# Programmation Orientée Objet
# Héritage

Cyril Prissette
prissette@univ-tln.fr

SeaTech

# Introduction

Lors des séances précédentes, la P.O.O. a été présentée comme un moyen d'obtenir un code source parfaitement organisé afin d'assurer sa fiabilité :

- L'encapsulation assure l'intégrité des données des objets.
- La forme canonique sécurise leur cycle de vie (création, recopie, affectation et destruction)

Nous abordons maintenant un autre aspect de la P.O.O. : la ré-utilisation de code source, dans un contexte proche de celui dans lequel elle a été développée.

# Contexte

Nous allons revenir à la classe Vehicule, en nous plaçant du point de vue d'une société qui gère un parc de véhicules. Avant de poursuivre, vérifiez que vous avez bien une classe Vehicule, encapsulée et son forme canonique, et séparée en fichiers .h et .cpp
Si ce n'est pas le cas, faites le nécessaire.
★ au travail!

# Contexte

Notre société de gestion de **Vehicule**s a pris de l'ampleur et notre activité se diversifie : hier, nous pensions en terme de trafic routier, mais aujourd'hui nous nous attaquons au trafic maritime.

Hélas, bien que la classe **Vehicule** marche correctement, on déplore déjà 3 naufrages faute de pouvoir gérer correctement le tirant d'eau[1] de nos bateaux.

[1] c'est à dire la hauteur immergée de la coque

# Classe Bateau

Pour résoudre ce problème à cela, on souhaite créer une nouvelle classe **Bateau**, très similaire à **Vehicule**, à quelques différences près. Cette classe devra posséder, en plus des attributs de Véhicule, un nouvel attribut :

- int tirant

Et trois nouvelles méthodes :

- int get_tirant()
- set_tirant(int), qui ne peut pas être négatif
- bool tester_surete_profondeur(int profondeur), qui renvoie true si le tirant d'eau est inférieur à la profondeur

# Classe Bateau : mauvaise solution 1

Pour concevoir la classe Bateau, on pourrait copier/coller la classe Vehicule, la renommer en Bateau, ajouter l'attribut tirant et les 3 méthodes demandées, et adapter les méthodes existantes si besoin.

Problème : si on doit modifier par la suite la classe Véhicule, il faudra reporter les modifications sur Bateau.

Et plus le système va se complexifier avec de nouveaux type de Vehicules (on parle déjà d'Avions, mais on pourrait aussi développer une activité sous-marine et une activité spatiale), plus la maintenance de l'ensemble du code sera pénible.

Conclusion : ce n'est pas satisfaisant, d'un point de vue maintenance du code.

# Classe Bateau : mauvaise solution 2

Créer une classe bateau qui possède 2 attributs: un Vehicule veh et un tirant, et écrire des méthodes pour appeler les méthodes de vehicule

```
class Bateau
{
protected :
        Vehicule veh;
        int tirant;
...
        int get_vitesse() { return veh.get_vitesse(); }
        void accelerer() { veh.accelerer(); }
// et idem pour toutes les autres méthodes
};
```

Problème pratique : si on ajoute une méthode à Vehicule, elle n'est pas immédiatement étendu à Bateau.

# Classe Bateau : mauvaise solution 2 (suite)

Problème conceptuel : la classe ainsi construite exprime l'idée qu'un Bateau est constitué d'un véhicule et d'un tirant d'eau.
C'est particulièrement visible au niveau des attributs :
Paquebot.veh.vitesse et Paquebot.tirant, les attributs vitesse et tirant devraient être hiérarchiquement au même niveau : ce sont des caractéristiques d'un Bateau.

Problème sémantique : on ne sait pas vraiment ce qu'exprime un "veh"

Conclusion : ce n'est pas satisfaisant, d'un point de vu modélisation des classes. Soyons franc, c'est même carrément du bricolage.

# Classe Bateau : bonne solution

Ce qu'on cherche à exprimer, c'est qu'un "Bateau **est un** Vehicule".

La relation **"... est un ..."** indique généralement qu'il y a un **héritage** entre deux classes.

C'est un mécanisme extrêmement important de la P.O.O. : il permet de définir correctement une nouvelle classe (ex: Bateau) grâce à une classe existante (ex : Vehicule), en la **spécialisant**.

La classe déjà existante est généralement appelée **classe mère**.
La nouvelle classe est généralement appelée **classe fille**.

# Déclaration d'un héritage

Ajoutez dans votre projet une nouvelle classe (fichiers .cpp et .h) avec cette définition de **Bateau** :

```
class Bateau : public Vehicule  // <= héritage déclaré ici
{
protected :
        int tirant;
public :
        int get_tirant() {return tirant;}
        void set_tirant(int t) {if (t>0) tirant=t;}
};
```

Puis dans le main(), déclarez et utilisez un objet de la classe Bateau.
Essayez par exemple get_vitesse(), accelerer(), set_tirant() et get_tirant()
Ecrivez et testez la méthode bool tester_surete_profondeur(int profondeur)
★ Au travail!

# Forme canonique

Ajoutez l'affichage de messages de debug dans **Vehicule**. Ceci va nous permettre d'explorer comment marche l'héritage, au cours de la vie d'un objet.

Afin de conserver une classe **Vehicule** "propre", ajoutez les affichages de la manière suivante :

```
Vehicule::Vehicule()
{
#ifdef __DEBUG
cerr << "construction par defaut de Vehicule" << endl;
#endif

vitesse=0;
vitesse_max=130;
poids=1.0;
}
```

# Forme canonique (suite)

Pour passer en mode affichage debug, il suffit à présent d'ajouter la ligne suivante en debut de fichier :

```
#define __DEBUG
```

Écrivez un main() permettant de tester le cycle de vie d'un objet de la classe **Bateau** (instanciation, recopie, affectation et destruction). Quelles méthodes de la forme canonique héritées de **Vehicule** sont implicitement invoquées?
Ajoutez maintenant des affichages debug dans la classe **Bateau**. Quelles méthodes de la forme canonique de **Bateau** sont invoquées? Dans quel ordre les méthodes héritées de **Vehicule** et celles spécifiques à **Bateau** sont-elles invoquées? ★

# Forme canonique (fin)

Complétez, si nécessaire, le constructeur par défaut, le constructeur par recopie, l'opérateur d'affectation et le destructeur pour qu'ils remplissent correctement leur office.
En particulier, le constructeur par défaut de **Bateau** doit définir sa vitesse maximale à 60 (ce sont des noeuds). ★

# Ecriture d'une méthode spécialisée

Bien que la classe **Bateau** semble fonctionner parfaitement, le service Trafic Maritime remonte un problème : d'après la classe qu'on vient de leur livrer, les bateaux accélèrent et ralentissent de 50 noeuds en 50 noeuds.

Pour pallier ce problème, un stagiaire leur a écrit le code suivant :

```
void accelerer(Bateau &B)
{
int V;
V = B.get_vitesse();
B.set_vitesse(V+5);
}
```

Il n'a hélas pas eu le temps de finir la fonction ralentir() avant la fin de son stage.

# Ecriture d'une méthode spécialisée (suite)

Cette fonction n'est pas souhaitable.

Elle définit une fonction (non orientée objet, donc), là où il serait parfaitement possible d'écrire une méthode.
Et surtout, on conserve une méthode acceler(), héritée de **Vehicule**, qui ne fonctionne pas correctement dans le contexte de **Bateau**.

Modifiez la classe **Bateau** afin que les méthodes accelerer() et ralentir() correspondent à ce comportement spécifique, différent du fonctionnement de **Vehicule**. ★

# Polymorphisme

L'héritage en C++ a une particularité : la classe fille a le même type de pointeur que la classe mere. On parle de **polymorphisme**.

Pour vous en convaincre, faites le test suivant :

```
int main()
{
Vehicule *ptr_v;
ptr_v = new Bateau;
cout << ptr_v->get_vitesse() << endl;
}
```

★ Si le programme fonction, cela signifie qu'on peut effectivement stocker l'adresse d'un Bateau, dans un pointeur de Vehicule.

# Polymorphisme

L'application la plus courante de ce mécanisme est de pouvoir créer un tableau qui mélange plusieurs types d'objets différents, à conditions qu'ils appartiennent à la même classe mère ou à des classes qui héritent de cette classe mère.
Ecrivez un programme ayant un `vector<Vehicule*> flotte` qui pointe vers 3 objets : un Bateau, un Vehicule et un autre Bateau.
Affichez, à l'aide d'une boucle, la vitesse_max de chacun de ces objets.

★

# Polymorphisme

Modifiez votre programme pour qu'il fasse les opérations suivantes :

- affiche la valeur de get_vitesse() pour chacun de ces 3 objets.
- execute la méthode accelerer() pour chacun de ces 3 objets.
- affiche à nouveau la valeur de get_vitesse() pour chacun de ces 3 objets.

A l'execution, les Bateaux accelerent-ils comme il le devraient?

★

# Polymorphisme

Le comportement étrange de la méthode accelerer() pour les Bateaux s'explique de la manière suivante : comme on utilise des pointeurs vers Vehicule, le comportement de base de C++ est de considérer que les adresses en question sont des adresses de Vehicule.
Quand on lui demande d'accelerer un Vehicule, il utilise donc naturellement la méthode Vehicule::accelerer()
Cela peut sembler incohérent avec le concept de polymorphisme, mais c'est en fait une question de vitesse d'execution : il ne perd pas de temps à vérifier finement le type des objets pointés, il se contente de supposer que leur type est celui indiqué par le pointeur.

# Polymorphisme - virtual

Bien sûr, le C++ fournit un mécanisme permettant de signaler qu'une méthode peut être surchargée dans une classe fille et qu'il faut alors qu'il analyse finement le type des pointeurs de classe.
Dans la classe Vehicule, ajoutez le mot clé `virtual` devant le prototype de la méthode accelerer(), et vérifiez dans votre programme que les Bateaux accelèrent maintenant correctement.
★

# Polymorphisme - virtual

Dans toutes les classes, une méthode en particulier devrait être toujours virtuelle : le destructeur.
Par exemple, analysons le code suivant :

```
int main()
{
Vehicule *ptr_v;
ptr_v = new Bateau;
delete ptr_v;
}
```

Le fonctionnement par défaut du C++ l'amène à utiliser Vehicule:: Vehicule() quand il execute le delete, puisque ptr_v est un Vehicule*
En déclarant `Vehicule::~Vehicule()` en virtual[2], il vérifiera finement si il doit détruite un Vehicule ou un Bateau.

[2] ce que proposent Codeblocks et d'autres IDE

# Polymorphisme - virtual

Definissez en virtual toutes les méthodes de Vehicules et de Bateaux qui le nécessitent.

★

# Classe abstraite

Malgré l'existence de la classe Bateau, on déplore un nouveau naufrage. Selon l'enquête en cours, il semblerait que la classe Vehicule soit encore utilisée dans certaines branches de l'entreprise pour manipuler ce qui est -en réalité- des bateaux.
On souhaite marquer une différence plus nette entre les différents types de véhicules, en mettant en place les règles suivantes :

- Les Bateaux **sont des** Vehicules
- Les Voitures **sont des** Vehicules
- Il est interdit d'instancier un Vehicule, on ne peut instancier que des Bateaux et des Voitures.

# Classe abstraite

Créez la classe Voiture qui hérite de la classe Véhicule.
Prenez soin d'encapsuler cette nouvelle classe et de la mettre sous forme canonique.
★

# Classe abstraite

A présent, on souhaite interdire l'instanciation de Vehicule. Il ne sera donc plus possible de créer des objets de cette classe. On parle de **classe abstraite**.
Une classe abstraite se distingue par la présence d'une **méthode virtuelle pure**. Une telle méthode est créée en ajoutant =0 à son prototype. Par exemple :

```
class Vehicule
{
..
virtual void accelerer()=0;
..
};
```

Faites cette modification dans vehicule.h, et toutes les modifications qui en découlent. Verifier dans votre main() qu'il n'est plus possible d'instancier de Vehicule, mais qu'il est toujours possible d'instancier des Bateaux et des Voitures. ★