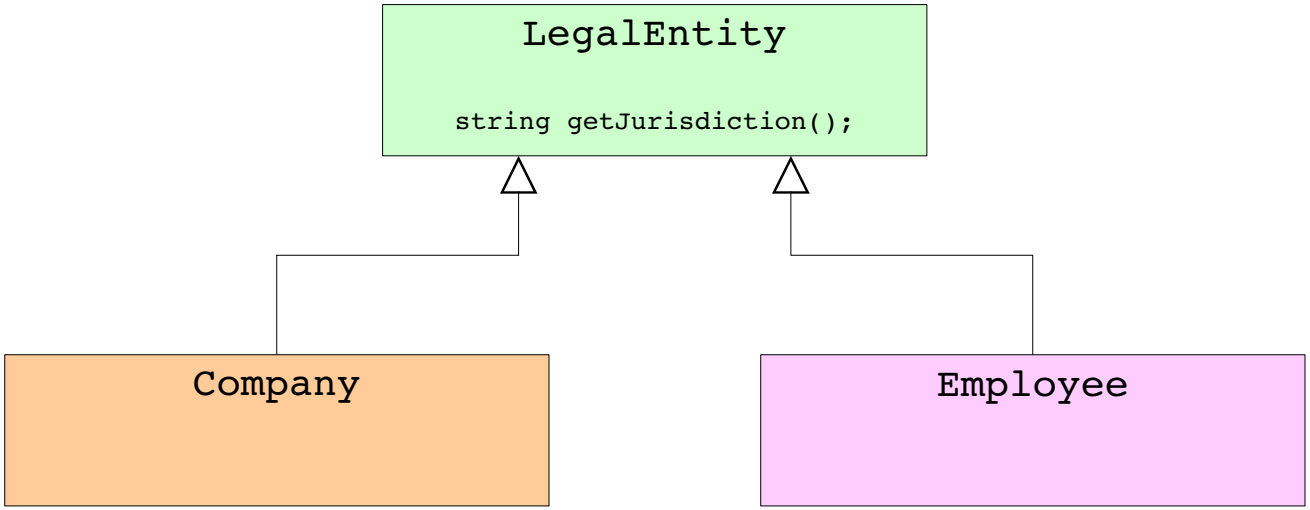


Attention avec l'héritage multiple

- Bien d'autres problèmes sont liés à l'héritage multiple
- L'héritage multiple peut compliquer substantiellement la compréhension d'un code, et si un code n'est pas clair, il est difficile à partager et réutiliser
- Il faut donc l'éviter le plus possible
- Que faire à la place?

Distinction entre type et interface

- En général une classe appartiendra à une seule hiérarchie de base:



Distinction entre type et interface

- À cette hiérarchie de base peuvent s'ajouter des interfaces (classes abstraites pures, i.e. classe ne contenant **que** des méthodes virtuelles pures):