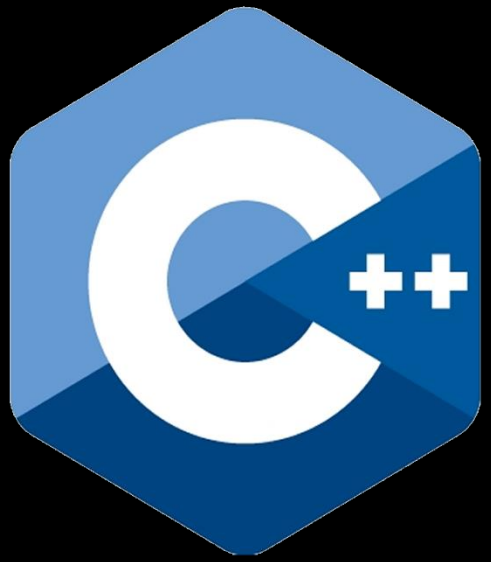


Programmation Orientée Objet

Maxime MARIA

maxime.maria@u-pem.fr

UNIVERSITÉ PARIS-EST MARNE-LA-VALLÉE | Esipe INGÉNIEURS PARIS EST

imac ingenieur

HÉRITAGE ET POLYMORPHISME

(ET TRANSTYPAGE)

Notions

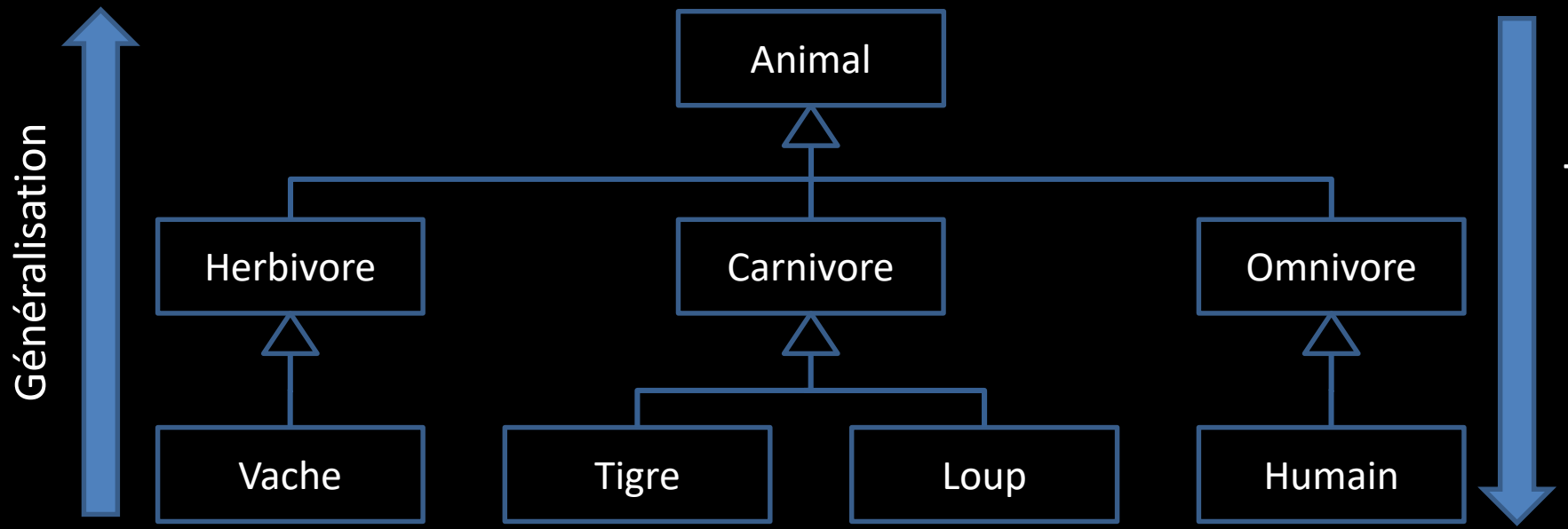
- Héritage
- Polymorphisme
- Transtypage (conversions de types)

Notions

- Héritage :
  - Rappels
  - L'héritage en C++
  - Héritage et constructeurs
  - Héritage et visibilité
  - Redéfinition de méthodes
- Polymorphisme
- Transtypage (conversions de types)

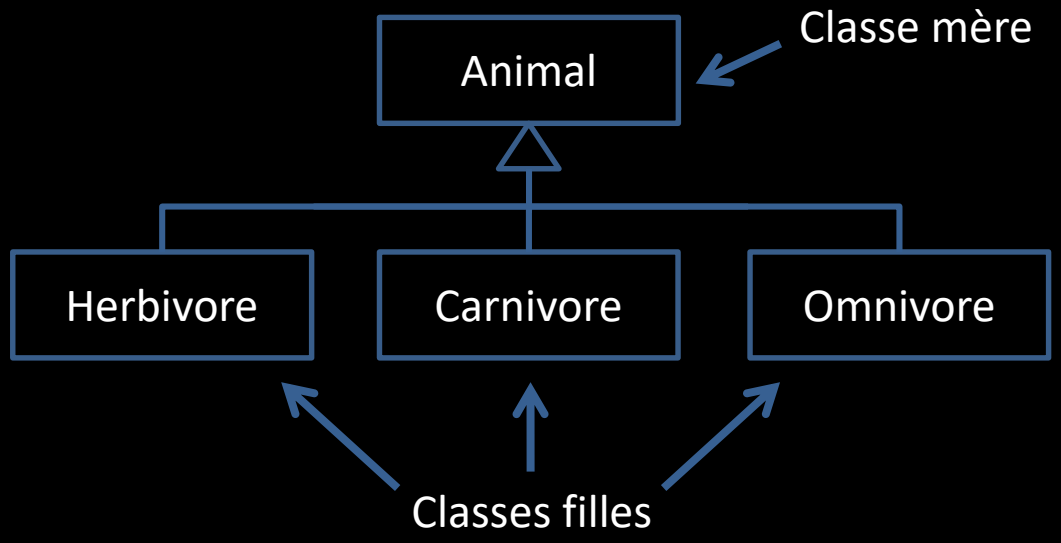
Héritage (1)

- 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



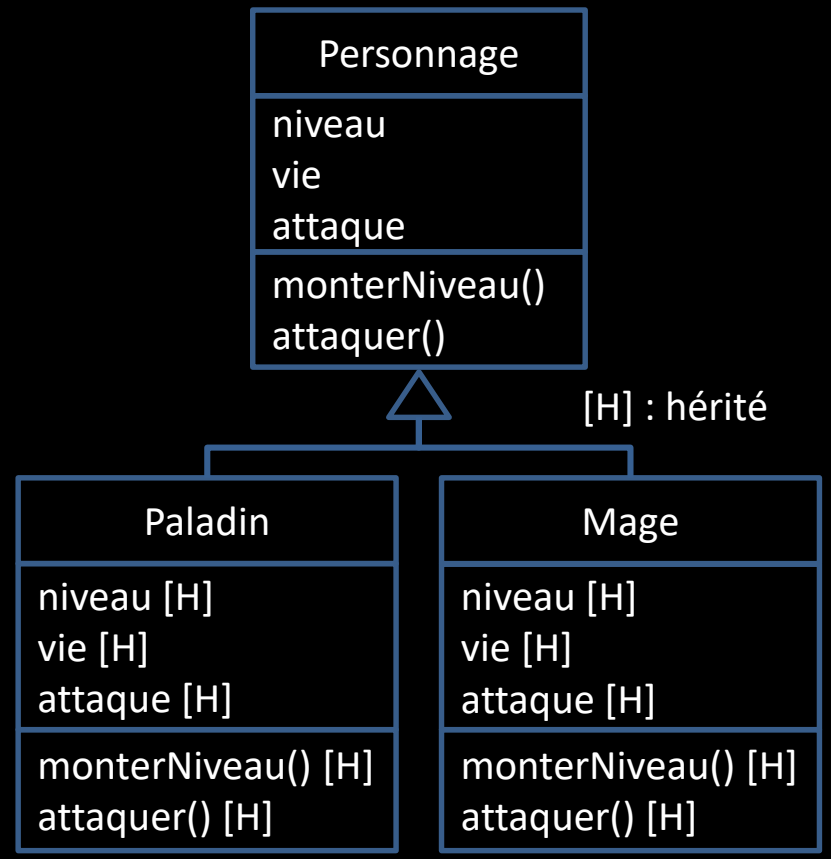
Héritage (2)

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



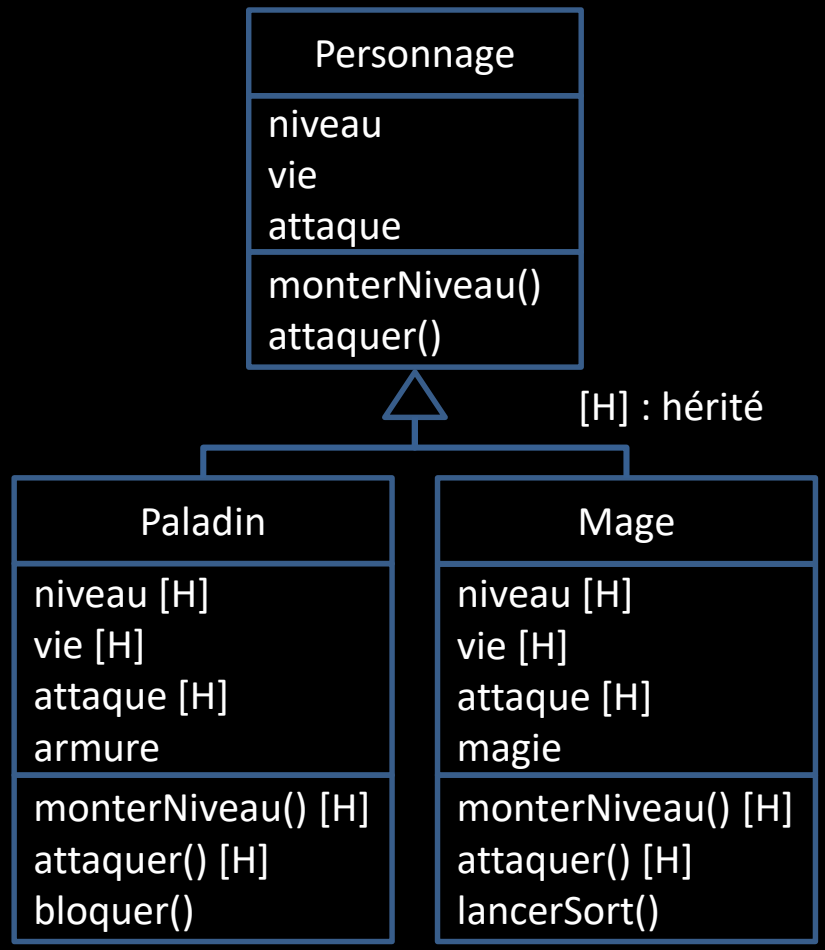
- La fille possède tous les attributs/méthodes de la mère

Spécialisation d'une classe



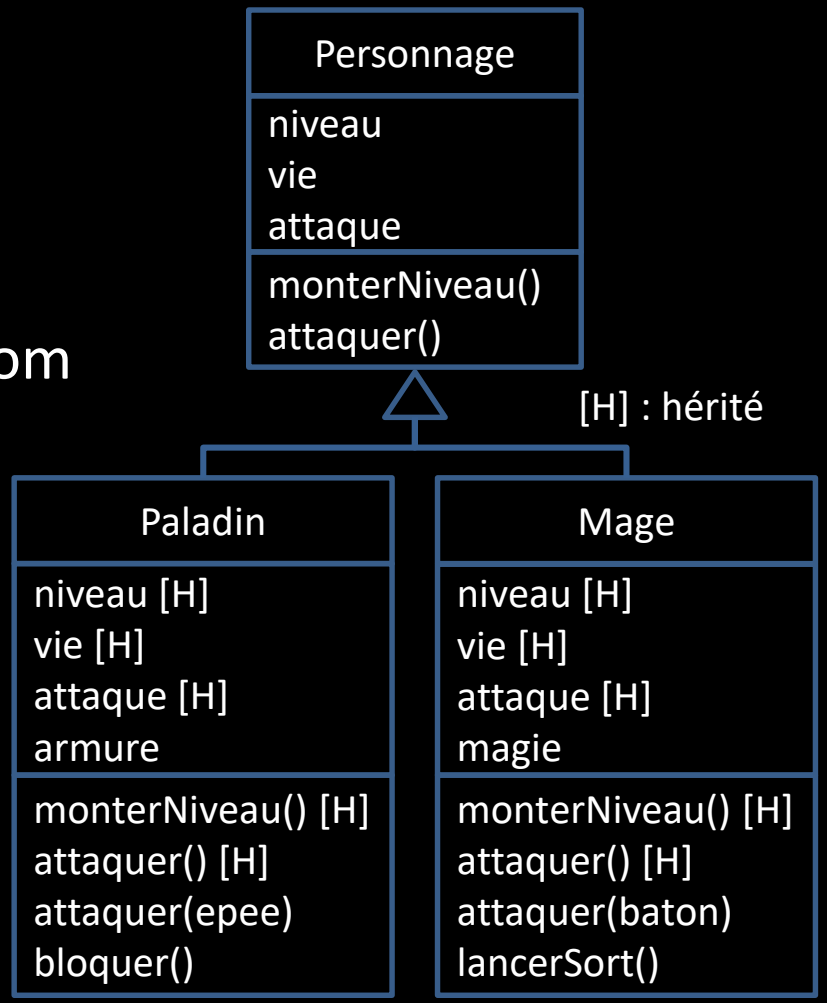
Spécialisation d'une classe

- Extension :
  - Ajout d'attributs/méthodes



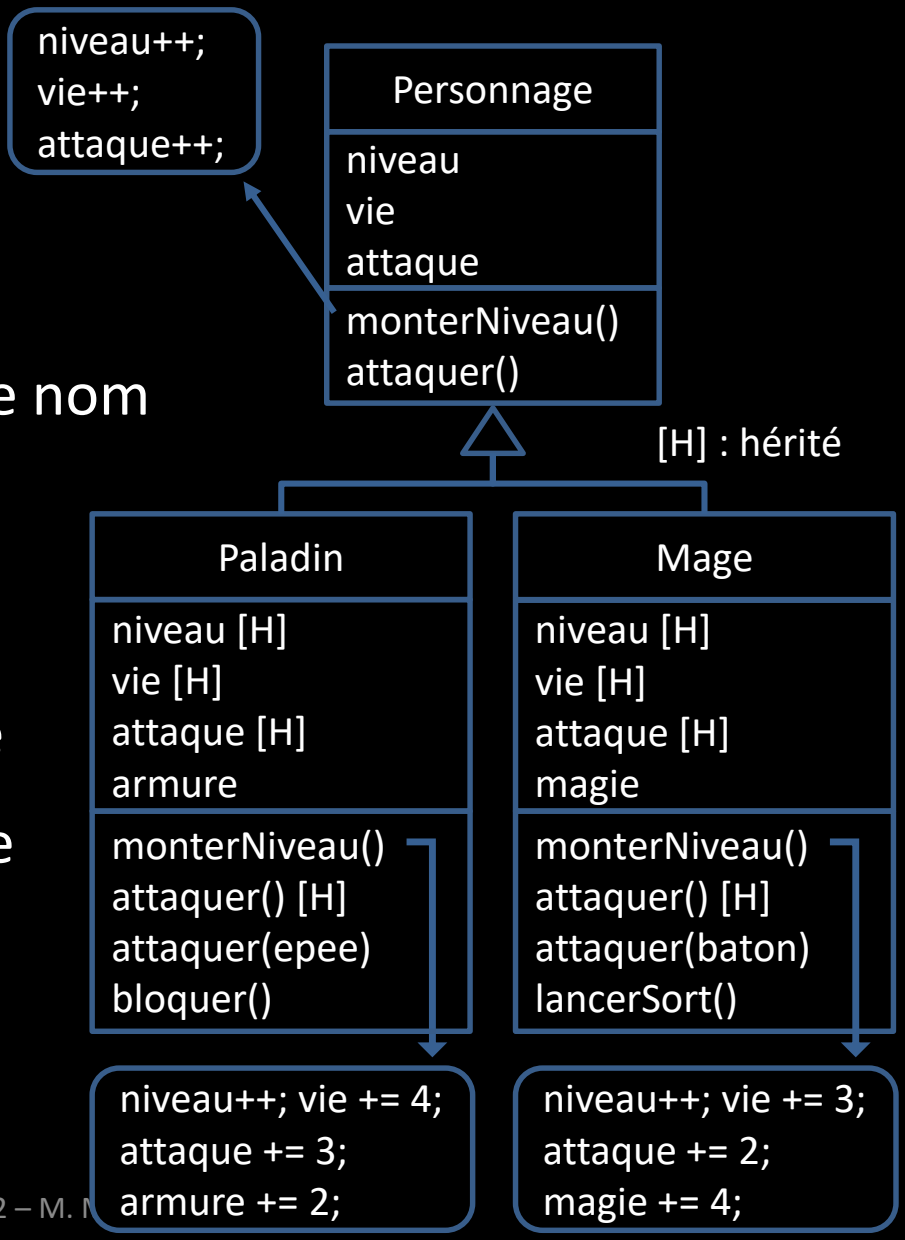
Spécialisation d'une classe

- 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



Notions

- Héritage :
  - Rappels
  - L’héritage en C++
  - Héritage et constructeurs
  - Héritage et visibilité
  - Redéfinition de méthodes
- Polymorphisme
- Transtypage (conversions de types)

Syntaxe générale

- Déclaration d'héritage
  - À l'aide des deux points ":"

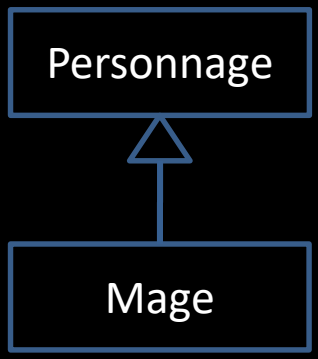
```
class ClasseMere
{
    // Attributs/Méthodes
};
```

Visibilité d'héritage (nous verrons ça plus tard)

```
class ClasseFille : public ClasseMere
{
    // Attributs/Méthodes supplémentaires
};
```

Notre exemple

```
class Personnage
{
public:
    Personnage() :  m_nom("John Doe"),
                    m_vie(100),
                    m_attaque(10) {}

    void attaquer(Personnage &ennemi) const
    {
        ennemi.encaisser(m_attaque);
    }

    void encaisser(const int degats)
    {
        m_vie -= degats;
    }

private:
    std::string m_nom;
    int m_vie;
    int m_attaque;
};
```

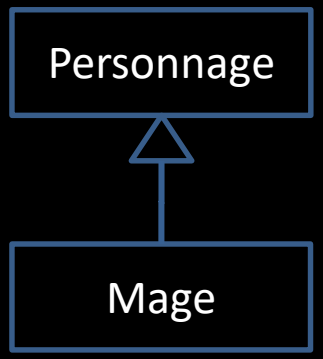



```
class Mage : public Personnage
{
public:
    void lancerSort();

private:
    int m_mana;
};
```

Partage les membres de Personnage
Pas besoin de les réécrire !

Type hérité

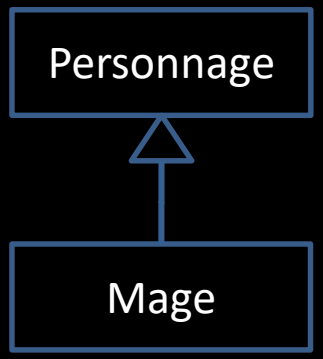
- Si B hérite de A :
  - Les instances de B sont aussi des instances de A
  - Conversion automatique !



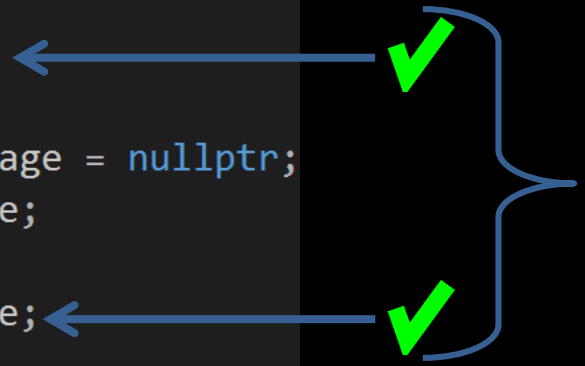

```
Personnage unPersonnage;
Mage unMage;

unPersonnage = unMage;
```

Type hérité

- Si B hérite de A :
  - Les instances de B sont aussi des instances de A
  - Conversion automatique !



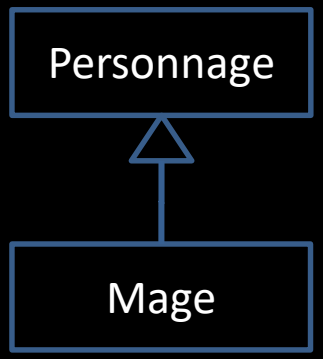

```
Personnage unPersonnage;
Mage unMage;

unPersonnage = unMage;

Personnage *ptrPersonnage = nullptr;
Mage *ptrMage = &unMage;

ptrPersonnage = ptrMage;
```



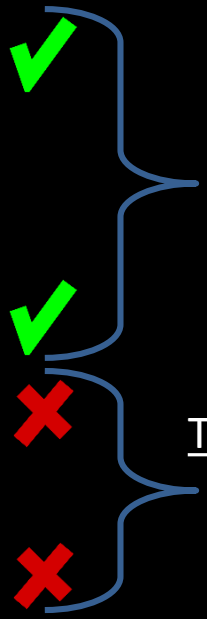
Transtypage ascendant (upcasting) automatique

Type hérité

- Si B hérite de A :
  - Les instances de B sont aussi des instances de A
  - Conversion automatique !




```
Personnage unPersonnage;
Mage unMage;

unPersonnage = unMage;

Personnage *ptrPersonnage = nullptr;
Mage *ptrMage = &unMage;

ptrPersonnage = ptrMage;

unMage = unPersonnage;

ptrPersonnage = &unPersonnage;
ptrMage = ptrPersonnage;
```

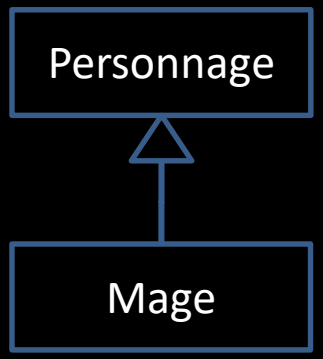


Transtypage ascendant (upcasting) automatique

Transtypage descendant (downcasting)
Non automatique, à spécifier
(Nous verrons plus tard)

Type hérité

- Si B hérite de A :
  - Les instances de B sont aussi des instances de A
  - Conversion automatique !




```
Personnage unPersonnage;
Mage unMage;

unPersonnage = unMage;

Personnage *ptrPersonnage = nullptr;
Mage *ptrMage = &unMage;

ptrPersonnage = ptrMage;

unMage = unPersonnage;

ptrPersonnage = &unPersonnage;
ptrMage = ptrPersonnage;
```

Pointeur nul en C++ 11

Transtypage ascendant (upcasting) automatique

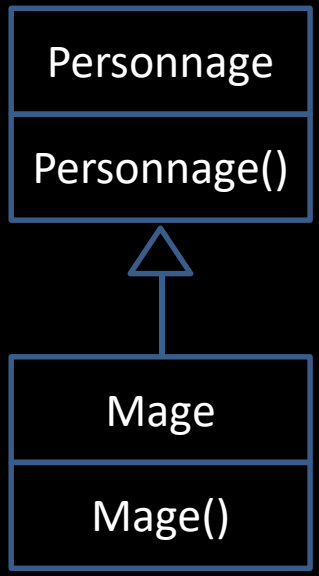
Transtypage descendant (downcasting) Non automatique, à spécifier (Nous verrons plus tard)

Notions

- Héritage :
  - Rappels
  - L'héritage en C++
  - Héritage et constructeurs
  - Héritage et visibilité
  - Redéfinition de méthodes
- Polymorphisme
- Transtypage (conversions de types)

Appels de constructeurs (1)

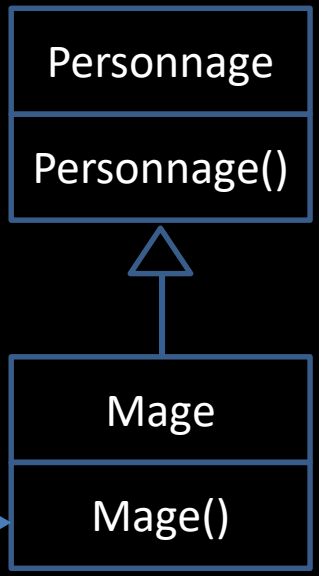
- Construction fille : appel constructeur mère




```
Mage gandalf;

gandalf.lancerSort();
```

Appels de constructeurs (1)

- Construction fille : appel constructeur mère



[1]
Création d'un Mage :
appel du constructeur

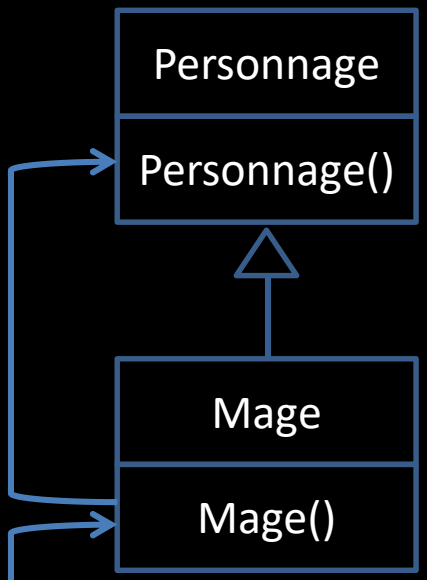
```
Mage gandalf;

gandalf.lancerSort();
```

Appels de constructeurs (1)

- Construction fille : appel constructeur mère

[2]
Constructeur de Mage appelle le constructeur de Personnage

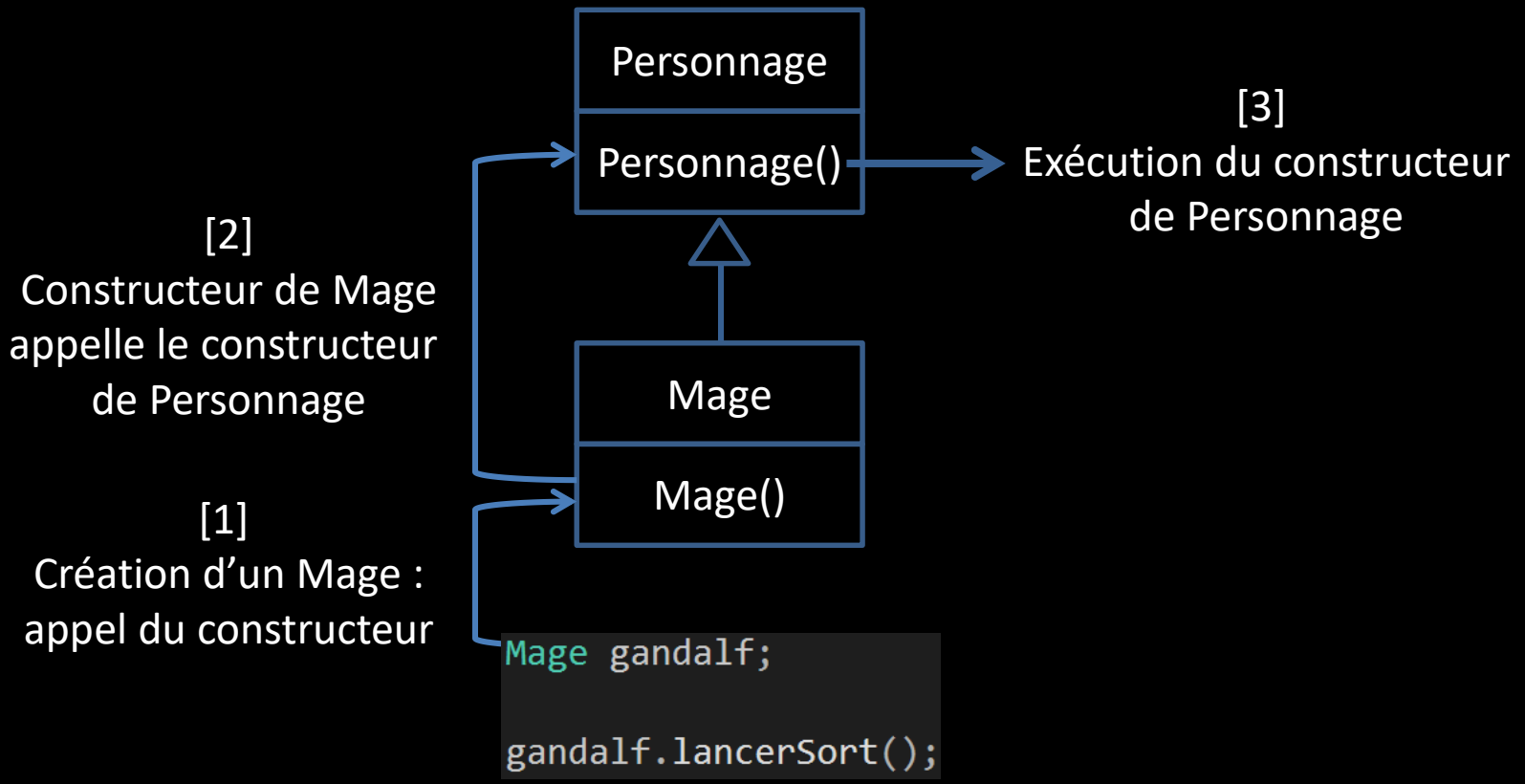
[1]
Création d'un Mage : appel du constructeur




```
Mage gandalf;

gandalf.lancerSort();
```

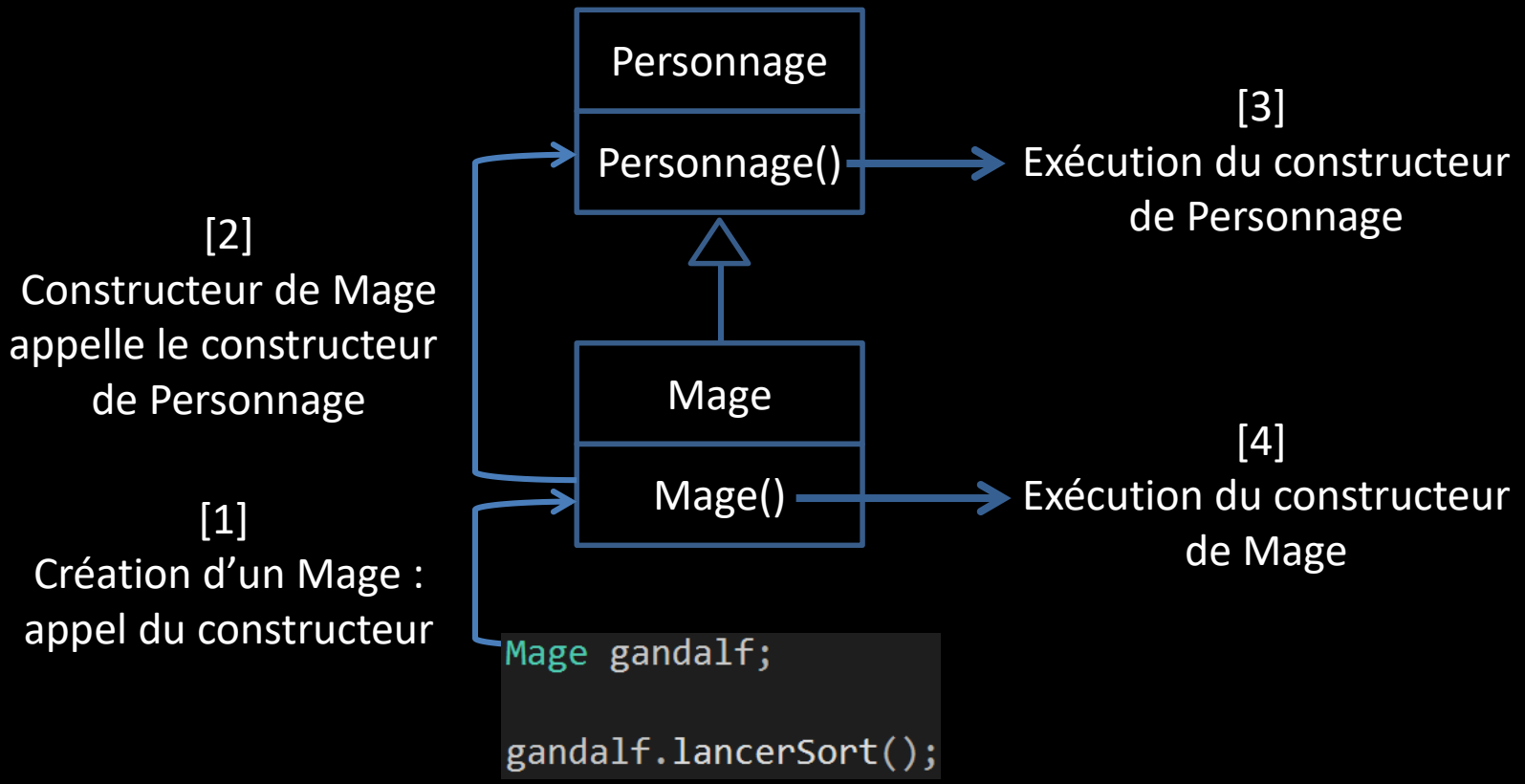
Appels de constructeurs (1)

- Construction fille : appel constructeur mère



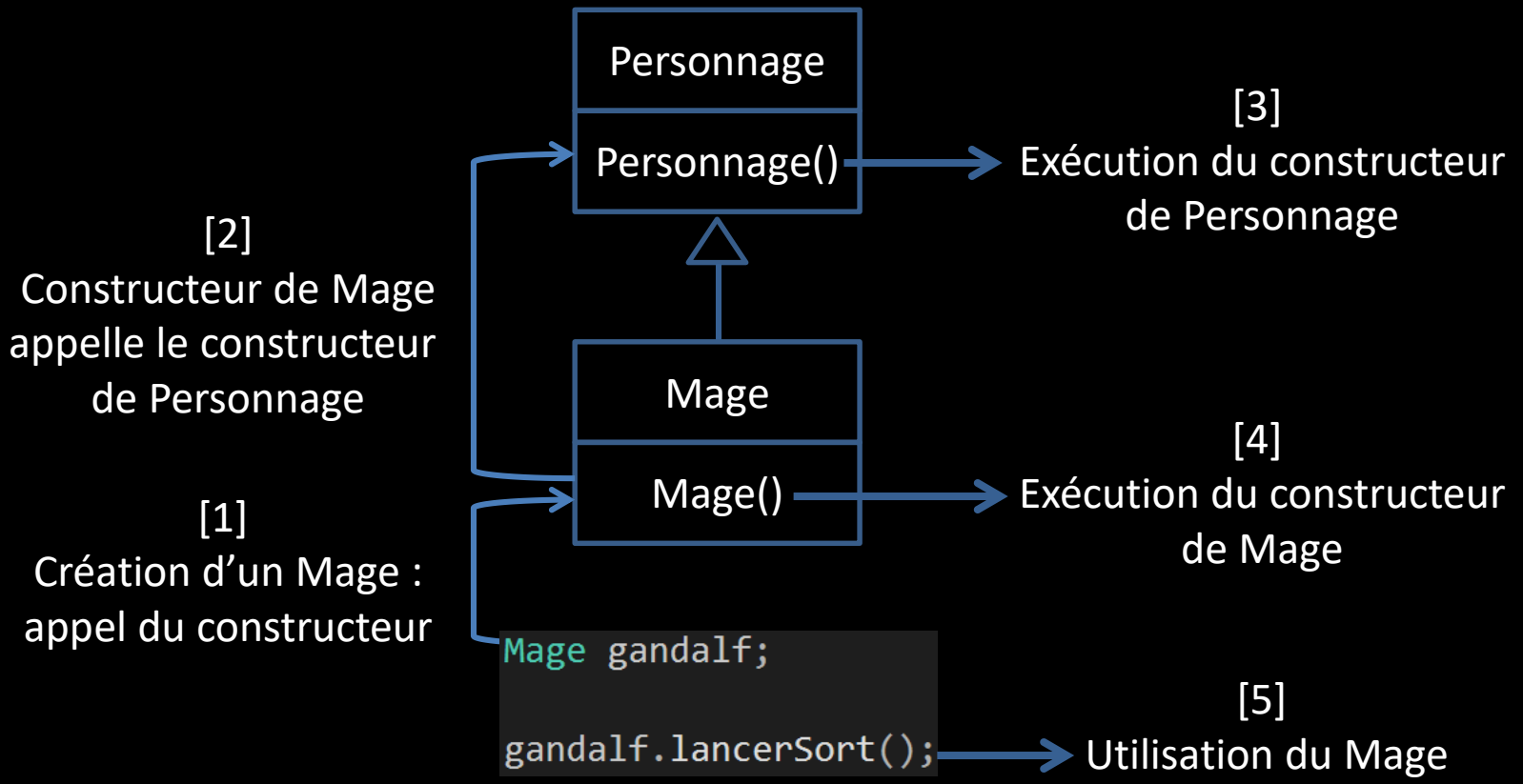
Appels de constructeurs (1)

- Construction fille : appel constructeur mère



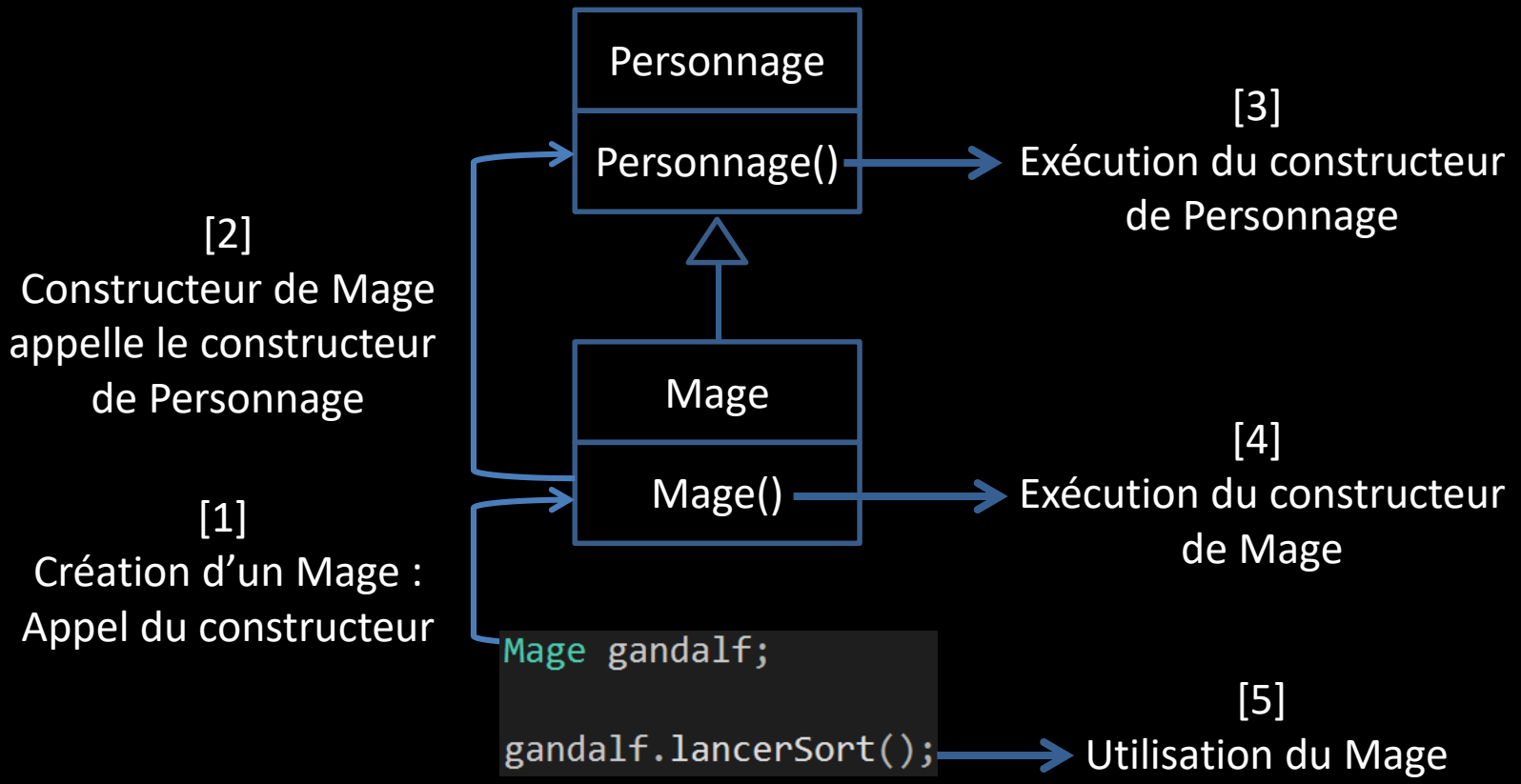
Appels de constructeurs (1)

- Construction fille : appel constructeur mère



Appels de constructeurs (1)

- Construction fille : appel constructeur mère



- Sens inverse pour les destructeurs !

Appels de constructeurs (2)

- Appel explicite
  - Liste d'initialisation !

```
class Personnage
{
public:
    Personnage() : m_nom("John Doe"),
                   m_vie(100),
                   m_attaque(10) {}

    // [...]
private:
    std::string m_nom;
    int m_vie;
    int m_attaque;
};
```

Appel du constructeur de Personnage

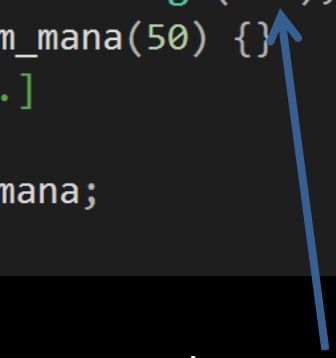
```
class Mage : public Personnage
{
public:
    Mage() : Personnage(),
             m_mana(50) {}

    // [...]
private:
    int m_mana;
};
```

Transmission de paramètres

- Transmettre un ou plusieurs paramètres au constructeur de la classe mère

```cpp
class Personnage
{
public:
    Personnage(const std::string &nom)
        : m_nom(nom),
          m_vie(100),
          m_attaque(10) {}
    // [...]
private:
    std::string m_nom;
    int m_vie;
    int m_attaque;
};
```

```cpp
class Mage : public Personnage
{
public:
    Mage(const std::string &nom)
        : Personnage(nom),
          m_mana(50) {}
    // [...]
private:
    int m_mana;
};
```



Paramètre transmis au constructeur de Personnage

Notions

- Héritage :
  - Rappels
  - L'héritage en C++
  - Héritage et constructeurs
  - Héritage et visibilité
  - Redéfinition de méthodes
- Polymorphisme
- Transtypage (conversions de types)

Visibilité des membres

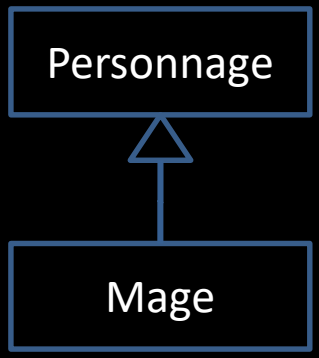
- 3 types de modificateurs d'accès :
  - private : accessible QUE depuis les méthodes de la classe
  - public : accessible partout
  - protected : accessible depuis les méthodes de la classe ET des classe filles

Par défaut !

Visibilité des membres

- 3 types de modificateurs d'accès :
  - private : accessible QUE depuis les méthodes de la classe
  - public : accessible partout
  - protected : accessible depuis les méthodes de la classe ET des classe filles

Par défaut !




```
class Personnage
{
public:
    // [...]

protected:
    std::string m_nom;
    int m_vie;
    int m_attaque;
};
```

Accessible depuis la classe Mage

Visibilité d'héritage

- 3 types (comme pour les membres) :
  - public, protected, private

```
class ClassFille : public ClasseMere
```

- Modifie les droits d'accès des membres des classes filles

Visibilité d'héritage	Visibilité membres classe mère	Visibilité depuis classes filles
public	public	public
	protected	protected
	private	inaccessible
protected	public	protected
	protected	protected
	private	inaccessible
private	public	private
	protected	private
	private	inaccessible

Visibilité d'héritage

- 3 types (comme pour les membres) :
  - public, protected, private

```
class ClassFille : public ClasseMere
```

- Modifie les droits d'accès des membres des classes filles

Le plus généralement utilisé → public

Par défaut ! → private

Visibilité d'héritage	Visibilité membres classe mère	Visibilité depuis classes filles
public	public	public
	protected	protected
	private	inaccessible
protected	public	protected
	protected	protected
	private	inaccessible
private	public	private
	protected	private
	private	inaccessible

Notions

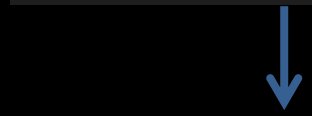
- Héritage :
  - Rappels
  - L'héritage en C++
  - Héritage et constructeurs
  - Héritage et visibilité
  - Redéfinition de méthodes
- Polymorphisme
- Transtypage (conversions de types)

Redéfinition de méthodes

- Modification du comportement pour la classe fille :
  - Même nom, même signature

```cpp
void Personnage::description() const
{
    std::cout << "Nom : " << m_nom << std::endl;
    std::cout << "Vie : " << m_vie << std::endl;
    std::cout << "Attaque : " << m_attaque << std::endl;
}
```

```cpp
Mage gandalf("Gandalf");

gandalf.description();
```



« Nom : Gandalf
Vie : 100
Attaque : 10 »

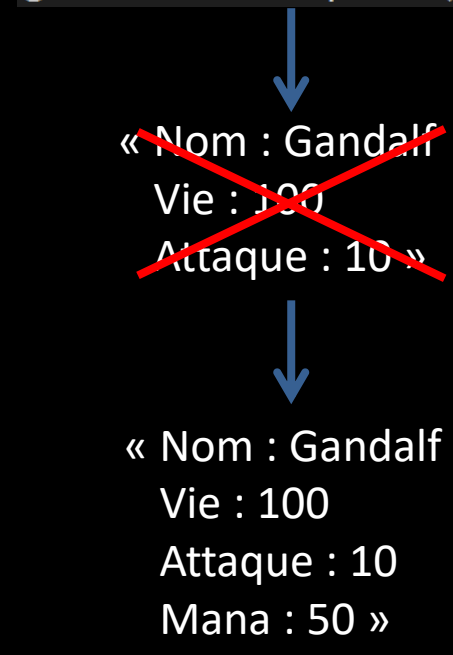
Redéfinition de méthodes

- Modification du comportement pour la classe fille :
  - Même nom, même signature

```cpp
void Personnage::description() const
{
    std::cout << "Nom : " << m_nom << std::endl;
    std::cout << "Vie : " << m_vie << std::endl;
    std::cout << "Attaque : " << m_attaque << std::endl;
}
```

```cpp
void Mage::description() const
{
    std::cout << "Nom : " << m_nom << std::endl;
    std::cout << "Vie : " << m_vie << std::endl;
    std::cout << "Attaque : " << m_attaque << std::endl;
    std::cout << "Mana : " << m_mana << std::endl;
}
```

```cpp
Mage gandalf("Gandalf");

gandalf.description();
```



Factorisation de code

- Réutilisation du code de la classe mère :
  - Facilite la maintenance
  - Moins de code !

```cpp
void Personnage::description() const
{
    std::cout << "Nom : " << m_nom << std::endl;
    std::cout << "Vie : " << m_vie << std::endl;
    std::cout << "Attaque : " << m_attaque << std::endl;
}
```

Réutilisation du code de Personnage

```cpp
void Mage::description() const
{
    Personnage::description();
    std::cout << "Mana : " << m_mana << std::endl;
}
```

```cpp
Mage gandalf("Gandalf");

gandalf.description();
```

« Nom : Gandalf
Vie : 100
Attaque : 10
Mana : 50 »

Notions

- Héritage
- Polymorphisme :
  - Rappels
  - Polymorphisme en C++
  - Utilisation : conteneurs hétérogènes
  - Classes abstraites et fonctions virtuelles pures
- Transtypage (conversions de types)

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

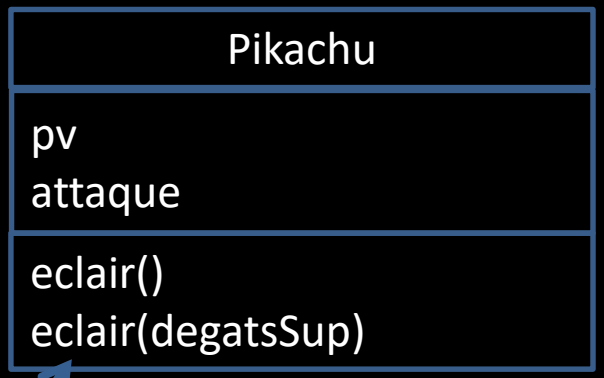
Polymorphisme ad hoc

- 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;
}
```

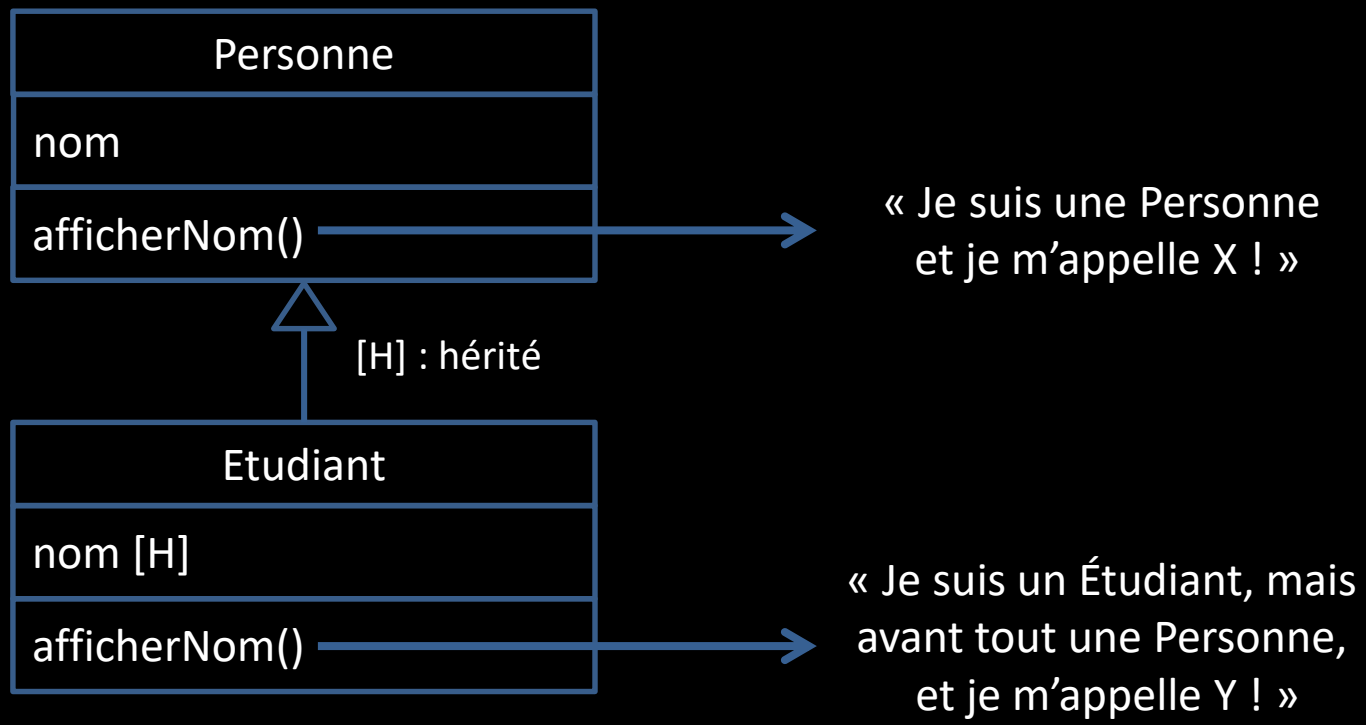
Une méthode



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.
- = <u>Redéfinition</u>



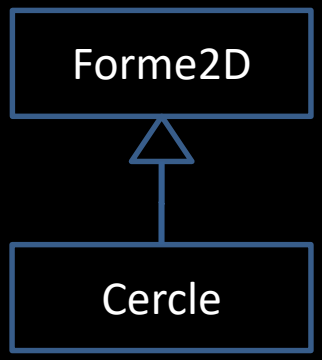
Notions

- Héritage
- Polymorphisme :
  - Rappels
  - Polymorphisme en C++
  - Utilisation : conteneurs hétérogènes
  - Classes abstraites et fonctions virtuelles pures
- Transtypage (conversions de types)

Exemple

```cpp
void Forme2D::description() const
{
    std::cout << "Forme 2D" << std::endl;
}
```

```cpp
void Cercle::description() const
{
    std::cout << "Cercle" << std::endl;
}
```




```cpp
Forme2D forme;
Cercle cercle;

forme.description();
cercle.description();
```

« Forme2D »

« Cercle »

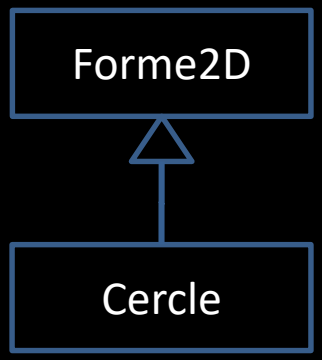
Comportement polymorphe

Problème

```cpp
void Forme2D::description() const
{
    std::cout << "Forme 2D" << std::endl;
}
```

```cpp
void Cercle::description() const
{
    std::cout << "Cercle" << std::endl;
}
```

```cpp
void decrire(const Forme2D f)
{
    f.description();
}
```




```cpp
Forme2D forme;
Cercle cercle;

decrire(forme);
decrire(cercle);
```

« Forme2D »

« Forme2D »

Comportement
NON polymorphe !

Résolution statique des liens
(à la compilation)

Solution : méthode virtuelle

- Résolution dynamique des liens (à l'exécution)
  - Utiliser une méthode virtuelle : mot-clef `virtual`
  - Ajout du mot-clef `override` (C++ 11)
  - Utilisation d'un pointeur ou d'une référence

```cpp
class Forme2D
{
public:
    virtual void description() const;
    // [...]
};
```

```cpp
class Cercle : public Forme2D
{
public:
    void description() const override;
    // [...]
};
```

```cpp
void decrire(const Forme2D &f)
{
    f.description();
}
```

```cpp
Forme2D forme;
Cercle cercle;

decrire(forme);
decrire(cercle);
```

→ « Forme2D »

→ « Cercle »

Comportement polymorphe !

Constructeurs et destructeur

- Ne peuvent pas être hérités (il faut donc les définir)

Constructeurs et destructeur

- Ne peuvent pas être hérités (il faut donc les définir)

- Constructeur virtuel ? IMPOSSIBLE
  - Construction : type connu → Résolution dynamique inutile
  - Pas d'appel à une méthode virtuelle dans le constructeur

Constructeurs et destructeur

- Ne peuvent pas être hérités (il faut donc les définir)

- Constructeur virtuel ? IMPOSSIBLE
  - Construction : type connu → Résolution dynamique inutile
  - Pas d'appel à une méthode virtuelle dans le constructeur

- Destructeur virtuel ?
  - DOIT être virtuel si polymorphisme utilisé

Destructeur virtuel

- But : s'assurer d'appeler le bon destructeur

```
class Forme2D
{
public:
    Forme2D();
    ~Forme2D();
    virtual void description() const;
    // [...]
};
```

```
class Cercle : public Forme2D
{
public:
    Cercle();
    ~Cercle();
    void description() const;
    // [...]
};
```

```
Forme2D *ptrForme = nullptr;

ptrForme = new Cercle;

ptrForme->description();

delete ptrForme;
```

Destructeur virtuel

- But : s'assurer d'appeler le bon destructeur

```
class Forme2D
{
public:
    Forme2D();
    ~Forme2D();
    virtual void description() const;
    // [...]
};
```

```
class Cercle : public Forme2D
{
public:
    Cercle();
    ~Cercle();
    void description() const;
    // [...]
};
```

```
Forme2D *ptrForme = nullptr;

ptrForme = new Cercle;

ptrForme->description();

delete ptrForme;
```

→ Déclaration d'un pointeur nul de type Forme2D

Destructeur virtuel

- But : s'assurer d'appeler le bon destructeur

```
class Forme2D
{
public:
    Forme2D();
    ~Forme2D();
    virtual void description() const;
    // [...]
};
```

```
class Cercle : public Forme2D
{
public:
    Cercle();
    ~Cercle();
    void description() const;
    // [...]
};
```

```
Forme2D *ptrForme = nullptr;

ptrForme = new Cercle;

ptrForme->description();

delete ptrForme;
```

→ Déclaration d'un pointeur nul de type Forme2D

→ Création d'un Cercle, adresse stockée dans pointeur

Destructeur virtuel

- But : s'assurer d'appeler le bon destructeur

```
class Forme2D
{
public:
    Forme2D();
    ~Forme2D();
    virtual void description() const;
    // [...]
};
```

```
class Cercle : public Forme2D
{
public:
    Cercle();
    ~Cercle();
    void description() const;
    // [...]
};
```

```
Forme2D *ptrForme = nullptr;

ptrForme = new Cercle;

ptrForme->description();

delete ptrForme;
```

→ Déclaration d'un pointeur nul de type Forme2D

→ Création d'un Cercle, adresse stockée dans pointeur

→ « Cercle »

Destructeur virtuel

- But : s'assurer d'appeler le bon destructeur

```
class Forme2D
{
public:
    Forme2D();
    ~Forme2D();
    virtual void description() const;
    // [...]
};
```

```
class Cercle : public Forme2D
{
public:
    Cercle();
    ~Cercle();
    void description() const;
    // [...]
};
```

```
Forme2D *ptrForme = nullptr;
ptrForme = new Cercle;
ptrForme->description();
delete ptrForme;
```

→ Déclaration d'un pointeur nul de type Forme2D

→ Création d'un Cercle, adresse stockée dans pointeur

→ « Cercle »

→ Appel du destructeur de Forme2D et non de Cercle !

Destructeur virtuel

- But : s'assurer d'appeler le bon destructeur

```
class Forme2D
{
public:
    Forme2D();
    virtual ~Forme2D();
    virtual void description() const;
    // [...]
};
```

```
class Cercle : public Forme2D
{
public:
    Cercle();
    ~Cercle();
    void description() const;
    // [...]
};
```

```
Forme2D *ptrForme = nullptr;
```
→ Déclaration d'un pointeur nul de type Forme2D

```
ptrForme = new Cercle;
```
→ Création d'un Cercle, adresse stockée dans pointeur

```
ptrForme->description();
```
→ « Cercle »

```
delete ptrForme;
```
→ OK : Appel du destructeur de Cercle

Le mot-clef `final`

- Signification différente en fonction du contexte
  - À la fin de la déclaration d'une fonction virtuelle

```
class ClasseMere
{
    virtual void function() final;
};
```

```
class ClasseFille : public ClasseMere
{
    void function();
};
```

Le mot-clef `final`

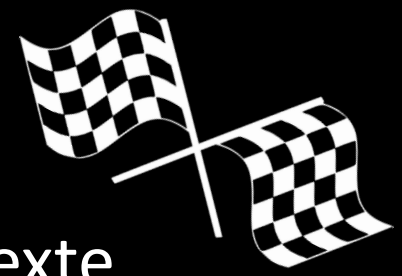
- Signification différente en fonction du contexte
  - À la fin de la déclaration d'une fonction virtuelle

```
class ClasseMere
{
    virtual void function() final;
};
```

```
class ClasseFille : public ClasseMere
{
    void function();
};
```

Redéfinition impossible car `final`

Le mot-clef `final`



- Signification différente en fonction du contexte
  - À la fin de la déclaration d'une fonction virtuelle

```
class ClasseMere
{
    virtual void function() final;
};
```

```
class ClasseFille : public ClasseMere
{
    void function();
};
```

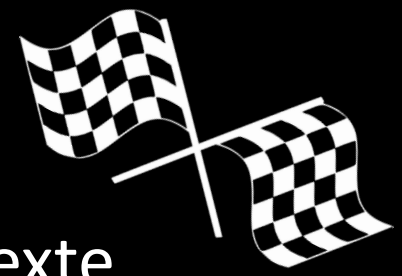
Redéfinition impossible car `final`

  - À la fin de la déclaration d'une classe

```
class ClasseMere final
{
};
```

```
class ClasseFille : public ClasseMere
{
};
```

Le mot-clef `final`



- Signification différente en fonction du contexte
  - À la fin de la déclaration d'une fonction virtuelle

```
class ClasseMere
{
    virtual void function() final;
};
```

```
class ClasseFille : public ClasseMere
{
    void function();
};
```

Redéfinition impossible car `final`

  - À la fin de la déclaration d'une classe

```
class ClasseMere final
{
};
```

```
class ClasseFille : public ClasseMere
{
};
```

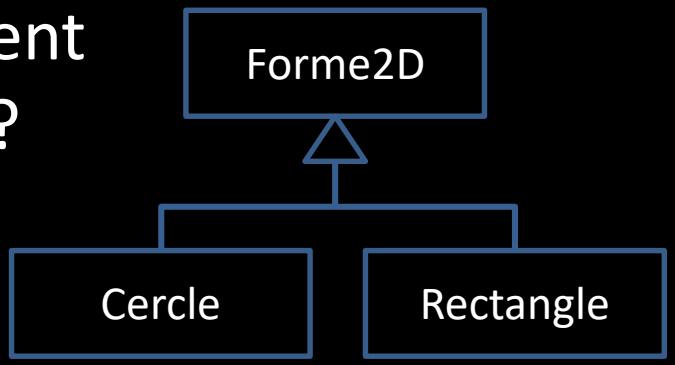
Dérivation impossible car `final`

Notions

- Héritage
- Polymorphisme :
  - Rappels
  - Polymorphisme en C++
  - Utilisation : conteneurs hétérogènes
  - Classes abstraites et fonctions virtuelles pures
- Transtypage (conversions de types)

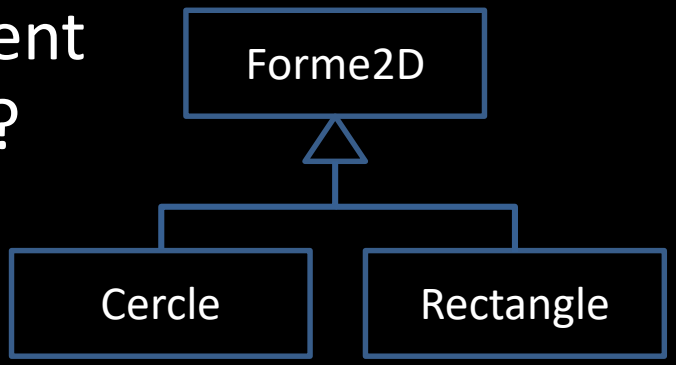
Conteneurs hétérogènes

- Comment faire une liste qui contient des Cercles et/ou des Rectangles ?



Conteneurs hétérogènes

- Comment faire une liste qui contient des Cercles et/ou des Rectangles ?
  - `std::vector` (par exemple)
  - Utilisation de pointeurs !

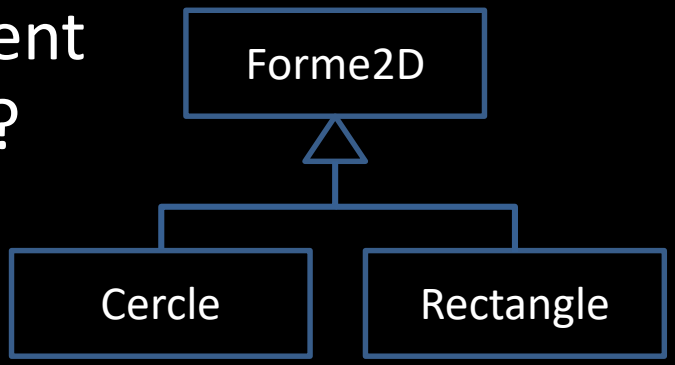



```
std::vector<Forme2D *> formes;
```

→ Déclaration d'un vecteur de pointeurs Forme2D

Conteneurs hétérogènes

- Comment faire une liste qui contient des Cercles et/ou des Rectangles ?
  - `std::vector` (par exemple)
  - Utilisation de pointeurs !



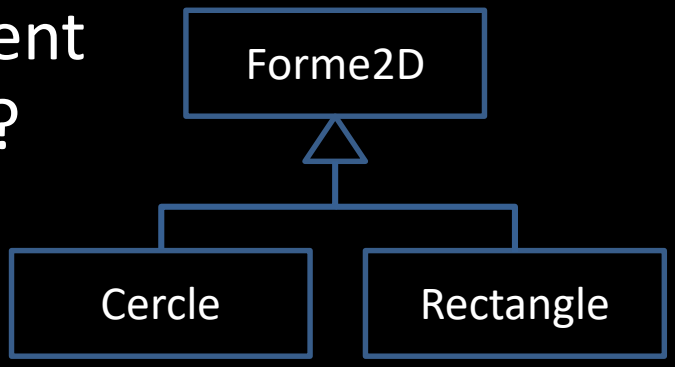

```
std::vector<Forme2D *> formes;

formes.push_back(new Cercle);
formes.push_back(new Rectangle);
formes.push_back(new Cercle);
```

Déclaration d'un vecteur de pointeurs Forme2D

Allocations dynamiques
Pointeurs stockés dans le vector
[Cercle, Rectangle, Cercle]

Conteneurs hétérogènes

- Comment faire une liste qui contient des Cercles et/ou des Rectangles ?
  - `std::vector` (par exemple)
  - Utilisation de pointeurs !




```
std::vector<Forme2D *> formes;

formes.push_back(new Cercle);
formes.push_back(new Rectangle);
formes.push_back(new Cercle);

formes[0]->description();
formes[1]->description();
```

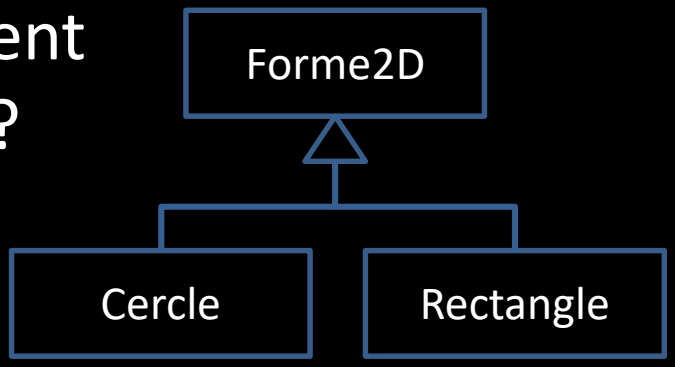
Déclaration d'un vecteur de pointeurs Forme2D

Allocations dynamiques
Pointeurs stockés dans le vector
[Cercle, Rectangle, Cercle]

Appel de la méthode description()
« Cercle » « Rectangle »

Conteneurs hétérogènes

- Comment faire une liste qui contient des Cercles et/ou des Rectangles ?
  - `std::vector` (par exemple)
  - Utilisation de pointeurs !




```
std::vector<Forme2D *> formes;

formes.push_back(new Cercle);
formes.push_back(new Rectangle);
formes.push_back(new Cercle);

formes[0]->description();
formes[1]->description();

for (int i = 0; i < formes.size(); ++i)
{
    delete formes[i];
    formes[i] = nullptr;
}
```

Déclaration d'un vecteur de pointeurs Forme2D

Allocations dynamiques
Pointeurs stockés dans le vector
[Cercle, Rectangle, Cercle]

Appel de la méthode description()
« Cercle » « Rectangle »

`new` → `delete` !

On remet le pointeur à nul (plus propre)

Notions

- Héritage
- Polymorphisme :
  - Rappels
  - Polymorphisme en C++
  - Utilisation : conteneurs hétérogènes
  - Classes abstraites et fonctions virtuelles pures
- Transtypage (conversions de types)

Classes abstraites (rappel)

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

Concepts identiques pour tous les véhicules.

Vehicule [abstraite]
…
demarrer() [virtuelle pure]
accelerer() [virtuelle pure]

Processus non définis pour la classe *Vehicule*. Comportements non concrets.

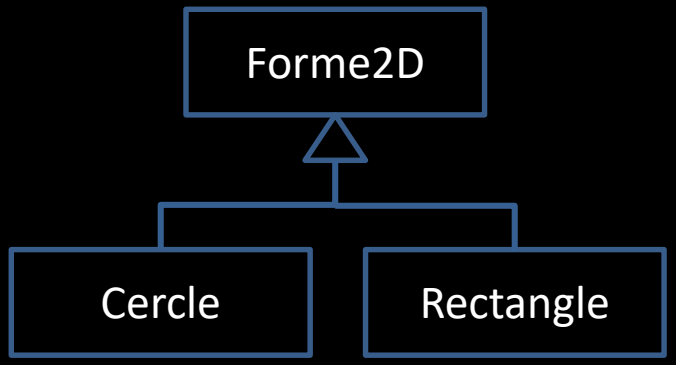
Avion
…
demarrer()
accelerer()

Voiture
…
demarrer()
accelerer()

Processus définis. Comportements concrets.

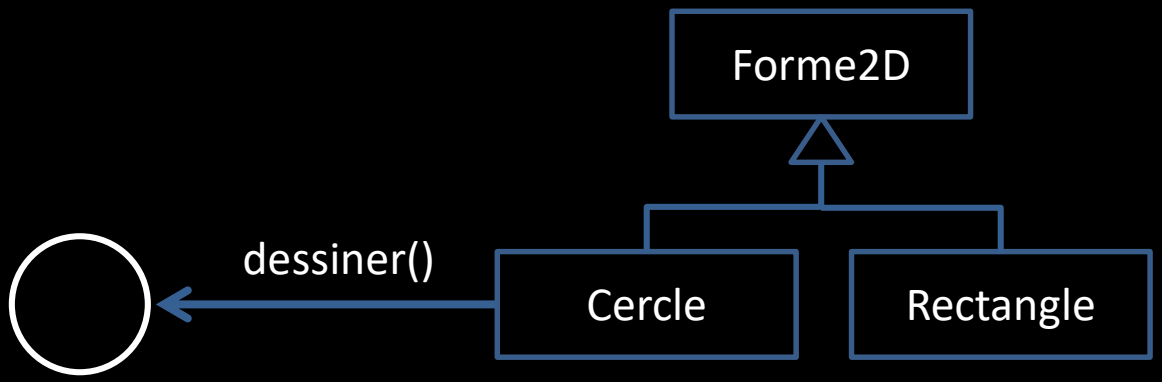
Exemple

- Méthode dessiner()



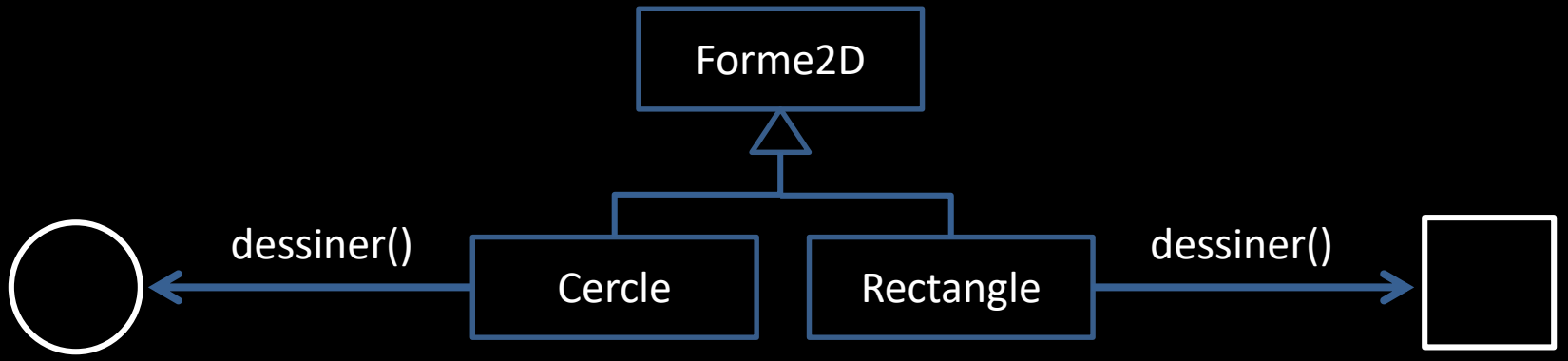
Exemple

- Méthode dessiner()



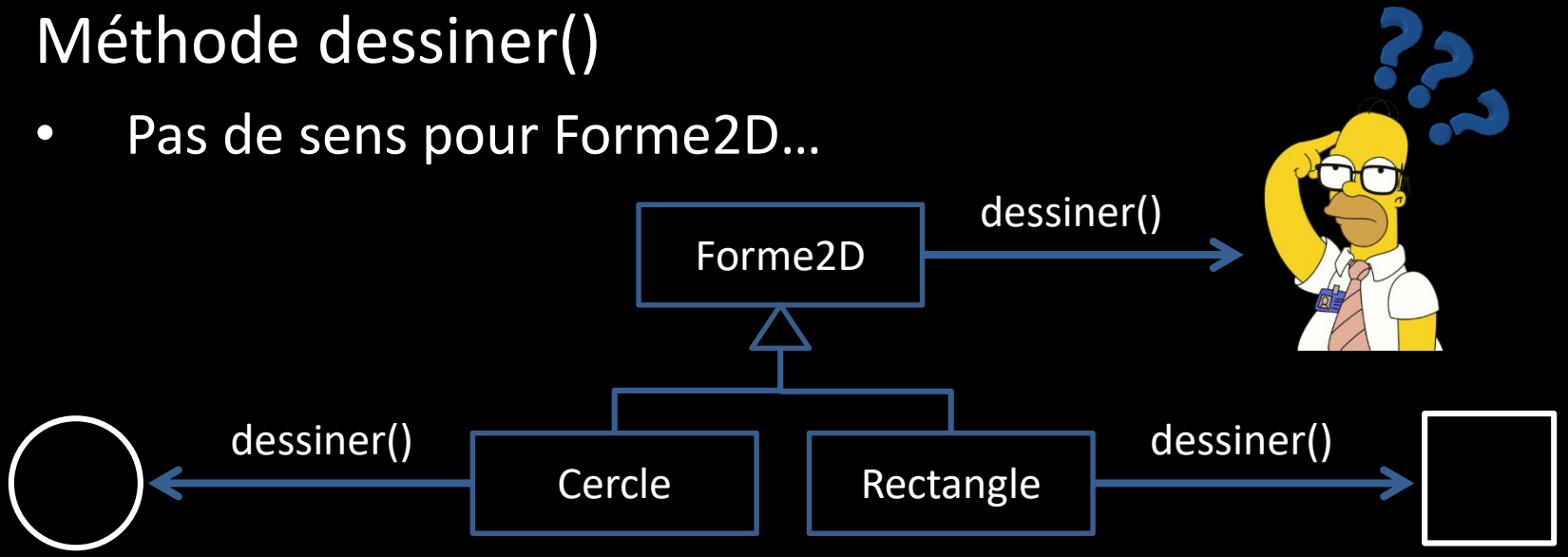
Exemple

- Méthode dessiner()



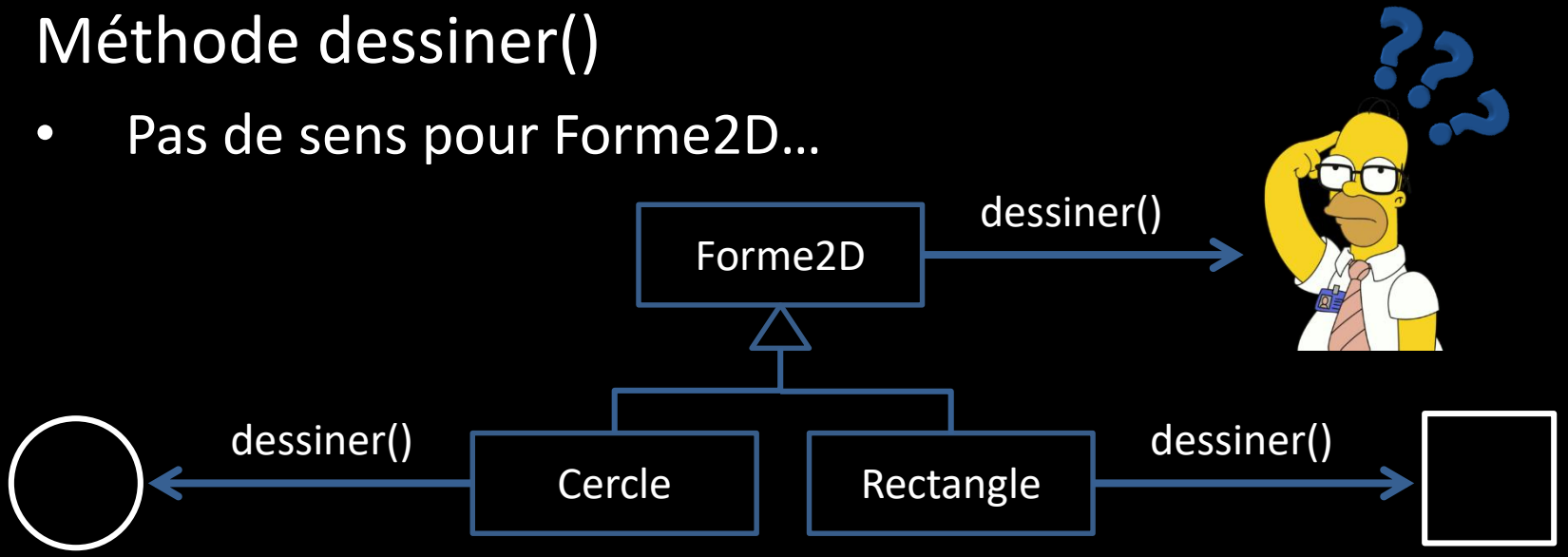
Exemple

- Méthode dessiner()
  - Pas de sens pour Forme2D…



Exemple

- Méthode dessiner()
  - Pas de sens pour Forme2D…



- Donc méthode virtuelle pure
  - Déclaration avec `virtual` ET "`= 0`"

Forme2D est donc abstraite !

```
class Forme2D
{
public:
    virtual void dessiner() const = 0;
    // [...]
};
```

À définir pour les classes filles

Notions

- Héritage
- Polymorphisme
- Transtypage (conversions de types)
  - Un avant-goût seulement… ☺
  - Retour sur le transtypage descendant

Conversion « classique »

- Conversion de type = *cast*

```
int A = 3;
int B = 5;

float C = A / B;

std::cout << C << std::endl;
```

→ « 0 »

En C

```
int A = 3;
int B = 5;

float C = (float)A / B;

std::cout << C << std::endl;
```

→ « 0.6 »

Conversion « classique »

- Conversion de type = *cast*

```
int A = 3;
int B = 5;

float C = A / B;

std::cout << C << std::endl;
```
→ « 0 »

En C

```
int A = 3;
int B = 5;

float C = (float)A / B;

std::cout << C << std::endl;
```
→ « 0.6 »

- Nouvelle syntaxe en C++

```
int A = 3;
int B = 5;

float C = float(A) / B;

std::cout << C << std::endl;
```

Conversion « classique »

- Conversion de type = *cast*

```
int A = 3;
int B = 5;

float C = A / B;

std::cout << C << std::endl;
```
→ « 0 »

En C

```
int A = 3;
int B = 5;

float C = (float)A / B;

std::cout << C << std::endl;
```
→ « 0.6 »

- Nouvelle syntaxe en C++

```
int A = 3;
int B = 5;

float C = float(A) / B;

std::cout << C << std::endl;
```

- On préfèrera utiliser les opérateurs de transtypage

Opérateurs de transtypage

- Buts : Classifier et contrôler les 4 types de conversions
  - `xxxx_cast<`nouveauType`>(`expression`)`

Opérateurs de transtypage

- Buts : Classifier et contrôler les 4 types de conversions
  - `xxxx_cast<`nouveauType`>(`expression`)`

- `static_cast:`
  - Explicite une conversion implicite (e.g. transtypage ascendant)
  - Contrôle au moment de la compilation (pas à l'exécution)

Opérateurs de transtypage

- Buts : Classifier et contrôler les 4 types de conversions
  - `xxxx_cast<`nouveauType`>(`expression`)`

- `static_cast` :
  - Explicite une conversion implicite (e.g. transtypage ascendant)
  - Contrôle au moment de la compilation (pas à l'exécution)
- `const_cast` :
  - Ajout/suppression d'une qualification `const` (ou `volatile`)

Opérateurs de transtypage

- Buts : Classifier et contrôler les 4 types de conversions
  - `xxxx_cast<`nouveauType`>(`expression`)`

- `static_cast` :
  - Explicite une conversion implicite (e.g. transtypage ascendant)
  - Contrôle au moment de la compilation (pas à l'exécution)
- `const_cast` :
  - Ajout/suppression d'une qualification `const` (ou `volatile`)
- `reinterpret_cast` :
  - Conversion de pointeurs non contrôlée (attention !)

Opérateurs de transtypage

- Buts : Classifier et contrôler les 4 types de conversions
  - `xxxx_cast<`nouveauType`>(`expression`)`

- `static_cast`:
  - Explicite une conversion implicite (e.g. transtypage ascendant)
  - Contrôle au moment de la compilation (pas à l'exécution)
- `const_cast`:
  - Ajout/suppression d'une qualification `const` (ou `volatile`)
- `reinterpret_cast`:
  - Conversion de pointeurs non contrôlée (attention !)
- `dynamic_cast`:
  - Spécialisé pour le transtypage descendant

Opérateurs de transtypage

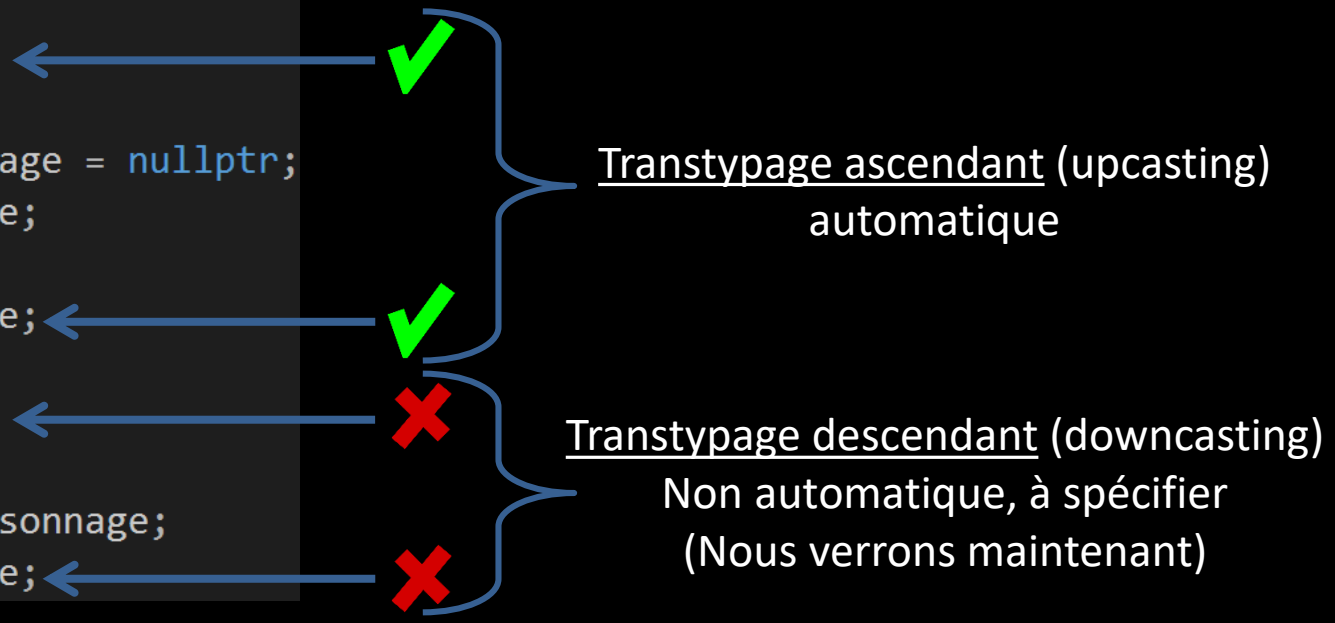
- Buts : Classifier et contrôler les 4 types de conversions
  - `xxxx_cast<`nouveauType`>(`expression`)`

- `static_cast` :
  - Explicite une conversion implicite (e.g. transtypage ascendant)
  - Contrôle au moment de la compilation (pas à l'exécution)
- `const_cast` :
  - Ajout/suppression d'une qualification `const` (ou `volatile`)
- `reinterpret_cast` :
  - Conversion de pointeurs non contrôlée (attention !)
- `dynamic_cast` :
  - Spécialisé pour le transtypage descendant

Transtypage descendant

- Petit rappel…

```
Personnage unPersonnage;
Mage unMage;

unPersonnage = unMage;

Personnage *ptrPersonnage = nullptr;
Mage *ptrMage = &unMage;

ptrPersonnage = ptrMage;

unMage = unPersonnage;

ptrPersonnage = &unPersonnage;
ptrMage = ptrPersonnage;
```



- Solution : `dynamic_cast`

dynamic_cast

- Ne fonctionne que sur les pointeurs ou références

```
Personnage unPersonnage;
Mage unMage;

unPersonnage = unMage;

Personnage *ptrPersonnage = nullptr;
Mage *ptrMage = &unMage;

ptrPersonnage = ptrMage;

unMage = unPersonnage;

ptrPersonnage = &unPersonnage;
ptrMage = ptrPersonnage;
```

```
unMage = *(dynamic_cast<Mage *>(ptrPersonnage));

ptrPersonnage = &unPersonnage;
ptrMage = dynamic_cast<Mage *>(ptrPersonnage);
```

Transtypage descendant
OK !

FIN DU CM !

À vos claviers, c’est l’heure de coder !