Où en est-on ?

Nous connaissons maintenant les aspects essentiels de la POO :

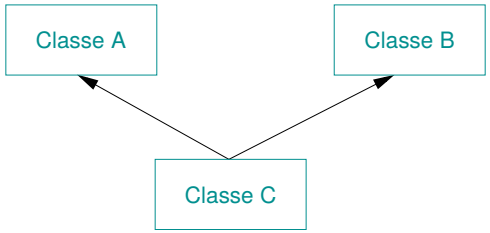
- Encapsulation et abstraction
  - regroupement traitements et données : `class Rectangle { ... };`
  - séparation entre interface et détails d'implémentation : `public`, `protected`, `private`
- Héritage : relation est-un

```
class Guerrier : public Personnage {...};
```

- Polymorphisme
  - pouvoir être vu de plusieurs façons, abstraction, généricité
  - 2 ingrédients nécessaires : pointeurs et méthodes virtuelles
  - classes abstraites et méthodes virtuelles pures

Qu'est-ce que l'héritage multiple ?

En C++, une sous-classe peut hériter de **plusieurs super-classes** :



Comme pour l'héritage simple, la sous-classe hérite des super-classes :

- tous leurs *attributs et méthodes* (sauf les constructeurs/destructeurs)
- leur *type*

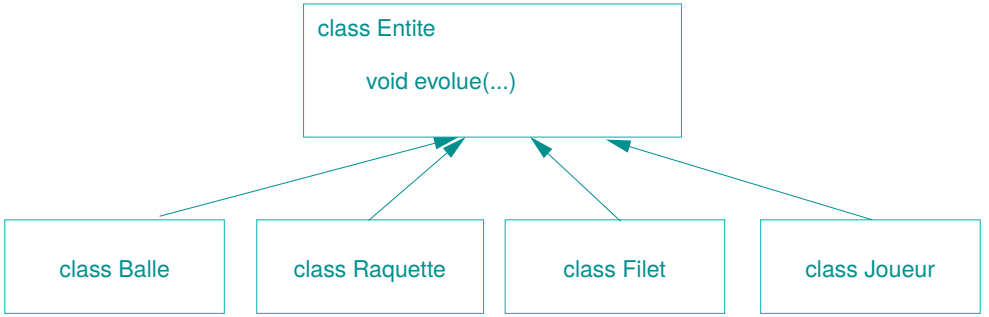
Encore un jeu ...

Supposons que l'on souhaite programmer un jeu mettant en scène les entités suivantes :

1. Balle
2. Raquette
3. Filet
4. Joueur

Chaque entité sera principalement dotée d'une méthode `evolue`, gérant l'évolution de l'entité dans le jeu.

Première ébauche de conception (1)



Première ébauche de conception (2)

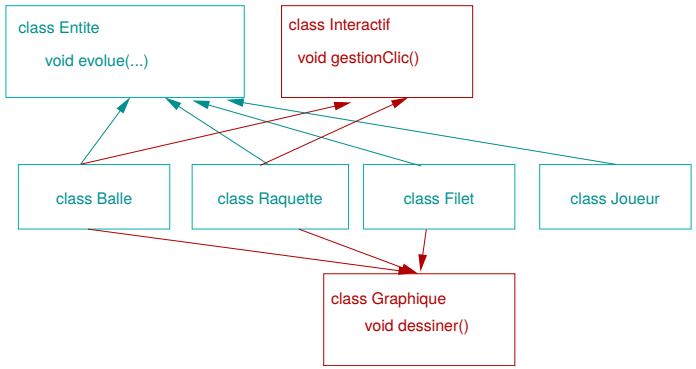
Si l'on analyse de plus près les besoins du jeu, on réalise que :

- certaines entités doivent avoir une représentation graphique (`Balle`, `Raquette`, `Filet`)
- ... et d'autres non (`Joueur`)
- certaines entités doivent être interactives (on veut par exemple pouvoir les contrôler avec la souris) : `Balle`, `Raquette`
- ... et d'autres non : `Joueur`, `Filet`

☞ Comment organiser tout cela ?

Jeu vidéo impossible

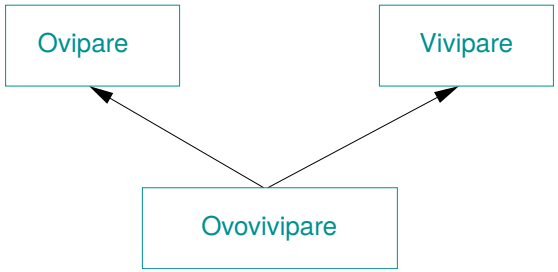
Il nous faudrait mettre en place une hiérarchie de classes telle que celle-ci :



☞ Possible en C++ grâce à l'héritage multiple !

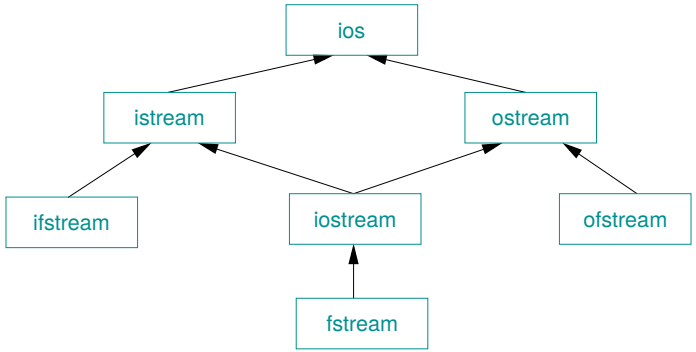
Autre exemple

Un exemple zoologique :



Encore un exemple

... informatique :



Héritage multiple

Syntaxe :

```
class nomSousClasse: public nomSuperClasse1, ...
                     public nomSuperClasseN {
//...
};
```

Exemple :

```
class Ovovivipare: public Ovipare, public Vivipare {
  public:
    Ovovivipare(unsigned int, unsigned int);
    virtual ~Ovovivipare();
  protected:
    bool espece_rare;
};
```

L'**ordre** de déclaration des super-classes est pris en compte lors de l'invocation des constructeurs/destructeurs

Constructeurs/destructeurs

Comme pour l'héritage simple, l'initialisation des attributs hérités doit être faite par invocation des constructeurs des super-classes :

Syntaxe :

```
SousClasse(liste de paramètres)
 : SuperClasse1(arguments1),
   ...
   SuperClasseN(argumentsN),
   attribut1(valeur1),
   ...
   attributK(valeurK)
{}
```

Lorsque l'une des super-classes admet un constructeur par défaut, il n'est pas nécessaire de l'invoquer explicitement.

Constructeurs/destructeurs (2)

Attention ! L'exécution des constructeurs des super-classes se fait **selon l'ordre de la déclaration d'héritage**, et non selon l'ordre des appels dans le constructeur !

L'ordre des appels des destructeurs de super-classes est **l'inverse** de celui des appels de constructeurs

Constructeurs/destructeurs : exemple

```
class Ovovivipare : public Ovipare, public Vivipare {
public:
  Ovovivipare(unsigned int nb_oeufs        ,
              unsigned int duree_gestation ,
              bool         rarete          = false );
  virtual ~Ovovivipare();
protected:
  bool espece_rare;
};

Ovovivipare::Ovovivipare(unsigned int nb_oeufs        ,
                         unsigned int duree_gestation ,
                         bool rarete)
  : Vivipare(duree_gestation), // Mauvais ordre !!
    Ovipare(nb_oeufs),
    espece_rare(rarete)
{}
```

Accès direct ambigu

Comme dans le cas de l'héritage simple, une sous-classe peut accéder directement aux attributs et méthodes protégés de ses super-classes.

... et si ces attributs/méthodes *portent le même nom* dans plusieurs super-classes ?

```
class Ovipare {
  // ...
  void afficher() const;
};


class Vivipare {
  // ...
  void afficher() const;
};
```

```
class Ovovivipare : public Ovipare,
                    public Vivipare
{ //...
};

int main()
{
  Ovovivipare o(...);
  o.afficher();

  return 0;
}
```

Accès direct ambigu (2)

Attention ! L'accès `o.afficher` provoquera une erreur à la compilation **même** si la méthode `afficher` **n'**avait **pas** les mêmes paramètres dans les deux classes `Ovipare` et `Vivipare` !!!

(La raison est qu'en C++, il n'y a surcharge que dans la même portée. Ici ce n'est pas une problème de surcharge, mais un problème de masquage [résolution de portée].)

```
class Ovipare {
  // ...
  void afficher() const;
};


class Vivipare {
  // ...
  void afficher(string const& entete) const;
};
```

```
class Ovovivipare : public Ovipare,
                    public Vivipare
{ //...
};

int main()
{
  Ovovivipare o(...);
  o.afficher("Un orvet : ");

  return 0;
}
```

Accès direct ambigu – solutions ?

Solutions ?

Première solution : utiliser l'*opérateur de résolution de portée*.

```
int main()
{
  Ovovivipare o(...);
  o.Vivipare::afficher("Un orvet : ");

  return 0;
}
```

mais...

Accès direct ambigu – critique de la solution 1

mais...

L'utilisation (externe) de l'opérateur de résolution de portée pour résoudre les ambiguïtés de noms des attributs/méthodes **n'est pas une bonne solution** :

C'est l'*utilisateur* de la classe `Ovovivipare` qui décide du fonctionnement correct de cette classe,

alors que cette responsabilité doit normalement **incomber aux concepteurs** de la classe.

Accès direct ambigu – solution 2

Une des solutions consiste à lever l'ambiguïté en *indiquant explicitement* dans la sous-classe quelle(s) méthode(s) on veut invoquer :

☞ ajouter à la sous-classe, une *déclaration spéciale* indiquant quel(s) méthode(s)/attribut(s) seront invoqué(s) exactement.

Syntaxe : `using NomSuperClasse::NomAttributOuMethodeAmbigu ;`

Exemple :

```
class Ovovivipare : public Ovipare, public Vivipare {
public:
  using Vivipare::afficher;
  // ...
};
```

Attention ! Pas de parenthèse (ni prototype) derrière le nom de la méthode !

Accès direct ambigu – solution 3

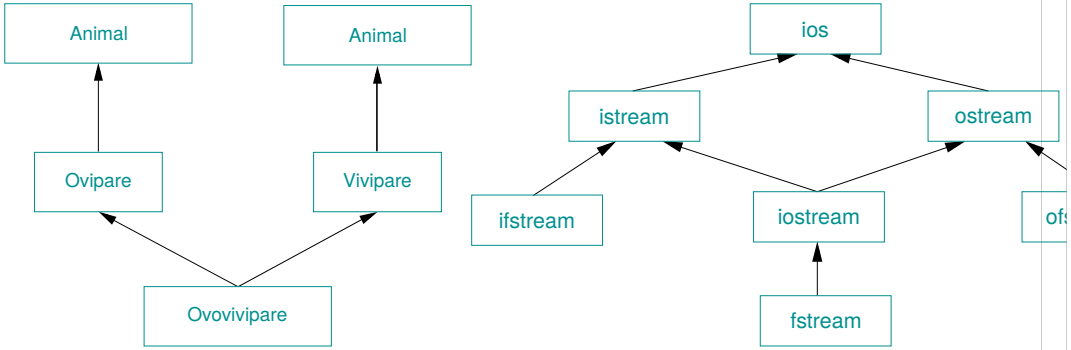
La meilleure solution consiste à ajouter dans la sous-classe une méthode *définissant la bonne interprétation* de l'invocation ambiguë.

Exemple :

```
class Ovovivipare: public Ovipare, public Vivipare {
  public:
  // ...
  void afficher() const {
    Ovipare::afficher();
    Vivipare::afficher(" mais aussi pour sa partie"
                       " vivipare : ");
  }
  // ...
};
```

Classes virtuelles : problème

Il peut se produire qu'une super-classe soit incluse *plusieurs fois* dans une hiérarchie à héritage multiple :



Les attributs/méthodes de la super-classe seront inclus plusieurs fois !

☞ Chaque objet de la classe `Ovovivipare` possédera **deux** copies des attributs de la classe `Animal`.

Classes virtuelles : exemple du problème

```
class Animal {
public:    Animal(string const& description) : tete(description) {}
protected: string tete;
};

class Ovipare  : public Animal { public:  Ovipare()  : Animal("à cornes")   {} };
class Vivipare : public Animal { public:  Vivipare() : Animal("de poisson") {} };

class Ovovivipare : public Ovipare, public Vivipare {
public:
  void affiche() const { cout << "j'ai une tête " << Ovipare::tete
                              << " et une tête "  << Vivipare::tete << " !" << endl;
                       }
};
// ...
    Ovovivipare x;
    x.affiche();
```

☞ `j'ai une tête à cornes et une tête de poisson !`

Classes virtuelles solution

Pour *éviter la duplication des attributs* d'une super-classe plusieurs fois incluse lors d'héritages multiples, il faut déclarer son **lien d'héritage** avec *toutes* ses sous-classes comme **virtuel**.

Cette super-classe sera alors dite « **virtuelle** »
(à *ne pas confondre* avec classe abstraite ! !)

Syntaxe :

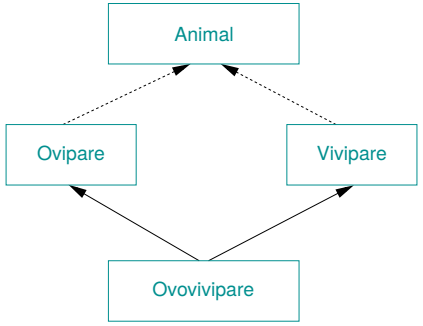
```
class NomSousClasse : public virtual NomSuperClasseVirtuelle
```

Exemple :

```
class Ovipare : public virtual Animal { // ...
// ...
class Vivipare: public virtual Animal { // ...
```

A noter que c'est la classe pouvant être héritée plusieurs fois qui est virtuelle (i.e. ici la super-super-classe) et non pas directement les classes utilisées dans l'héritage multiple (i.e. les super-classes).

Classes virtuelles (3)



Un seul objet de la super-classe `Animal` est hérité par l'héritage commun des sous-classes `Ovipare` et `Vivipare`.

Constructeurs et Classes virtuelles

Rappel : dans un héritage usuel, le constructeur d'une sous-classe ne fait appel *qu'aux constructeurs de ses super-classes immédiates* (et ceci récursivement)

Dans une dérivation *virtuelle*, la super-classe virtuelle est **initialisée directement par la sous(-sous-...)-classe instanciée**

- ☞ Toutes ses sous(-sous-...)-classes instanciables (= non-abstraites) **doivent** donc faire **directement** *appel au constructeur de la super(-super-...)-classe virtuelle*

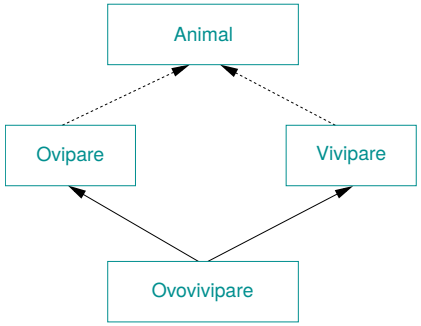
Constructeurs et Classes virtuelles : Exemple

```
Ovovivipare::Ovovivipare(string nom, Habitat habitat, Regime regime,
                         unsigned int nb_oeufs,
                         unsigned int gestation,
                         bool rarete = false)
   : Animal(nom, habitat, regime),
     Ovipare(nb_oeufs),
     Vivipare(gestation),
     espece_rare(rarete)
{}
```

Classes virtuelles : appel des constructeurs

Comment sont gérés les appels au constructeur de la super-classe virtuelle ?

```
Ovovivipare::Ovovivipare(// ...
                         )
   : Animal(nom, habitat, regime),
     Ovipare(nb_oeufs),
     Vivipare(gestation),
     espece_rare(rarete)
{}
```




```
Ovovivipare o(...);
```

Classes virtuelles : appel des constructeurs

Lors de la création d'un objet d'une classe plus dérivée, son constructeur invoque directement le constructeur de la super-classe virtuelle.
Les appels au constructeur de la super-classe virtuelle dans les classes *intermédiaires* **sont ignorés**.

Si la super-classe virtuelle a un *constructeur par défaut*, il n'est pas nécessaire de faire appel à ce constructeur explicitement. Il sera directement appelé par le constructeur de la classe dont on crée une instance.

S'il n'y a pas d'appel explicite au constructeur de la super-classe virtuelle et si celle-ci n'a pas de constructeur par défaut, *la compilation signalera une erreur*.

Ordre des constructeurs/destructeurs

Dans une hiérarchie de classes où il existe des super-classes virtuelles :

- le soin d'initialiser les super-classes virtuelles incombe à la sous-classe la plus dérivée
- les constructeurs des super-classes virtuelles *sont invoqués en premier*
- les appels explicites au constructeur de la super-classe virtuelle dans les classes intermédiaires *sont ignorés*.
- ceux des classes non-virtuelles le sont ensuite *dans l'ordre de déclaration de l'héritage*
- l'ordre d'appel des *constructeurs de copie* est identique
- l'ordre d'appel des *destructeurs* est *l'inverse* de celui des appels de constructeurs