



# Programmation Orientée Objet

Maxime MARIA

[maxime.maria@u-pem.fr](mailto:maxime.maria@u-pem.fr)



# Plan du cours (à peu près)

- 6 CM de 2h
  - 21/09 : Notion de POO et premiers pas en C++
  - 18/10 : POO en C++
  - 25/10 : Héritage et polymorphisme
  - 08/11 : Templates et utilisation de la STL
  - 15/11 : Les exceptions
  - 21/11 : On verra... Mais ce sera dur, très dur ! 😊

**APRÈS ÇA, VOUS SEREZ DES BÊTES DE C++ !**

(Ou pas...)

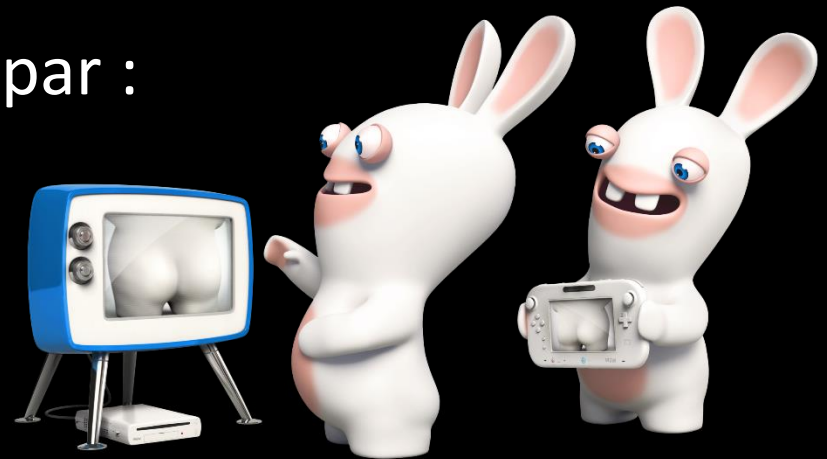
# INTRODUCTION À LA PROGRAMMATION ORIENTÉE OBJET (POO)

On se réveille !



# La POO c'est quoi ?

- Paradigme de programmation
  - Faciliter le développement et la maintenance des programmes
  - Représenter plus concrètement le monde réel
- Modélisation d'un problème par :
  - Les objets qui interviennent
  - Leur comportement
  - Leurs interactions
- Exemple : fonctionnement d'un jeu sur console



# Objectifs de la POO

- Améliorer la vie du développeur !
- Conception plus intuitive
  - Notions de classe et d'objet
- Faciliter la réutilisation et l'évolution du code :
  - Encapsulation
  - Héritage
  - Polymorphisme



# Objectifs de la POO

- Améliorer la vie du développeur !
- Conception plus intuitive
  - Notions de classe et d'objet
- Faciliter la réutilisation et l'évolution du code :
  - Encapsulation
  - Héritage
  - Polymorphisme



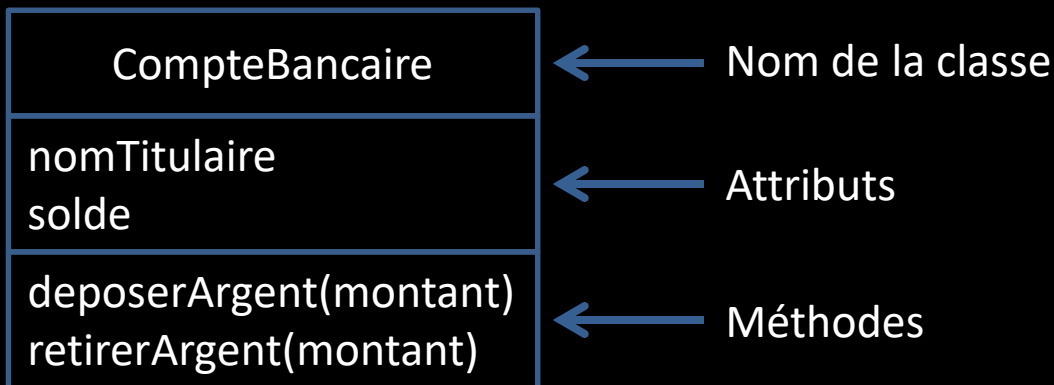
# Notion de classe

- Classe = structure de données représentant un objet
- Deux types de membres :
  - Attributs : variables définissant l'état de l'objet
  - Méthodes : fonctions définissant le comportement de l'objet

|  |                    |
|--|--------------------|
| CompteBancaire                                   | ← Nom de la classe |
| nomTitulaire<br>solde                            | ← Attributs        |
| deposerArgent(montant)<br>retirerArgent(montant) | ← Méthodes         |

# Notion de classe

- Classe = structure de données représentant un objet
- Deux types de membres :
  - Attributs : variables définissant l'état de l'objet
  - Méthodes : fonctions définissant le comportement de l'objet

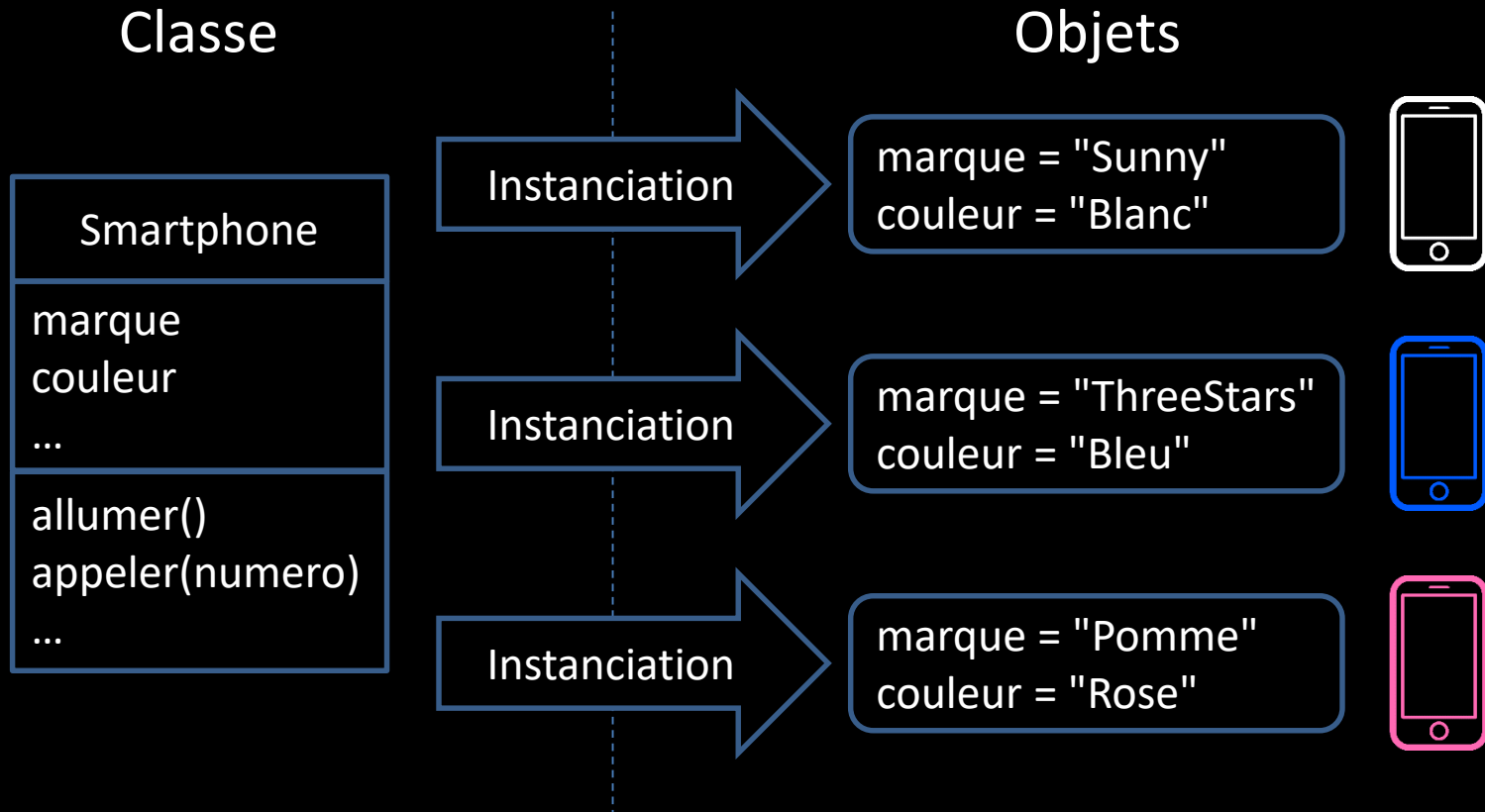


- UML : Langage de modélisation (pas le sujet de ce cours)



# Notion d'objet

- Objet = instance de classe



- Instantiation = construction

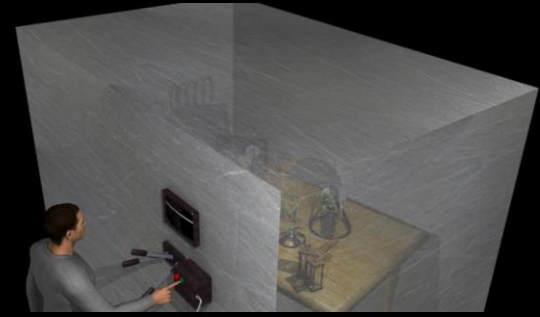
# Objectifs de la POO

- Améliorer la vie du développeur !
- Conception plus intuitive
  - Notions de classe et d'objet
- Faciliter la réutilisation et l'évolution du code :
  - Encapsulation
  - Héritage
  - Polymorphisme



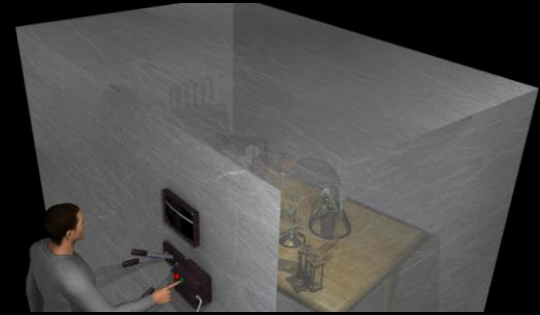
# Encapsulation

- Implémentation interne cachée
  - On ne manipule pas directement les attributs
  - On utilise les méthodes mises à disposition (interface)



# Encapsulation

- Implémentation interne cachée
  - On ne manipule pas directement les attributs
  - On utilise les méthodes mises à disposition (interface)
- Exemple : translation d'un point 2D

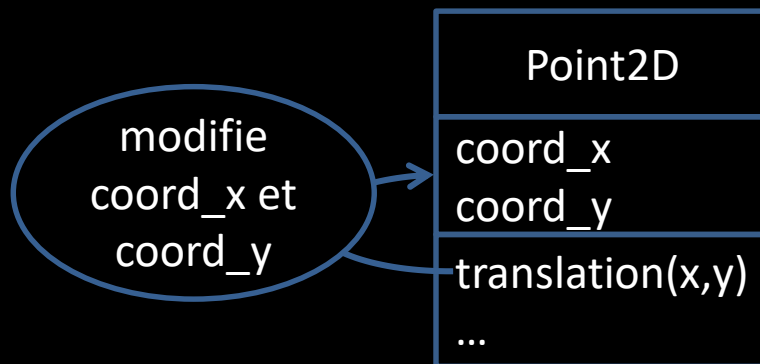
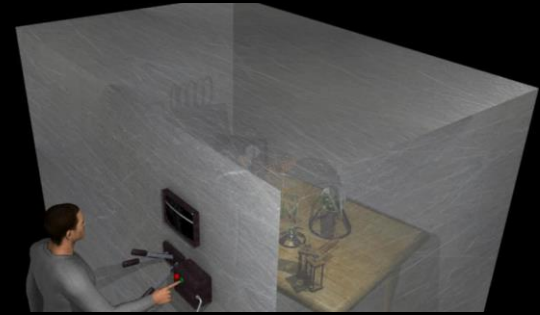


| Point2D                 |
|-------------------------|
| coord_x<br>coord_y      |
| translation(x,y)<br>... |

```
Point2D p;  
  
p.coord_x += x;  
p.coord_y += y;
```

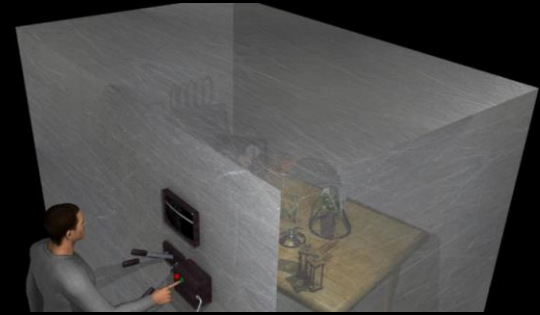
# Encapsulation

- Implémentation interne cachée
  - On ne manipule pas directement les attributs
  - On utilise les méthodes mises à disposition (interface)
- Exemple : translation d'un point 2D

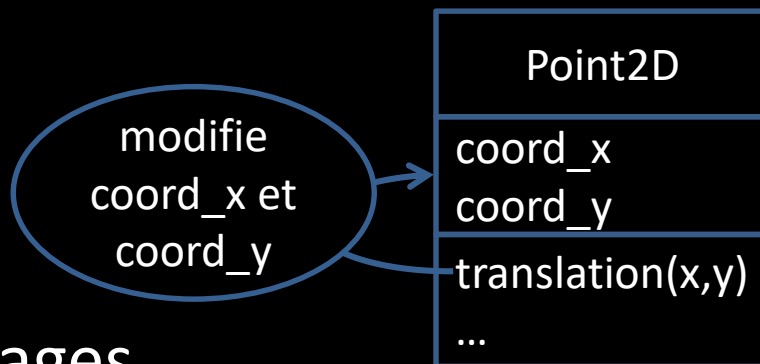


```
Point2D p;  
p.coord_x += x;  
p.coord_y += y;  
p.translation(x, y);
```

# Encapsulation



- Implémentation interne cachée
  - On ne manipule pas directement les attributs
  - On utilise les méthodes mises à disposition (interface)
- Exemple : translation d'un point 2D



```
Point2D p;  
p.coord_x += x;  
p.coord_y += y;  
p.translation(x, y);
```

- Avantages
  - Facilite la réutilisation, la maintenance et l'évolution du code
  - Le développeur ne se préoccupe pas de l'implémentation

# Objectifs de la POO

- Améliorer la vie du développeur !
- Conception plus intuitive
  - Notions de classe et d'objet
- Faciliter la réutilisation et l'évolution du code :
  - Encapsulation
  - Héritage
  - Polymorphisme



# Héritage (1)

- Forme de réutilisation du code :
  - Définition d'une classe à partir d'une classe existante
  - Déclinaison d'un concept général en concepts spécialisés



# Héritage (1)

- Forme de réutilisation du code :
  - Définition d'une classe à partir d'une classe existante
  - Déclinaison d'un concept général en concepts spécialisés

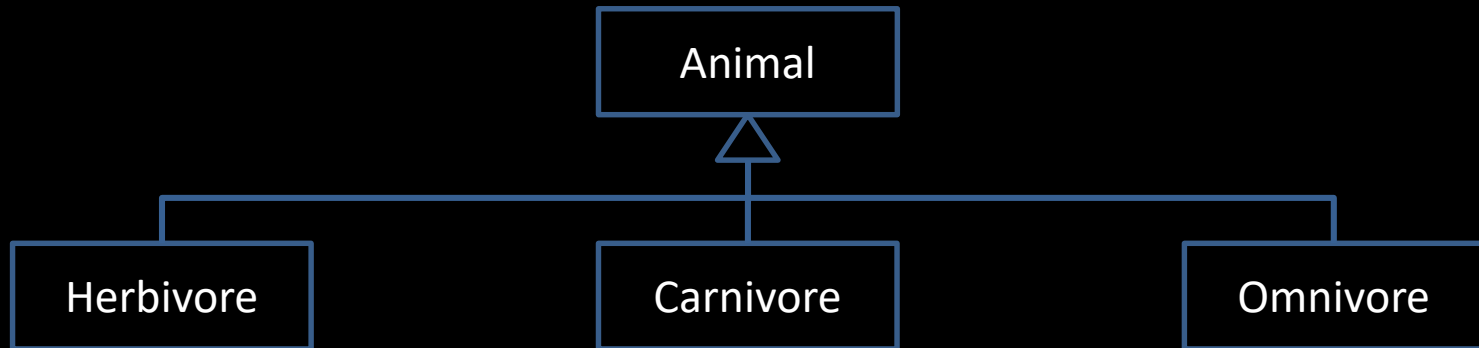


```
classDiagram
    class Animal
```

Animal

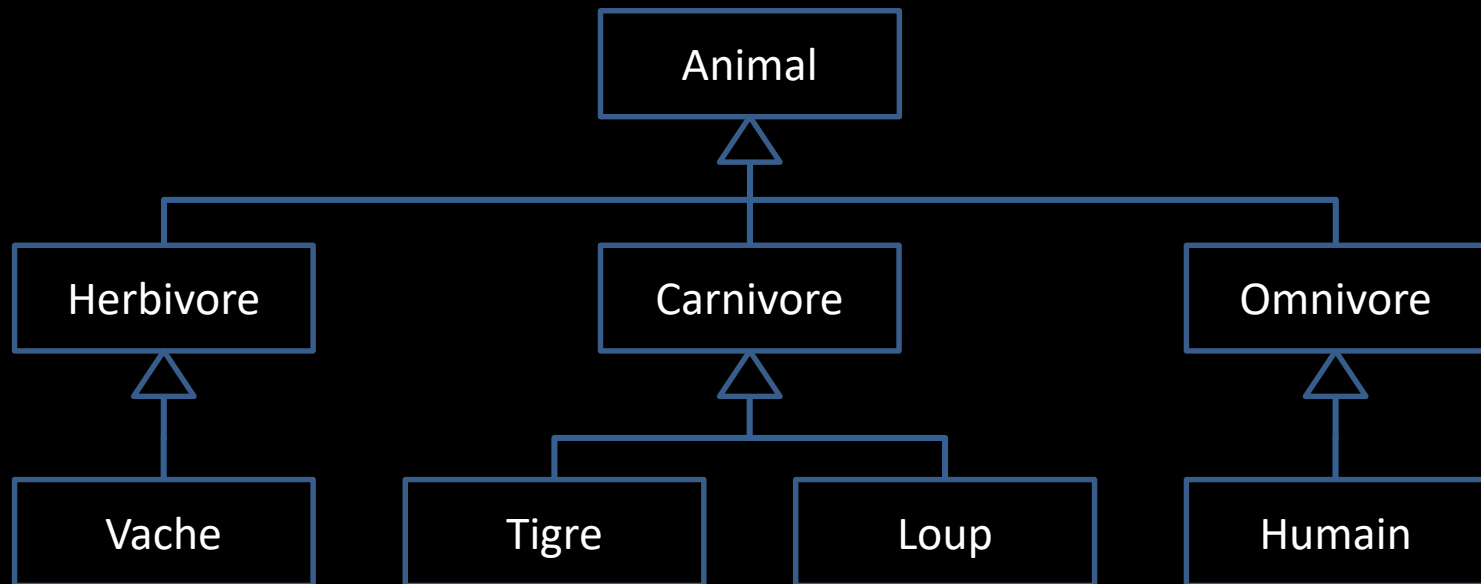
# Héritage (1)

- Forme de réutilisation du code :
  - Définition d'une classe à partir d'une classe existante
  - Déclinaison d'un concept général en concepts spécialisés



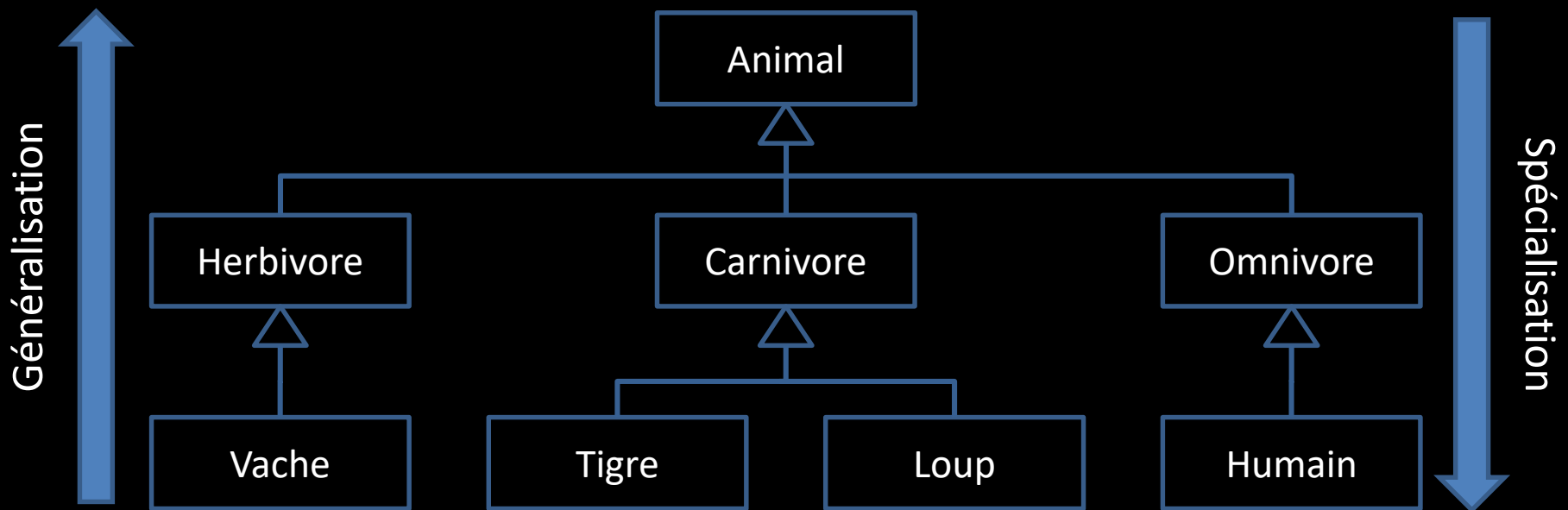
# Héritage (1)

- Forme de réutilisation du code :
  - Définition d'une classe à partir d'une classe existante
  - Déclinaison d'un concept général en concepts spécialisés
- Hiérarchie de classes



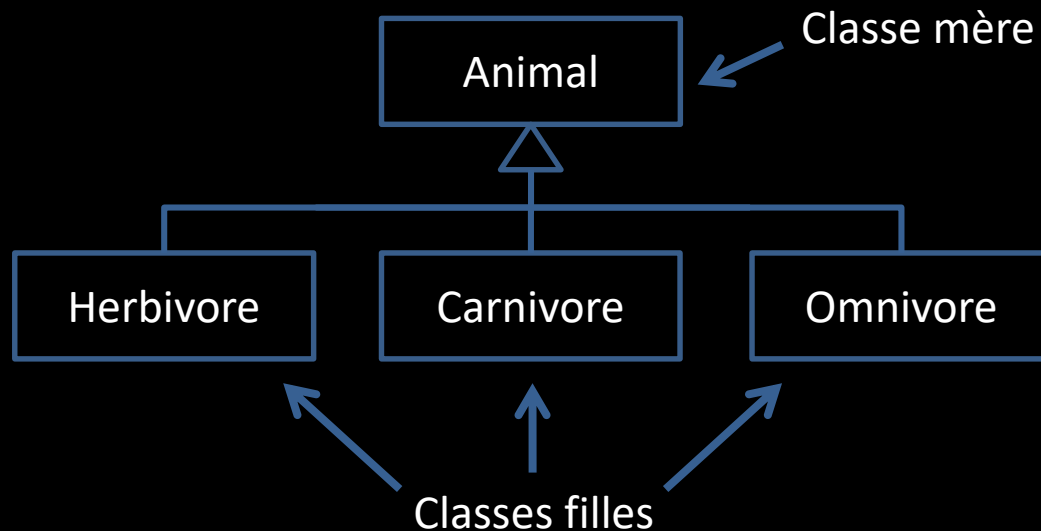
# Héritage (1)

- Forme de réutilisation du code :
  - Définition d'une classe à partir d'une classe existante
  - Déclinaison d'un concept général en concepts spécialisés
- Hiérarchie de classes



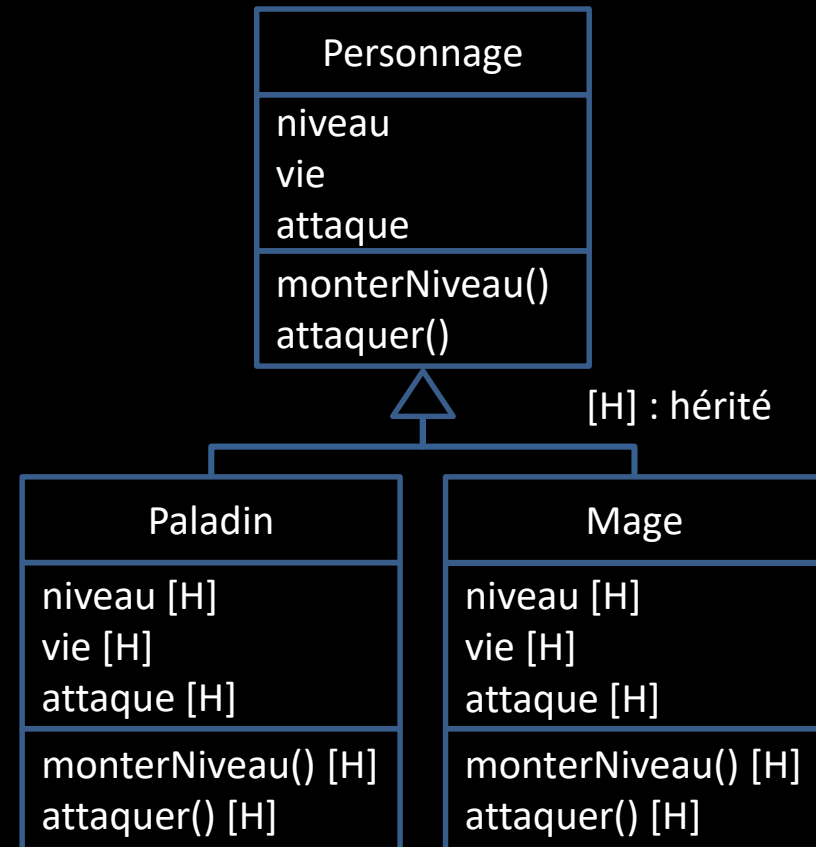
# Héritage (2)

- La classe fille  
classe dérivée  
sous-classe dérive / hérite de la classe mère  
classe de base  
super-classe



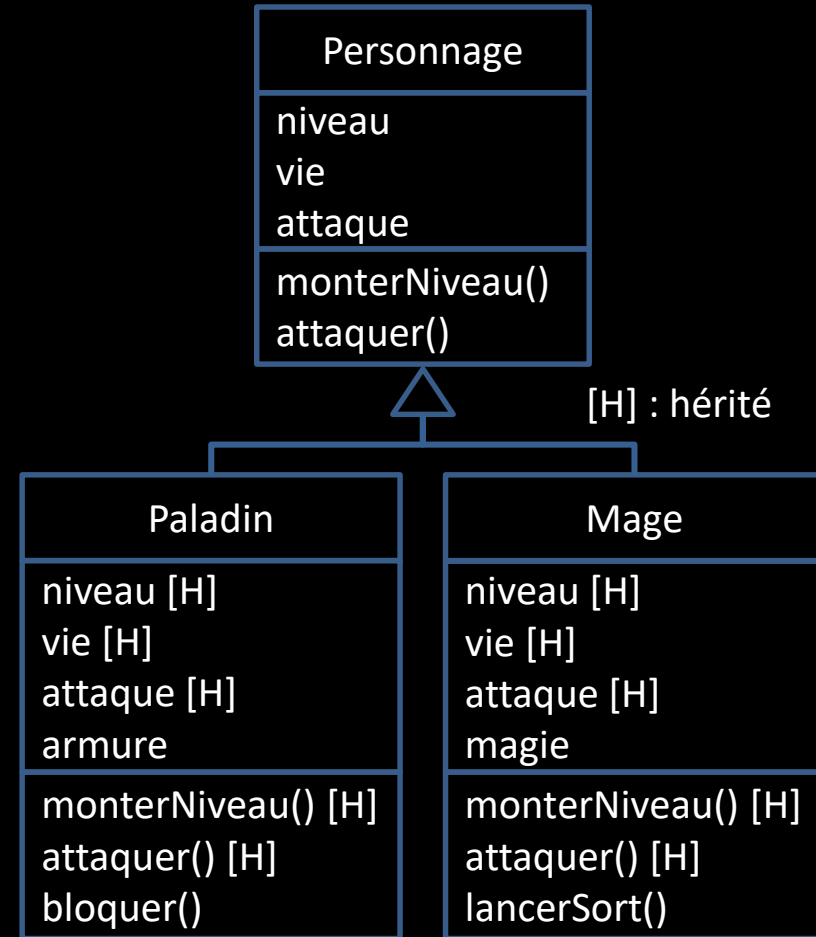
- La fille possède tous les attributs/méthodes de la mère

# Spécialisation d'une classe



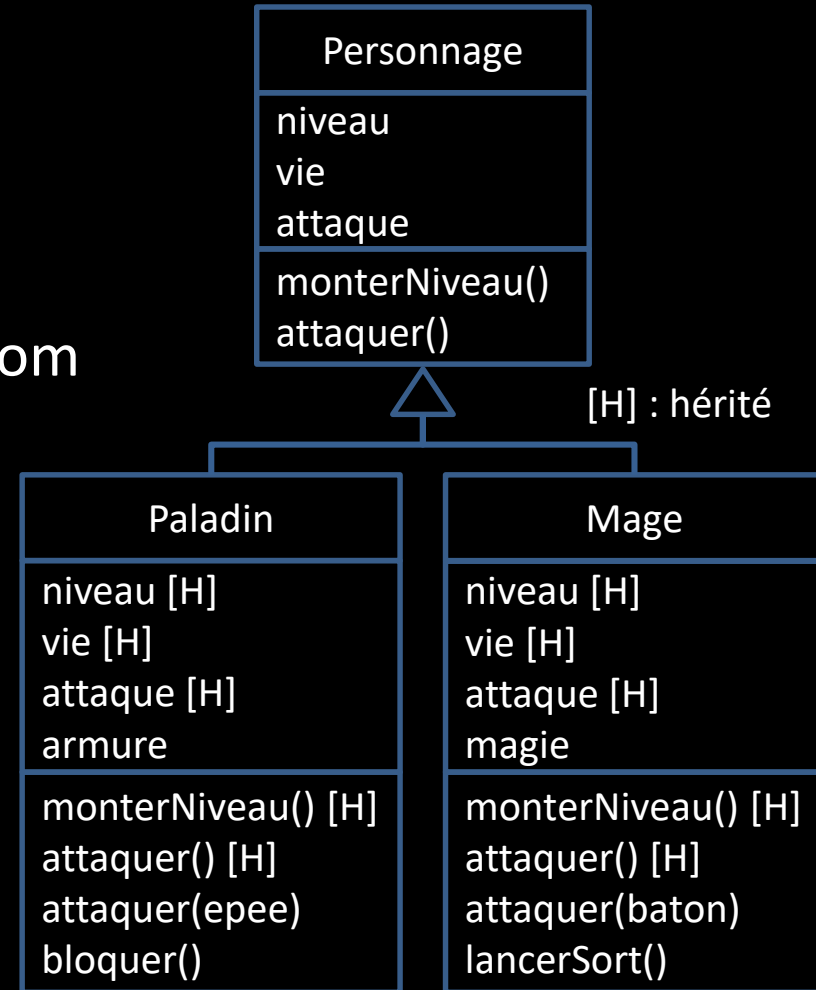
# Spécialisation d'une classe

- Extension :
  - Ajout d'attributs/méthodes



# Spécialisation d'une classe

- Extension :
  - Ajout d'attributs/méthodes
- Surcharge :
  - Ajout de méthodes de même nom
  - Signature différente





# Spécialisation d'une classe

- Extension :

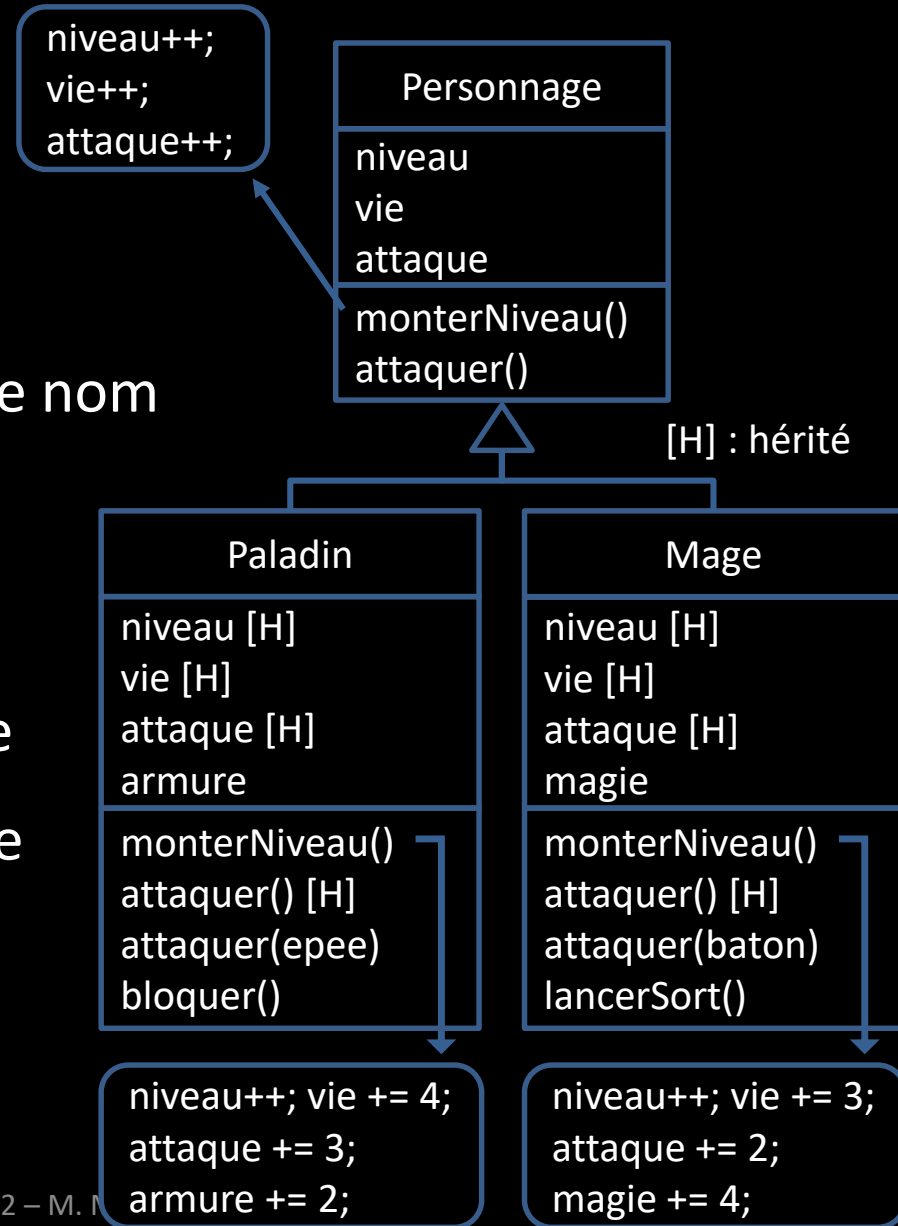
- Ajout d'attributs/méthodes

- Surcharge :

- Ajout de méthodes de même nom
- Signature différente

- Redéfinition :

- Modification d'une méthode
- Même nom, même signature
- Comportement différent

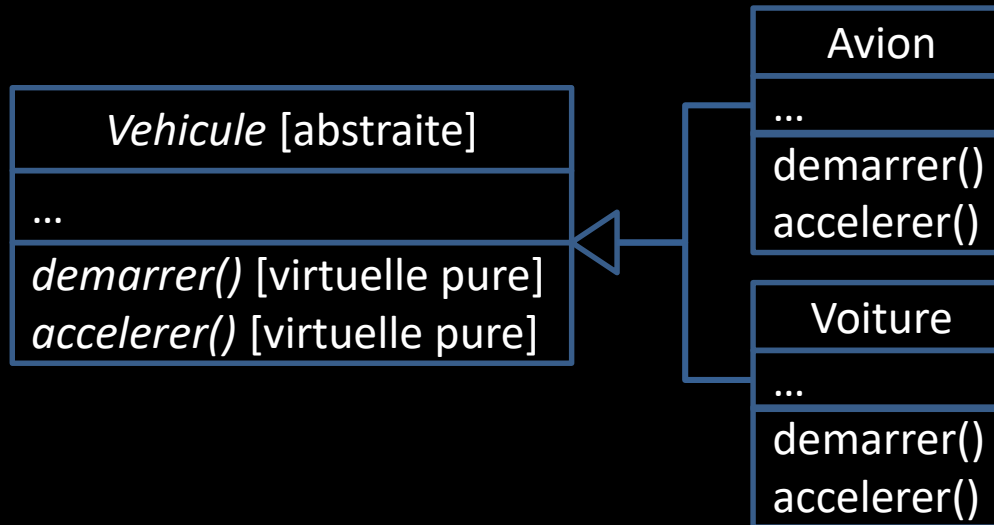


# Intérêts de l'héritage

- Simplifie la conception :
  - Hiérarchie de classes décrivant concrètement le problème
- Réutilisation du code :
  - S'appuyer sur des classes existantes, testées et validées
  - Factorisation du code (évite la redondance)
  - Gain de temps de développement
- Maintenance :
  - Corriger une erreur de la classe mère corrige la classe fille

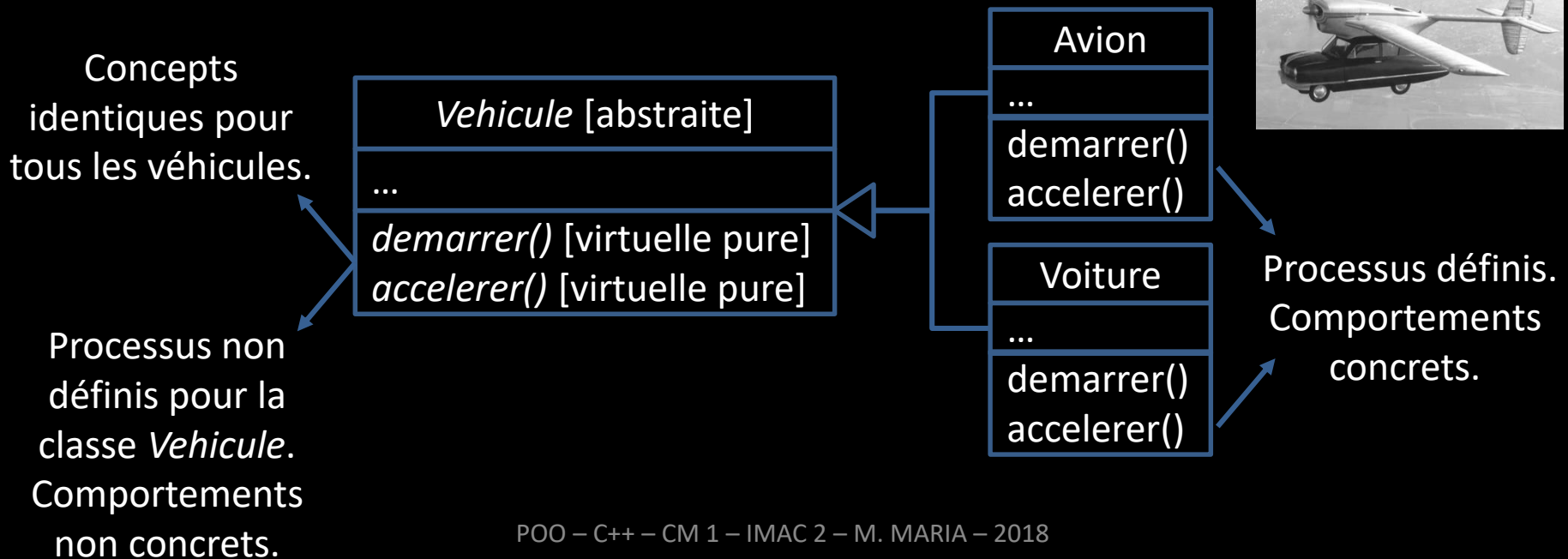
# Classes abstraites

- Classe non instanciable
  - Certaines méthodes non implémentées : virtuelles pures
  - Implémentées obligatoirement dans les classes filles
- Définir un concept pour toute la hiérarchie



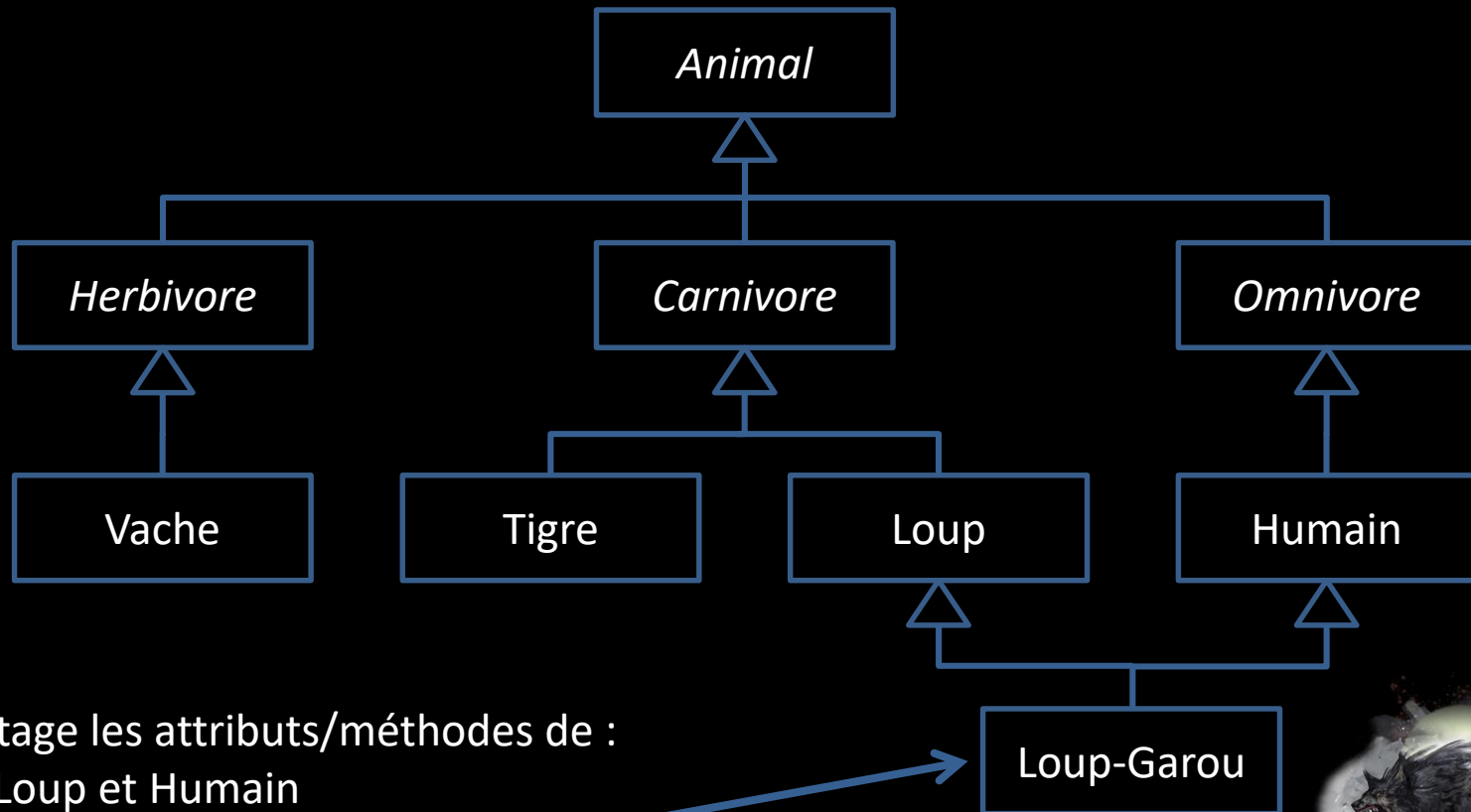
# Classes abstraites

- Classe non instanciable
  - Certaines méthodes non implémentées : virtuelles pures
  - Implémentées obligatoirement dans les classes filles
- Définir un concept pour toute la hiérarchie



# Héritage multiple (1)

- Une classe hérite de plusieurs classes mères



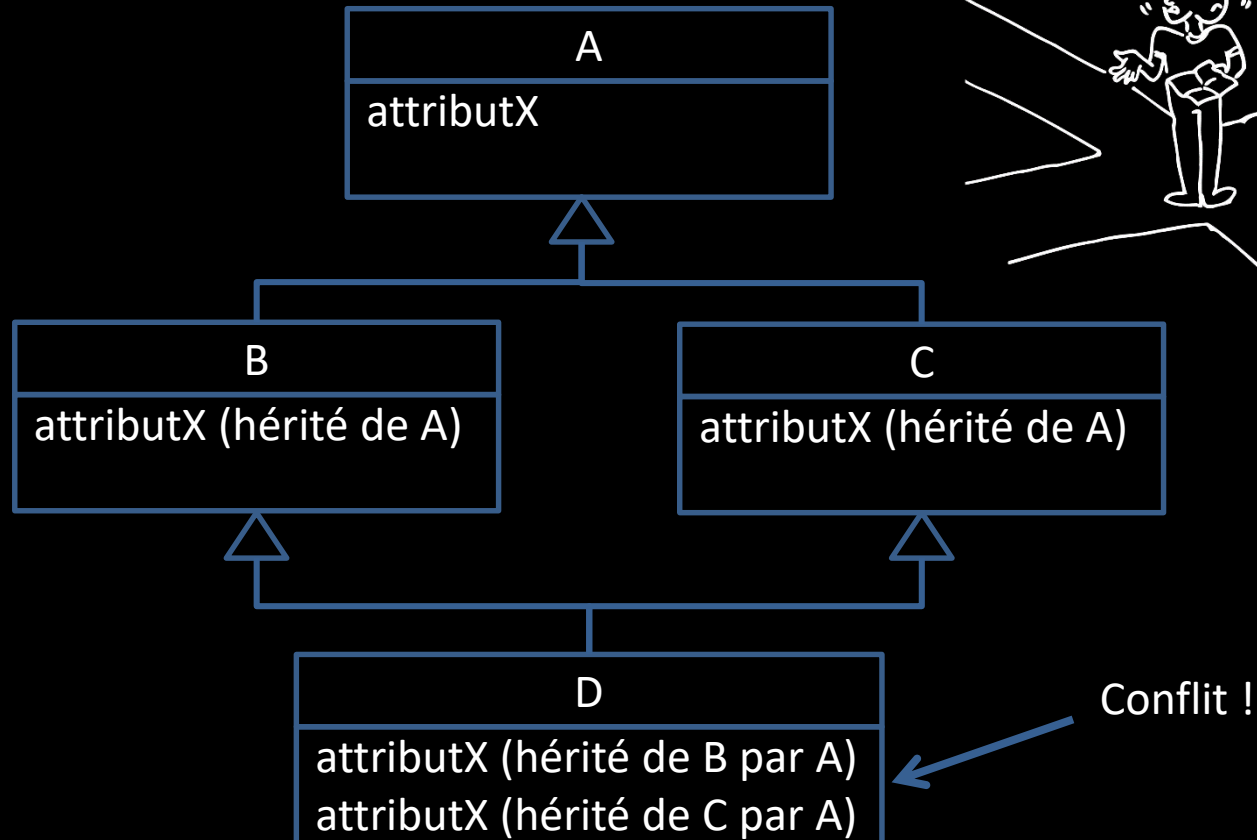
Partage les attributs/méthodes de :

- Loup et Humain
- Carnivore et Omnivore
- Animal



# Héritage multiple (2)

- Problématique (banni de certains langages *e.g.* Java)
  - Ex : Problème du diamant



# Objectifs de la POO

- Améliorer la vie du développeur !
- Conception plus intuitive
  - Notions de classe et d'objet
- Faciliter la réutilisation et l'évolution du code :
  - Encapsulation
  - Héritage
  - Polymorphisme



# Polymorphisme

- Vient du grec : *poly* = plusieurs / *morphê* = forme
- Littéralement :
  - Qui peut prendre plusieurs formes





# Polymorphisme

- Vient du grec : *poly* = plusieurs / *morphê* = forme
- Littéralement :
  - Qui peut prendre plusieurs formes
- En programmation :
  - Code fonctionnant différemment selon le type utilisé
  - *i.e.* comportement différent selon la situation



# Polymorphisme

- Vient du grec : *poly* = plusieurs / *morphê* = forme
- Littéralement :
  - Qui peut prendre plusieurs formes
- En programmation :
  - Code fonctionnant différemment selon le type utilisé
  - *i.e.* comportement différent selon la situation
- Deux grands types (vus ici) :
  - Polymorphisme ad hoc
  - Polymorphisme d'héritage



# Polymorphisme ad hoc

- Même nom, signatures différentes
- = Surcharge

## Une procédure/fonction

```
int addition(const int a, const int b)
{
    return a + b;
}

int addition(const int a, const int b, const int c)
{
    return a + b + c;
}
```

## Une méthode

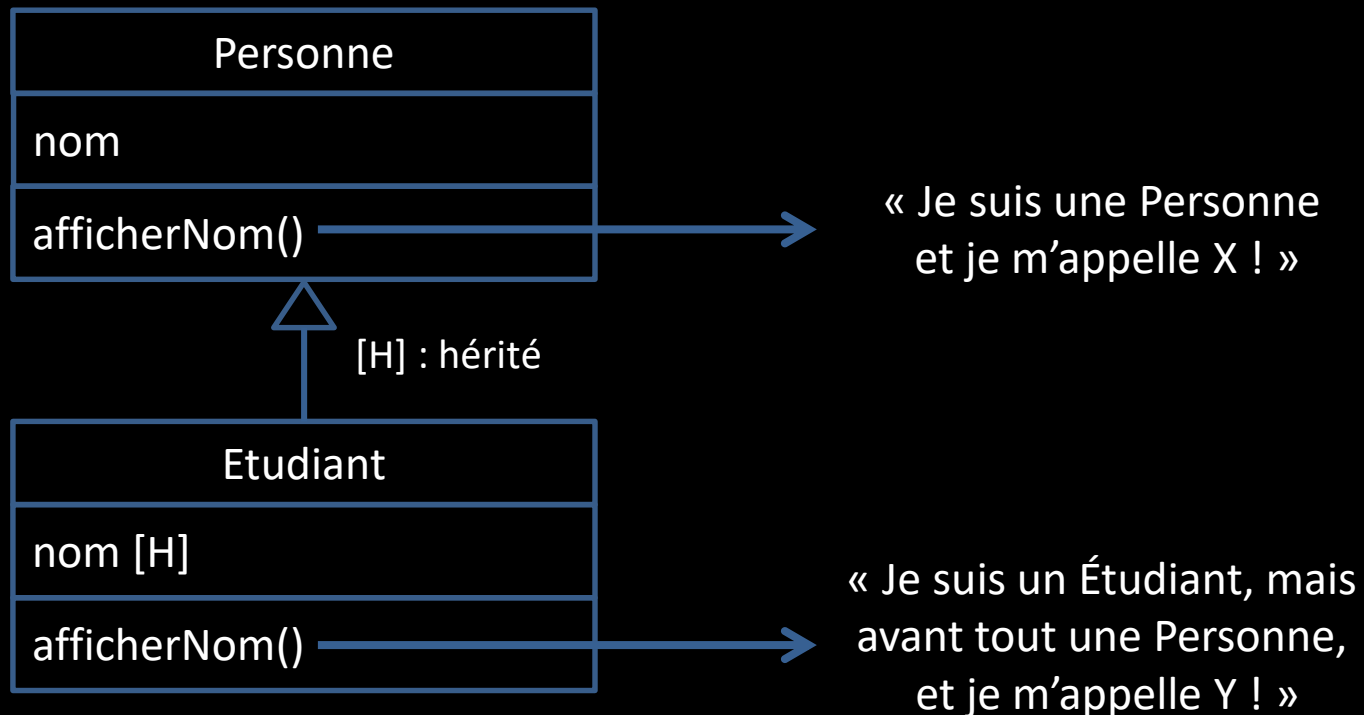
| Pikachu                       |
|-------------------------------|
| pv<br>attaque                 |
| eclair()<br>eclair(degatsSup) |

La méthode "eclair" est surchargée  
pour infliger des dégâts supplémentaires



# Polymorphisme d'héritage

- Modification d'une méthode dans un classe fille
  - Même nom, même signature, comportement différent.
- = Redéfinition



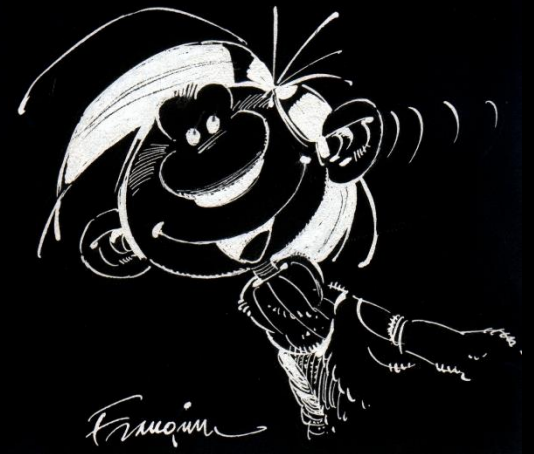
# PREMIERS PAS EN C++ (Pour le développeur C)

P'tite pause ?



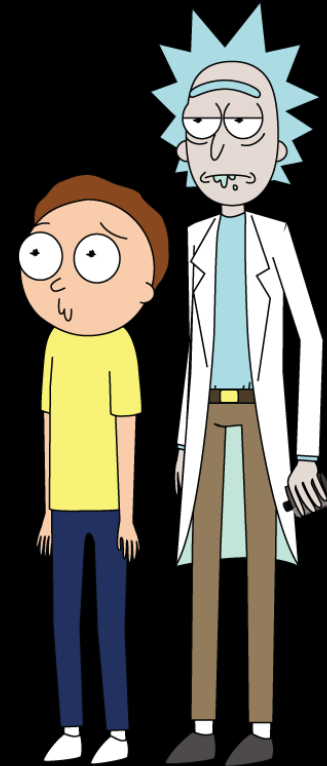
# PREMIERS PAS EN C++ (Pour le développeur C)

On est reparti !



# Notions

- Commentaires
- Les booléens
- Entrée/sorties
- Chaînes de caractères
- Les espaces de noms
- Allocation dynamique
- Les vecteurs
- Les références
- Les valeurs par défaut



À expérimenter en codant !

# Commentaires

```
int a; // Commentaire sur une ligne
```

```
/*  
Commentaires sur  
plusieurs lignes  
*/
```

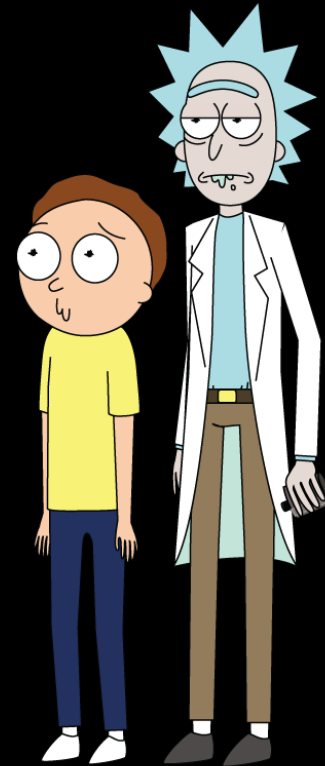


Des commentaires  
pertinents tu rédigeras !



# Notions

- Commentaires
- Les booléens
- Entrée/sorties
- Chaînes de caractères
- Les espaces de noms
- Allocation dynamique
- Les vecteurs
- Les références
- Les valeurs par défaut



À expérimenter en codant !

# Le type bool

- En C, pas de booléen (ou si en fait, dans stdbool.h)

```
typedef enum { false, true } bool;
```

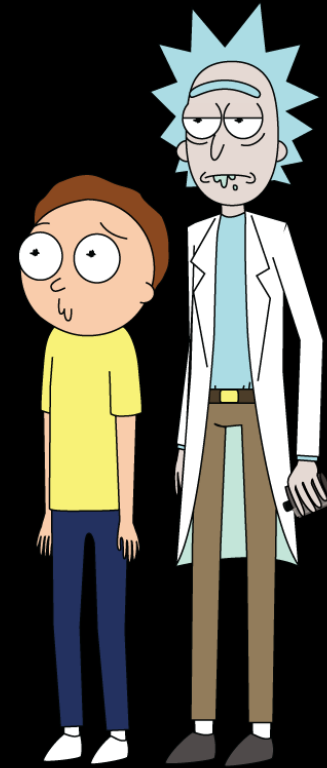
- En C++, les booléens existent !

```
bool boolean = true;  
  
if (boolean)  
{  
    boolean = false;  
}
```



# Notions

- Commentaires
- Les booléens
- **Entrée/sorties**
- Chaînes de caractères
- Les espaces de noms
- Allocation dynamique
- Les vecteurs
- Les références
- Les valeurs par défaut



À expérimenter en codant !

# Entrée/Sorties

- printf ? scanf ?

# Entrée/Sorties

- ~~printf ? scanf ?~~ Utilisation de flux !

# Entrée/Sorties

- ~~printf ? scanf ?~~ Utilisation de flux !

- Trois types de flux :

- std::cout : sortie standard (écran)
- std::cerr : sortie erreur (écran)
- std::cin : entrée standard (clavier)

```
#include <iostream>
```

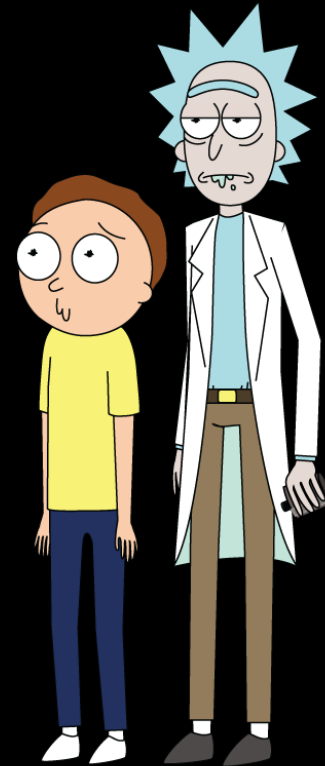
- S'utilisent avec << et >>

Fin de ligne

```
int i;  
std::cout << "Entrez un nombre" << std::endl;  
std::cin >> i;  
std::cout << "Le nombre est " << i << std::endl;  
std::cerr << "Une fausse erreur" << std::endl;
```

# Notions

- Commentaires
- Les booléens
- Entrée/sorties
- Chaînes de caractères
- Les espaces de noms
- Allocation dynamique
- Les vecteurs
- Les références
- Les valeurs par défaut



À expérimenter en codant !

# Chaînes de caractères

- En C : tableau de caractères (char)
- En C++: std::string !
  - Gestion mémoire dynamique
  - Plein de méthodes proposées

```
#include <string>
```

```
std::string nom = "Toto";  
std::string msg = "Salut " + nom;  
msg += "!";  
std::cout << msg << std::endl;
```

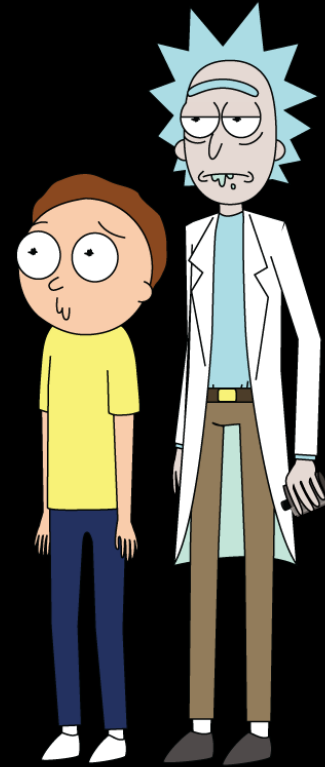
« Salut Toto! »





# Notions

- Commentaires
- Les booléens
- Entrée/sorties
- Chaînes de caractères
- Les espaces de noms
- Allocation dynamique
- Les vecteurs
- Les références
- Les valeurs par défaut



À expérimenter en codant !


# Les espaces de noms (namespace) (1)

- Zone de déclaration :
  - Évite les conflits de noms
  - Organise le code
- Portée restreinte au namespace
  - Variables/procédures/fonctions
- Appel extérieur via l'opérateur de résolution de portée ::
- À utiliser !

```
namespace A
{
    void doSomething()
    {
        // Do something
    }
}

namespace B
{
    void doSomething()
    {
        // Do something else
    }
}

A::doSomething();
B::doSomething();
```



# Les espaces de noms (namespace) (2)

- Donc "std::cout" est le flux "cout" du namespace "std" ?
- Et "std::string" la classe "string" du namespace "std" ?

# Les espaces de noms (namespace) (2)

- Donc "std::cout" est le flux "cout" du namespace "std" ?
- Et "std::string" la classe "string" du namespace "std" ?

**VRAI**

# Les espaces de noms (namespace) (2)

- Donc "std::cout" est le flux "cout" du namespace "std" ?
- Et "std::string" la classe "string" du namespace "std" ?

**VRAI**

- Et on est obligé d'écrire "std::" tout le temps ?

# Les espaces de noms (namespace) (2)

- Donc "std::cout" est le flux "cout" du namespace "std" ?
- Et "std::string" la classe "string" du namespace "std" ?

**VRAI**

- Et on est obligé d'écrire "std::" tout le temps ?

- NON :

```
using namespace std;  
  
string ok = "OK";  
cout << "ok" << endl;
```



# Les espaces de noms (namespace) (2)

- Donc "std::cout" est le flux "cout" du namespace "std" ?
- Et "std::string" la classe "string" du namespace "std" ?

**VRAI**

- Et on est obligé d'écrire "std::" tout le temps ?

- NON :

```
using namespace std;  
  
string ok = "OK";  
cout << "ok" << endl;
```



- Mais c'est MAL (surtout dans le header), alors OUI !
- C'est mon point de vue... (pour std je tolère)



# Les espaces de noms anonymes

- Permet de restreindre l'utilisation d'une fonction au fichier où elle est définie
  - Remplace le "static" du C

Pas de nom !  
(Anonyme quoi)

Uniquement  
accessible dans  
monFichier.cpp

```
#include <iostream>                                     monFichier.cpp

namespace
{
    void direBonjour()
    {
        std::cout << "Bonjour" << std::endl;
    }
}

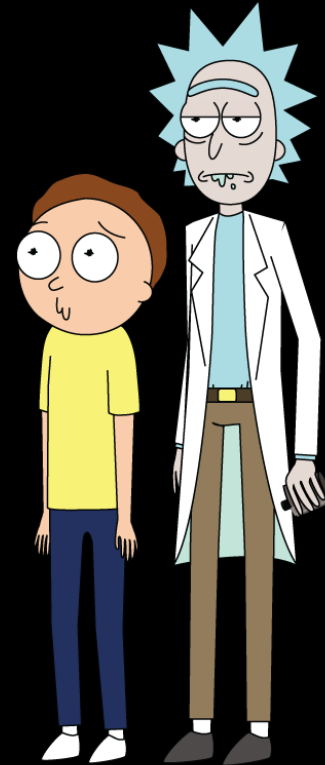
int main(int argc, char *argv[])
{
    direBonjour();
    return 0;
}
```





# Notions

- Commentaires
- Les booléens
- Entrée/sorties
- Chaînes de caractères
- Les espaces de noms
- **Allocation dynamique**
- Les vecteurs
- Les références
- Les valeurs par défaut



À expérimenter en codant !

# Allocation dynamique

- malloc ? free ?

```
/* Pointeur sur un entier en C */  
int *ptr_C = (int *)malloc(sizeof(int));  
/* Tableau de 10 entiers en C */  
int *tab_C = (int *)malloc(10 * sizeof(int));  
  
/* Libération mémoire en C */  
free(ptr_C);  
free(tab_C);
```

# Allocation dynamique

- ~~malloc ? free ?~~ Utilisation de new et delete !

```
/* Pointeur sur un entier en C */  
int *ptr_C = (int *)malloc(sizeof(int));  
/* Tableau de 10 entiers en C */  
int *tab_C = (int *)malloc(10 * sizeof(int));  
  
/* Libération mémoire en C */  
free(ptr_C);  
free(tab_C);
```

delete[]  
pour les tableaux !

```
// Pointeur sur un entier en C++  
int *ptr_CPP = new int;  
// Tableau de 10 entiers en C++  
int *tab_CPP = new int[10];  
  
// Libération mémoire en C++  
delete ptr_CPP;  
delete[] tab_CPP;
```

# Allocation dynamique

- ~~malloc ? free ?~~ Utilisation de new et delete !

```
/* Pointeur sur un entier  
int *ptr_C = (int *)malloc(4);  
/* Tableau de 10 entiers  
int *tab_C = (int *)malloc(10 * 4);  
/* Libérer la mémoire en C
```

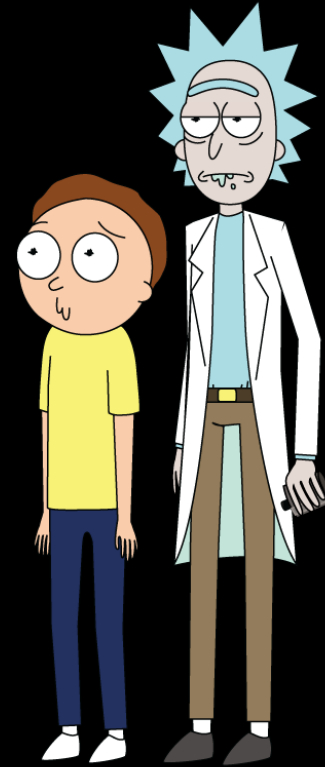
Si tu ne veux pas de problème, je te conseille de penser à libérer ta mémoire !

```
Pointeur sur un entier en C++  
int *ptr_CPP = new int;  
Tableau de 10 entiers en C++  
int *tab_CPP = new int[10];  
Libérer la mémoire en C++
```

delete[]  
pour les tableaux

# Notions

- Commentaires
- Les booléens
- Entrée/sorties
- Chaînes de caractères
- Les espaces de noms
- Allocation dynamique
- Les vecteurs
- Les références
- Les valeurs par défaut



À expérimenter en codant !

# Les vecteurs (un avant-goût de la STL)

- `std::vector` : sorte de tableau dynamique
- Gestion mémoire automatique

# Les vecteurs (un avant-goût de la STL)

- `std::vector` : sorte de tableau dynamique
- Gestion mémoire automatique

Type quelconque entre < >

`intVec = { }`

```
std::vector<int> intVec; ← Un vecteur d'entiers vide
```

# Les vecteurs (un avant-goût de la STL)

- `std::vector` : sorte de tableau dynamique
- Gestion mémoire automatique

Type quelconque entre < >

`intVec = { }`

```
std::vector<int> intVec;  ← Un vecteur d'entiers vide
std::cout << intVec.size() << std::endl;  ← Taille : .size()
                                     → « 0 »
```



# Les vecteurs (un avant-goût de la STL)

- `std::vector` : sorte de tableau dynamique
- Gestion mémoire automatique

Type quelconque entre < >

`intVec = { 79 }`

```
std::vector<int> intVec; ← Un vecteur d'entiers vide  
std::cout << intVec.size() << std::endl; ← Taille : .size()  
→ « 0 »  
intVec.push_back(79); ← Ajout par l'arrière
```

# Les vecteurs (un avant-goût de la STL)

- `std::vector` : sorte de tableau dynamique
- Gestion mémoire automatique

Type quelconque entre < >

`intVec = { 79, 17 }`

```
std::vector<int> intVec; ← Un vecteur d'entiers vide

std::cout << intVec.size() << std::endl; ← Taille : .size()
      → « 0 »

intVec.push_back(79); ← Ajout par l'arrière
intVec.push_back(17);
```

# Les vecteurs (un avant-goût de la STL)

- `std::vector` : sorte de tableau dynamique
- Gestion mémoire automatique

Type quelconque entre < >

`intVec = { 79, 17, 12 }`

```
std::vector<int> intVec; ← Un vecteur d'entiers vide

std::cout << intVec.size() << std::endl; ← Taille : .size()
      → « 0 »

intVec.push_back(79); ← Ajout par l'arrière
intVec.push_back(17);
intVec.push_back(12);

std::cout << intVec.size() << std::endl;
      → « 3 »
```

# Les vecteurs (un avant-goût de la STL)

- `std::vector` : sorte de tableau dynamique
- Gestion mémoire automatique

Type quelconque entre < >

`intVec = { 79, 17, 12 }`

```
std::vector<int> intVec; // Un vecteur d'entiers vide

std::cout << intVec.size() << std::endl; // Taille : .size()
//                                     → « 0 »

intVec.push_back(79); // Ajout par l'arrière
intVec.push_back(17);
intVec.push_back(12);

std::cout << intVec.size() << std::endl;
//                                     → « 3 »

std::cout << "Premier entier : " << intVec[0] << std::endl;
//                                     → « Premier entier : 79 »
```

Accès via [ ]

# Les vecteurs (un avant-goût de la STL)

- `std::vector` : sorte de tableau dynamique
- Gestion mémoire automatique

Type quelconque entre < > intVec = { 79, 17, 12 }

```
std::vector<int> intVec; // Un vecteur d'entiers vide
```

`std::cout << intVec.size() << std::endl;` Taille : .size()  
→ « 0 »

```
intVec.push_back(79); // Ajout par l'arrière
intVec.push_back(17);
intVec.push_back(12);
```

`std::cout << intVec.size() << std::endl;` → « 3 »

`std::cout << "Premier entier : " << intVec[0] << std::endl;` Accès via [ ]  
→ « Premier entier : 79 »

```
int *ptrIntVec = intVec.data();
```

Pointeur vers les données : .data()

# Les vecteurs (un avant-goût de la STL)

- `std::vector` : sorte de tableau dynamique
- Gestion mémoire automatique

PUISSANT NON ?!



Type quelconque entre < >

`intVec = { 79, 17, 12 }`

```
std::vector<int> intVec; // Un vecteur d'entiers vide

std::cout << intVec.size() << std::endl; // Taille : .size()
// « 0 »

intVec.push_back(79); // Ajout par l'arrière
intVec.push_back(17);
intVec.push_back(12);

std::cout << intVec.size() << std::endl;
// « 3 »

std::cout << "Premier entier : " << intVec[0] << std::endl;
// « Premier entier : 79 »

int *ptrIntVec = intVec.data();
```

Accès via [ ]

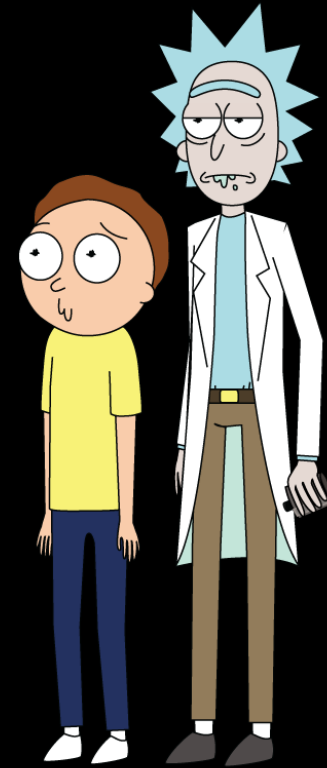
Pointeur vers les données : `.data()`

# La STL : *Standard Template Library*

- Normalisée ISO
- Ensemble de conteneurs (vecteur, liste, pile...)
  - Dynamiques
  - Mécanismes de parcours (itérateurs)
  - Algorithmes (recherche, tri, ...)
- Détails dans le 4<sup>ème</sup> CM ! Il est encore trop tôt...

# Notions

- Commentaires
- Les booléens
- Entrée/sorties
- Chaînes de caractères
- Les espaces de noms
- Allocation dynamique
- Les vecteurs
- Les références
- Les valeurs par défaut



À expérimenter en codant !



# Les références

- Référence (déclaré par &) =
  - Alias/identificateur/autre nom d'une variable
  - Fonctionne comme un pointeur

C

```
int a = 12;      /* un entier */  
int *pt = &a;    /* un pointeur sur a */  
int b = *pt + 4; /* équivaut à b = a + 4 */
```

C++

```
int a = 12;      // un entier  
int &ref = a;    // une référence sur a  
int b = ref + 4; // équivaut à b = a + 4
```



# Les références

- Référence (déclaré par &) =
  - Alias/identificateur/autre nom d'une variable
  - Fonctionne comme un pointeur

C

```
int a = 12;      /* un entier */
int *pt = &a;    /* un pointeur sur a */
int b = *pt + 4; /* équivaut à b = a + 4 */
```

C++

```
int a = 12;      // un entier
int &ref = a;    // une référence sur a
int b = ref + 4; // équivaut à b = a + 4
```

- Différence avec les pointeurs
  - Ne peut être nulle : initialisation à la déclaration
  - La variable référencée ne peut être changée
    - Mais son contenu oui (sauf si const)



# Passage d'arguments par référence (1)

- Paramètres modifiables
  - En C : via pointeur

Via pointeur

C

```
void setIntTo2(int *x)
{
    *x = 2;
}

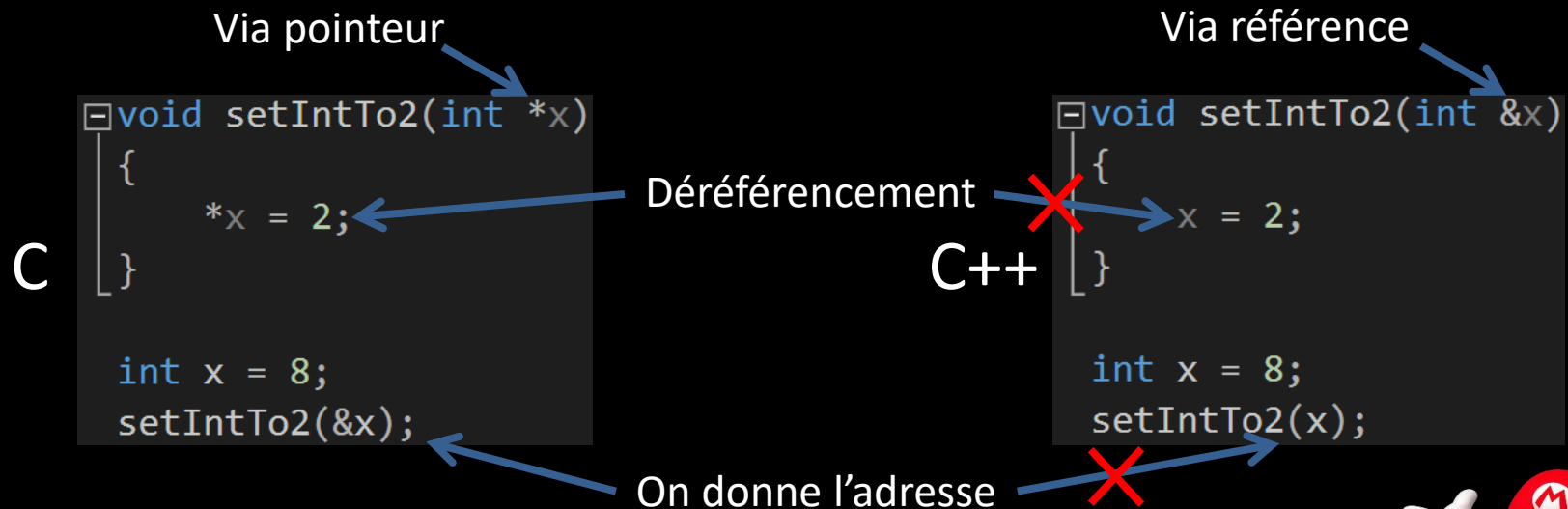
int x = 8;
setIntTo2(&x);
```

Déréférencement

On donne l'adresse

# Passage d'arguments par référence (1)

- Paramètres modifiables
  - En C : via pointeur / En C++ : via référence



TADAAA!



# Passage d'arguments par référence (2)

- Variables lourdes à copier
  - Pour des raisons de performances

```
void function(const GrosseStruct s);  
void function(const GrosseStruct &s);
```

"s" est copiée :  
possiblement lent...

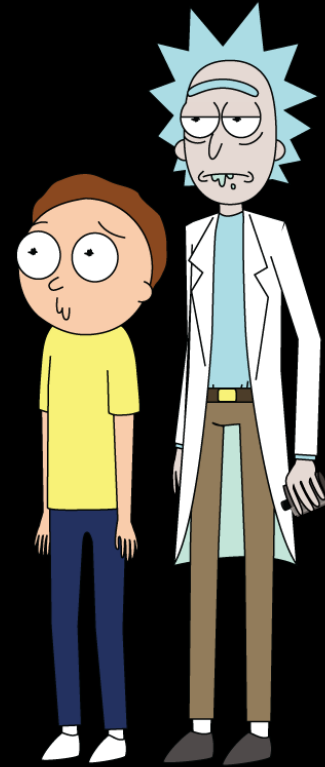
Empêche la modification

Évite la copie, on passe une référence

**V.I.P.**  
*Very Important Practice*

# Notions

- Commentaires
- Les booléens
- Entrée/sorties
- Chaînes de caractères
- Les espaces de noms
- Allocation dynamique
- Les vecteurs
- Les références
- Les valeurs par défaut



À expérimenter en codant !

# Les valeurs par défaut

- Arguments de fonction avec valeur par défaut
  - Déclaration uniquement dans le prototype

```
int convertToSeconds(int hours, int minutes = 0, int seconds = 0);  
  
int convertToSeconds(int hours, int minutes, int seconds)  
{  
    return hours * 3600 + minutes * 60 + seconds;  
}  
  
int a = convertToSeconds(1);           // a = 3600  
int b = convertToSeconds(1, 2);        // b = 3720  
int c = convertToSeconds(1, 2, 3);     // c = 3723
```



- Les valeurs par défaut sont obligatoirement à la fin
  - Pour la signature et l'appel

# TERMINÉ POUR CE MATIN !

## On se voit cette après-midi en Synthèse d'Images 2 !

